WebWriter: Remixing Interactive Learning Resources with Widgets and Explorable Authoring

Von der Fakultät für Informatik der RWTH Aachen University
zur Erlangung des akademischen Grades

eines Doktors der Ingenieurwissenschaften

genehmigte Dissertation

vorgelegt von

Frederic Salmen, M. Ed.

Berichter:
Prof. Dr.-Ing. Ulrik Schroeder

Prof. Dr. rer. nat. Niels Pinkwart

Tag der mündlichen Prüfung: 17.04.2026

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek verfügbar.

Abstract

The reusability paradox is a long-standing problem in educational technology (EdTech), positing that: The more generalized and thus reusable a learning resource is made to be, the less useful and likely to be reused it actually is, since it is decontextualized. To reuse a learning resource, it must be recontextualized by the educator. This recontextualization depends on the form of the resource – while static learning resources such as text or images are easily copied, cropped, and generally modified at any granularity desired, fluid recontextualization is not possible for interactive learning resources without programming. To solve the reusability paradox for interactive learning resources, we present two concepts and a prototypical implementation.

First, we introduce the software design pattern of widgets, which combine packages and components into a new approach on the web, using Node Packages and Web Components. Reuse of widgets is possible in several contexts, such as in other code projects, on websites, or in authoring tools. The design pattern is evaluated in development projects with student EdTech developers (n=11) across 5-7 months, where a qualitative content analysis, interviews and questionnaires showed a high degree of usability and expressiveness for the concept.

Second, we present explorable authoring as a new authoring paradigm which extends rich text editing with the concept of self editing for interactive content (widgets). In self editing, widgets provide affordances to make changes to themselves or their content, and changes triggered by user interactions are processed in the widget module, causing a new document state by modifying attributes or the nested content. This is useful for authoring tools, which can provide traditional rich text editing capabilities with insertable elements, a selection, copying, cutting, and pasting, and allow widgets to be changed by the author's interaction. The authoring paradigm is evaluated in three hour workshops with educators (n=23), where a qualitative content analysis and questionnaires indicate usability and expressiveness.

Both widgets and explorable authoring are implemented in WebWriter. WebWriter is a web app enabling authors to create self-contained web documents, which can be shared over the web or as files. The authoring interface offers a main editor, a palette with insertable elements, a toolbox with configuration options, an app bar for document-specific commands and information, and a separate settings page. The development interface consists of the documentation on the WebWriter website, a gallery with widgets including their source code as examples, and the live package interface in the main application for development and debugging inside WebWriter. In terms of implementation, WebWriter is a single page web app in the MVVM architecture, using TypeScript, Lit, MobX, and ProseMirror as key technologies. Overall, we present WebWriter as a prototypical solution for the reusability paradox.

Contents

1 Introduction **1**
- 1.1 Situated Construction – Context in Learning . . . 2
- 1.2 Escaping the Reusability Paradox - Remix . . . 6
- 1.3 The Turing Tar Pit - Expressiveness & Usability . . . 8
- 1.4 Summary: Research Questions . . . 10

2 Foundations **12**
- 2.1 Parts and Wires – Educational Technology . . . 13
- 2.2 Hyper Medium – The Web . . . 21
- 2.3 Summary: Research Context . . . 28

3 Related Work **29**
- 3.1 Design Patterns for Interactive Elements . . . 30
- 3.2 Authoring Paradigms for Interactive Resources . . . 46
- 3.3 Explorable Authoring Systems . . . 57
- 3.4 Summary: Hypotheses . . . 64

4 Methodology **66**

5 WebWriter **70**
- 5.1 Development Process . . . 71
- 5.2 Requirements . . . 74
- 5.3 Concept . . . 82
- 5.4 Implementation . . . 93
- 5.5 Summary: Prototype . . . 107

6 Widgets **108**
- 6.1 Widget Pattern . . . 109
- 6.2 Examples of Widgets . . . 119
- 6.3 Developer Study . . . 127
- 6.4 Summary: Widgets . . . 134

7 Explorable Authoring **137**
- 7.1 Explorable Authoring Paradigm . . . 138
- 7.2 Examples of Explorable Authoring . . . 145
- 7.3 Author Study . . . 149
- 7.4 Summary: Explorable Authoring . . . 152

8 Conclusion **155**

Appendix **160**

A Instruments and Data **161**

B Own Publications **162**

C Supervised Theses **164**

D Software References **167**

E Literature References **172**

F Glossary **189**

Chapter 1 Introduction

When Educational Technology (EdTech) as a research field was just beginning, David Wiley raised a problem that is still unsolved [Wil+04]: Intentionally generalizing learning resources so they can be reused elsewhere decontextualizes the learning resource, but the value of the resource lies in its contextualization, the degree to which it is fitted to the context. So the more generalized and thus reusable a learning resource is made to be, the less useful and likely to be reused it actually is. Put shortly: *Maximizing reusability minimizes usefulness.* That issue has later become known as the reusability paradox. To solve this problem, Wiley argued, the educator would need to recontextualize the resource. For interactive learning resources, this is not practical to accomplish without programming, excluding most educators from the process. In this dissertation, we propose, implement, and evaluate a possible solution to remix (recontextualize) interactive learning resources to escape the paradox.

Notes on formatting: Important points are highlighted in a box such as this. Software is referenced similar to research literature (e.g. [Goo16]) and can be found in a separate list in the appendix (Appendix D). Margin notes are used to mark sections partially sourced from own publications (see Appendix B). A glossary is provided (Appendix F), and in the digital version, entries are shown as tooltips on mouseover. Footnotes are comments by the authors.[1] A summarizing research summary is shown at the end of each chapter (with new parts in red).

Own publication references look like this

[1] We added this information to help you navigate the text.



Figure 1.1: The process of remixing vs. experts-only remix of technology

1.1 Situated Construction – Context in Learning

To know an object is [...] to lead to it through a context which the world supplies.

William James, 1909 [Jam97]

In learning, context matters: Think about a class in a school setting, for example, involving teachers, students, administrators, and even parents to a degree – how do they relate to each other, and what is happening in their communities? Also, where is this school? There is a culture and a history that has shaped the formal institutions of learning and the informal expectations and values of its members. What is a helpful example in one culture may be puzzling or even offensive in another, such as the term of "race" to categorize groups of people, which is considered appropriate in North America, but highly problematic in many parts of Europe (e.g. Germany). As Wiley put it [Wil+04]: "The simple concatenation or sequencing of decontextualized educational resources will likely fail to produce a meaningful context for learning, necessitating the creation of alternative models [...]".

In this section, we will derive the importance of several types of contexts for learning, supporting Wiley's argument for the high value of contextual learning resources. The contexts are based on the learning theories commonly taught in education.

1.1.1 Immediate Context: Behaviorism

Behaviorism centers around the behavior of the learner, remaining agnostic towards the inner workings of the learner's mind – behaviorists do not try to understand how the mind produces certain behavior, only to predict the behavior produced. Stemming from empirical psychology, theorists like Thorndike [Tho98], Pavlov [Pav06] and Guthrie [Gut30] sought to explain learning as an interplay of stimuli and responses. In B. F. Skinner's operant conditioning [Ski65], features of the environment such as situations or events are antecedents for behaviors. The likelihood of a certain behavior to occur in response can be influenced through consequences. Consequences can either be rewarding, increasing the likelihood (e.g. praising a student's answer), or punishing, decreasing it (e.g. giving a bad grade). Additionally, the withholding of an expected reward has a similar effect to a punishment (e.g. giving the whole class ice cream, except for a misbehaving student) – vice versa for the withholding of an expected punishment (e.g. letting a student off instead of giving detention after a reflective talk). The chain of (antecedent $\rightarrow$ response $\rightarrow$ consequence) is called a three-term contingency. Three-term contingencies of operant conditioning are the basic mechanism of learning in behaviorism. [Sch19]

More complex learning (e.g. playing tennis, see Figure 1.2) is explained with the theory of shaping. Shaping means chaining three-term contingencies such that the correct behavior is gradually approximated (e.g. starting by playing forehand only in tennis). Transfer (or generalization) in behaviorism means that a learner responds similarly between one stimulus and another. The learner's background is only relevant insofar that previous experiences may have reinforced certain behaviors (e.g.

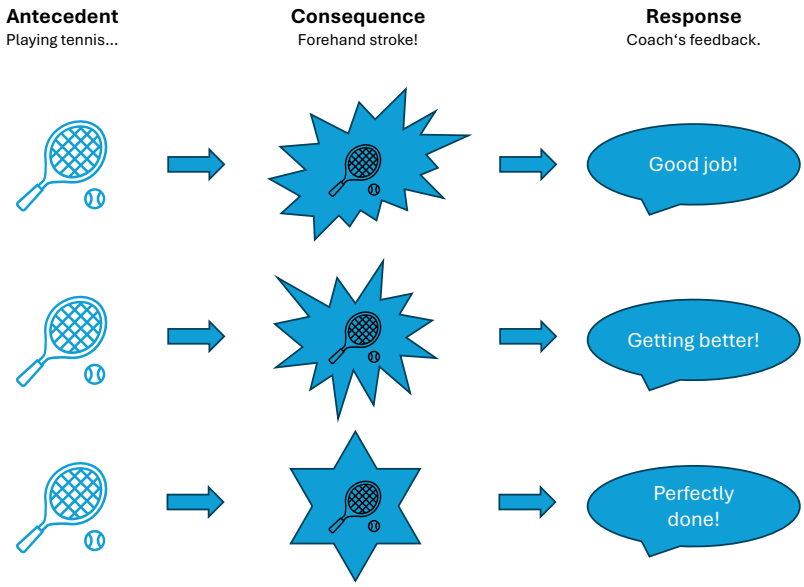


Figure 1.2: Shaping & operant conditioning visualized with the example of tennis, where approximations of correct behavior are rewarded, leading to stepwise improvement.

experience in table tennis leading to difficulties with tennis technique), and that learners must be able to perform the behavior physically. As behaviorism takes an external view, elements such as a memory or motivation are not addressed. Self-regulated learning means choosing between alternative behaviors/actions – this can involve self-reinforcement (rewarding yourself with candy after a hard task), and often deferring short-term rewards in favor of greater long-term rewards (e.g. not going to a party but studying for an exam to pass a class). For instruction, behaviorism favors learning in small steps, involving a lot of practice with tight feedback loops, so shaping can occur. The role of the educator is to break down a given topic into steps with clear, measurable learning objectives, and to deliver rewards or punishments [Sch19].

From behaviorism, we can draw that learning has an immediate context: The situation learning occurs in matters, so educators need to shape the immediate context when designing a learning experience.

1.1.2 Social & Cognitive Context: Cognitivism

Cognitivism applies psychological methods to explain learning, but is differentiated from behaviorism by focusing on processes of cognition rather than on behaviors. There are at least two paradigms within cognitivism: Social cognitive theory focuses on the interpersonal social environments, while information processing theory focuses on personal cognitive aspects, e.g. memory, retrieval, and perception [Sch19].

Albert Bandura is one of the most popular proponents of social cognitive theory. In his theory, knowledge is a symbolic representation that influences actions, and learning can occur enactively (by doing) and vicariously (by observing models and actively reading content). It retains ideas of behaviorism in that Bandura proposes that behaviors with successful consequences are kept and those leading to failure are

discarded. However, it goes beyond behaviorism in focusing on the observation of models – a model presents the consequences of a specific action to a learner, motivating learners to choose specific behaviors. Rather differently than in behaviorism, the learner has agency and can set their own goals based on personal values, which consequences of actions are compared against. Several educational techniques build on social cognitive theory, such as mentoring, worked examples, and tutoring [Sch19].

Information processing theory focuses less on the interpersonal interactions, but rather on the processing of information that occurs in a learner's mind. A common metaphor to understand the theory is to compare the human mind to a computer with an input-process-output loop and several types of memory (working, long-term) for example in the working memory model (see Figure 1.3) of Baddeley et al. [Bad92; Bad00], or Sweller's cognitive load theory [Swe11]. For them, learning is mainly storage in the long-term memory of the learner: Information is sensed, processed by perception which involves existing information in memory, and finally stored. In memory, information can be linked and structured according to schemata. Overall, the theory focuses less on the motivational side of learning compared to social cognitive theory, while social cognitive theory on the other hand does not make statements about how memory works, making both largely compatible. Techniques stemming from information processing theory include advance organizers (so learners can fit information into existing structures), and activities that encourage retrieval [Sch19].

From cognitivism, we can take away that learning has both a social and cognitive context: The social context has a large impact on the motivation of the learner, as it determines their personal values and perceived self-efficacy. Accordingly, learning resources may need to be adjusted to fit the goals of the learners. The cognitive context of existing, memorized structures means that learning resources also need to properly build on existing knowledge, using terms and schemata familiar to learners to make connections and encourage transfer of knowledge [Sch19].

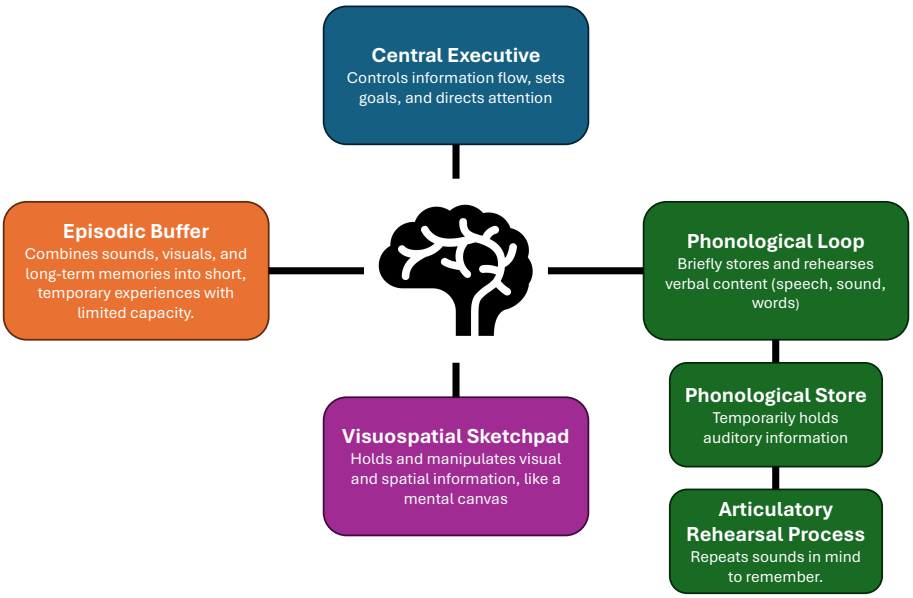


Figure 1.3: Baddeley's working memory model [Bad92] with the revision of the episodic buffer [Bad00]. [Illustration based on [Sta24], used under CC-BY-SA-4.0]

1.1.3 Personal Context: Constructivism

Constructivism has different origins than both behaviorism and cognitivism, as it is more of a learning philosophy. Nonetheless, it has been very influential on modern education. Structurally similar to cognitivist research, there are two streams, one focusing on the interpersonal (Vygotsky's social constructivist views [VC78]), and one on the personal perspective (Piaget's theory of development [Pia76]). In Piaget's tradition, learning is the active construction of knowledge and skills. While similar to the schemata proposed in cognitivism, more of an emphasis is put on personal nature of knowledge: It cannot be transferred from teacher to learner, but must be constructed by the learner based on experience. The process of learning for Piaget is a process of equilibration, where internal, mental structures are progressively made consistent with the external reality. In practice, constructivist teaching emphasizes giving more control of the process to learners, with methods such as discovery learning (providing activities which learners can test hypotheses against) and inquiry teaching [Sch19].

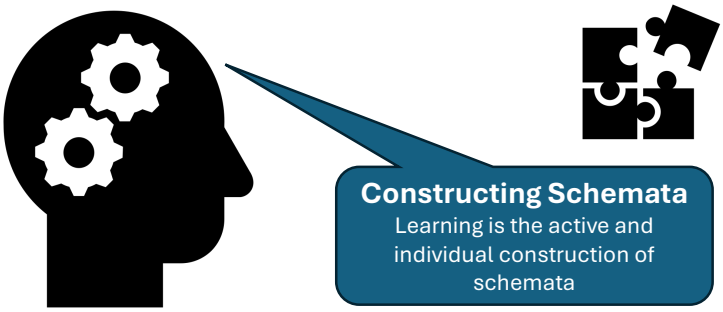


Figure 1.4: Core of the the constructivist theory summarized

Modern learning principles are deeply rooted in constructivism: The principle of active learning [Fre+14], which champions providing learners with activities for learning, rather than presenting information which is passively consumed, is a constructivist idea. Similarly, advocates of personalized learning suggest that learning experiences need to be differentiated by educators, but also customized by learners, emphasizing the active role of learners in the process. Out of constructivism, we draw the idea that learning has a personal context: There can be no "one size fits all" approach to learning resources, as each learner is involved in their unique construction process, meaning the learner should be able to change the learning resource.

Overall, we arrive at a multifaceted view of what high value learning resources need to be. As many different contexts influence learners, educators face the task of considering each context when designing a learning experience.

Principle of Contextual Learning Resources: Many different, overlapping contexts affect learning, such as the immediate (physical), the cognitive, the social, and the personal context – for resources to facilitate learning, they need to be adjust the context of the learners, both by educators and by learners.

1.2 Escaping the Reusability Paradox - Remix

Everything made now is either a replica or a variant of something made a little time ago and so on back without break to the first morning of human time.

George Kubler, 1962 [Kub62]

Let us look again at the shortened statement of Wiley's reusability paradox [Wil+04] we introduced:

Reusability Paradox: Maximizing reusability minimizes usefulness.

Consider an example to better understand it (see Figure 1.5): Suppose a history teacher in a small French town wants to introduce his 6th grade class to the medieval time period. They do an online search to find learning resources, finding many results. There are very general resources about either the medieval period ("learning about knights", generalizing across all of Europe), or about France (e.g. a text about the geography). On the surface, these resources seem highly reusable, since they fit many contexts, but at the same time, that limits their value. They discard these results as they neither fit the age or level of his class, nor the cultural context, as they are interested in showing the specifics of France. They continue their search and find a video about the medieval French Kingdom, produced for children. This could serve as the start of the lesson, but they augment it by formulating some leading questions for students to consider while watching the video, also limiting themselves to a shorter clip of the video. This still leaves open the main part of the lesson, which they want to include some kind of student activity. Unsatisfied with the available material, they ask a colleague, who shares a small quiz about the medieval history of their own town they made some years ago.

Looking at that example, we can see how very generalized learning resources may not see much use in practice. We can also see remix at play for the example of learning video (moderate educational value and potential for reuse): Clipping the video to a shorter length makes it fit the planned lesson's length, and adding leading questions recontextualizes the perception of the video to the specific lesson. From this example, we can infer a more abstract understanding of how educators prepare learning resources for their lessons (for example, a worksheet where each learner receives a copy). To author learning resources, educators take many sources and compose them into one, remixing content through cutting, changing and adding as needed for their group of learners. This is an expert's work, requiring a deep understanding of both the learners, and of the learning process. We can observe that for something to be remixed, it needs to be decomposable. For example, an educator can easily decompose a textbook by cropping or photocopying parts of the book. They are also free to rewrite parts of the text, decomposing at a much lower level. Educators are able

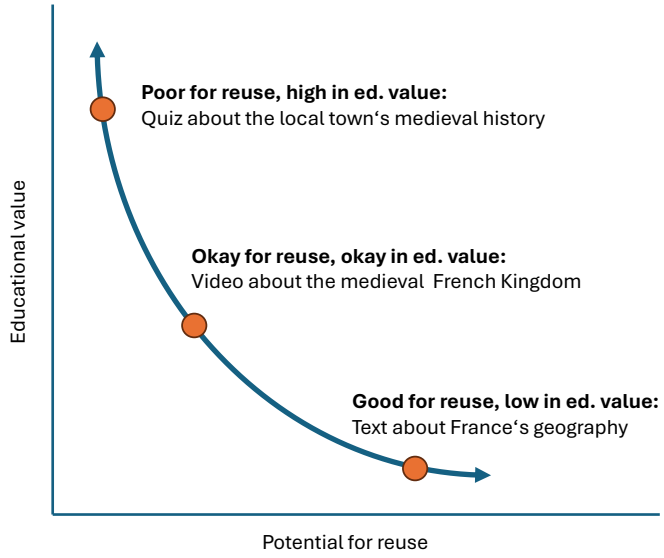


Figure 1.5: Wiley’s reusability paradox, visualized with examples

to freely choose the granularity of reuse – they may use only a single assignment or figure they may like, or the whole textbook across all their lessons.

> **Reuse through Remix**: As learning resource need to fit a context, recontextualization is necessary, which can be accomplished by remixing. Remixing means decomposing sources into parts to recompose them into a new resource.

Perils of Interactivity When talking about reuse and modularity, a common metaphor used is the LEGO brick: There are all kinds of bricks in different shapes and sizes, but they share a common interface and can be plugged together. By consistently adhering to this interface, bricks made decades ago are still compatible. We would argue that today’s educational technologies (EdTech) are less like LEGO bricks, and more like household appliances with parts and wires: Put in the right place by experts and surrounded by an interface (e.g. a toaster with some buttons), they work fine and may even be rewired and adapted (decomposed) by those expert mechanics or tinkerers. Think of a successful EdTech product such as Scratch for block-based programming. Experts, in this case programmers, may use the available API or fork the `Blockly` [Fra+12] repository to rewire Scratch. But educators and learners without that expertise are out of luck, as dealing with this complexity in the digital world fundamentally means programming – remix is possible with source code access, but only to experts (just like parts and wires in an appliance).

Programming forms a hard skill barrier between authoring non-interactive learning resources and interactive learning resources. Right now, educators typically do not program. Rough estimates put the worldwide number of people with any programming skills at 28.7 million in 2024 (around 0.03% of the world’s population) [Vai23].

Among educators, we can expect the number to be a bit higher due to the higher than average level of education and specialization in STEM (science, technology, engineering and mathematics) fields. Still, even if we generously assumed that 1% or even 10% of educators could program, most would need to learn this additional skill. While the goal of everyone learning programming is laudable, and paradigms such as computational thinking [Win06] argue that anyone may benefit through transferrable skills, it remains a time-consuming activity, which is likely not practical to integrate into educators' workflows on top of their other duties. The same argument is easy to apply in reverse: Education is its own, time-consuming specialty which usually requires a degree, not something that is quickly read up on by programmers.

Static-Interactive Divide: In comparison, interactive learning resources are mostly programmed, while non-interactive resources are mostly authored through tools.

We can conclude that tooling is necessary to bridge the divide, so educators are able to remix interactive learning resources without learning programming. Still, we would not argue for a solution which excludes programmers from the process completely: While we can imagine composite learning resources with interactive elements interspersed in them (more on that in Section 2.1), the interactive elements themselves still have to be programmed. Thus, we want to investigate not a complete "no-code" solution, but a hybrid approach where interactive elements programmed by EdTech practitioners can be remixed by educators.

1.3 The Turing Tar Pit - Expressiveness & Usability

Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

Alan Perlis, 1982 [Per82]

We established the reusability paradox and aim to solve it for interactive learning resources in our research. We theorize that through remixing, the paradox can be solved, but that interactive content is programmed, not authored. A solution must then enable the authoring of interactive content, but what makes a good solution? We propose two measures any solution would need to live up to, but are also in a tension with each other: Expressiveness and usability.

Expressiveness One angle to understand expressiveness we could apply is the formal one: As established, interactive elements are a type of software. Software is typically developed using formal languages, such as HTML/CSS/JS (HyperText Markup Language, Cascading Style Sheets, JavaScript) in case of the web. A pattern of implementation is a subset of these formal languages, such as library that developers can use, a documented approach, or something similar. From a rigorous perspective then, expressiveness is Turing completeness – the power of universal computation [Tur37].

There are also more nuanced formalisms of expressiveness such as the Chomsky hierarchy [CS59] or Felleisen's framework to compare the expressive power of programming languages [Fel91]. However, the formal approach to expressiveness is of limited use for our research: As we set the goal to support real developers, we must consider practical limits such as time, motivation, and prior knowledge. Simply put, an implementation pattern for interactive elements may retain Turing completeness, but could still lack practical expressiveness due to the mentioned practical limits.

If we are more interested in practical expressiveness, what is it exactly? This topic is generally considered to be underexplored [WU17], but there are a few ideas we can adapt to gain a better understanding. In 1982, Alan Perlis wrote [Per82]: "Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy." This quote helps us get closer to a proper definition (others [WU17] made similar arguments). Practical expressiveness depends much more on what developers or educators may want to express (in Perlis' words: What is "of interest"). Other domains such as linguistics distinguish between syntactic expressiveness (the set of expressions that can be generated by a language) and semantic expressiveness (the set of unique meanings that can be generated by a language) [BZF05]. With this distinction, we can see that we are less after full syntactic expressiveness (Turing completeness), but more after a limited semantic expressiveness for our domain. Our implementation pattern does not need to be useful for every type of program, but only for interactive elements in the domain of EdTech. Similarly, our method of authoring does not need to support every imaginable document, only interactive learning resources in our domain. We can bring these ideas together in a working definition: Practical expressiveness of a language or approach means that actors of a certain domain (EdTech developers, or educators) can make use of that language or approach to express any product desired (interactive element or learning resource).

Usability While with practical expressiveness, there were no established definitions and we had to make due with a working definition, with usability, there is an abundance of definitions and models to pick from. For instance, there are the definitions by established human-computer interaction (HCI) researchers such as Jakob Nielsen. Nielsen [Nie93] defines usability as "a quality attribute that assesses how easy user interfaces are to use" and identifies five components of learnability, efficiency, memorability, low error rate with easy recovery, and satisfaction. These form the basis of later standardized definitions such as ISO 9241-11:2018 [ISO18]: "[The] extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specific context of use."

Relationship between Expressiveness & Usability There are many other ways of framing expressiveness and usability. For example, Myers et al. suggested a memorable framing for evaluating software tools [MHP00]: "The "threshold" is how difficult it is to learn how to use the system, and the "ceiling" is how much can be done using the system. The most successful current systems seem to be either low threshold and low ceiling, or high threshold and high ceiling. However, it remains an important challenge to find ways to achieve the very desirable outcome of systems with both a

low threshold and a high ceiling at the same time." In our wording, usability determines the threshold or barrier of entry for users, while expressiveness determines the ceiling or limits of the system. Building on that framing, Conlen and Heer examined web-based markup languages for publishing on the two axes of expressiveness and readability [CH18], showing the approach to be viable in the interactive resources space.

We will later discuss how to measure and compare the expressiveness and usability of different systems. For now, we conclude that practical expressiveness and usability are two meaningful qualities which allow us to evaluate technical solutions in the space of remixing learning resources.

1.4 Summary: Research Questions

To summarize our problem: Good learning resources are highly contextual, which limits reuse. In turn, highly generalized learning resources lack context, making them low value and undesirable for reuse. We note that to break out of the paradox, interactive resources are usually remixed, but for the remix to take place, they need to be decomposable.

That multimodal content such as text, images, audio and video is clearly decomposable, as illustrated by the example above. Yet, interactive content including quizzes, simulations, or games poses a challenge as it is programmed, not authored. Thus, decomposition of interactive learning resources requires two distinct types of expertise: Didactical and technical. That puts interactive learning resources out of reach for most educators because they lack the technical expertise to decompose them.

This thesis is intended to help lower the technical expertise required by educators, leading us to the research questions, which are concerned with how to make web-based interactive learning resources decomposable by nature, and how to aid the remix process on the technical side with tooling and standards.

RESEARCH SUMMARY

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

- **RQ_1**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
- **RQ_2**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?

RQ=Research Question

In the next chapter (Chapter 2), we will review foundational concepts, namely in the area of educational & web technologies. After that (Chapter 3), we will form hypotheses for each research question based on related work. In a short chapter (Chapter 4), we lay out our overall research design intended to answer the research questions. We then (Chapter 5) introduce a prototype: WebWriter is an explorable authoring system that considers both the perspective of educators and of developers, for which we will look at the requirements, concept, and implementation. Next, we present our answers to the research questions: First (Chapter 6), widgets are described as a software design pattern for interactive elements that leads to findability, accessibility, interoperability, and reusability (FAIRness) – examples are given and a study with developers is presented. Second (Chapter 7), explorable authoring is introduced, which is an authoring paradigm extending rich text editing with the idea of self editing, which allows for interactive content to become a standard part of a document. Finally, all research results will be summarized and discussed, limitations shown, and future work outlined (Chapter 8).

Chapter 2 Foundations

In this chapter, we establish foundational assumptions for our research. The assumptions form an overall context for our project in which all results should be interpreted. This serves two functions: First, it limits the scope of the research questions. Second, the introduction here can help guide the reader by establishing some basics.

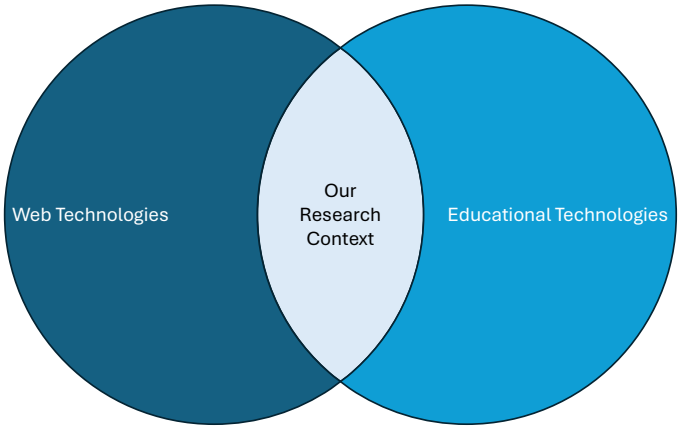


Figure 2.1: The research context of this project visualized

First (Section 2.1), we discuss educational technologies. Next (Section 2.2), we discuss the internet and web technologies. We arrive at the conclusion (Section 2.3) that our research should occur at the intersection of both these domains and introduce some additional practical limitations to further reduce the scope.



Figure 2.2: Educational technology and the internet/web

2.1 Parts and Wires – Educational Technology

Previously (Section 1.2), we likened educational technology to household appliances: While highly useful, usually only experts are able to take them apart and repurpose them. However, just as appliances, EdTech products are innovative and useful. In this section, we will examine how modern EdTech came to be, why it is useful, and why they are so hard to recontextualize.

2.1.1 History of EdTech

Taking a broad, philosophical view, educational technology is of course very old. Education is an act of communication, and any technology aiding communication can become an educational technology: Even a blackboard or a writing slate is an innovation useful for learning. Indeed, writing itself can be seen as the original educational technology. Before Information & Communication Technologies (ICTs), the main medium of education in the western world was the textbook [FB18]. Predecessors in the form of canonised texts did appear all across the ancient and medieval world, made out of papyrus, palm leaves, bamboo, or amate, but the printing press and emergence of written national languages across Europe in the 16th century created the national textbooks as we understand them today [FB18].

Focusing more narrowly on educational technology using ICTs, the field of EdTech is much younger.[1] In the early mainframe era of computing (1940s to 1970s), usage of digital technology for education was very limited, as cost, size, and usability constrained computing to scientists and technicians [LW23]. Rather, the dominant ICTs were radio and later television, the first mass media of national and global reach [LW23]. This world of dominant broadcast media and niche computer use yielded several paradigms of educational technology. Socially, this was also the time of the postwar "baby boom", leading to increased interest in technology to aid educators with overflowing classes. Furthermore, cold war competition fueled education and science in both the west and the east, as exemplified by the "Sputnik Shock" of 1957 in the west, when the USSR launched the world's first satellite into space, ahead of their US rival. Much later, another paradigm was born with the invention of the microchip and microprocessor in the 1970s, which lowered the size and (eventually) cost of computers so that personal computers (PCs) became possible. This meant that educators and learners could have access to digital technology. Each paradigm is not only rooted in these breakthroughs, but also in tge different learning theories (see Section 1.1), changing how new technologies are viewed and employed, and each paradigm also influences today's EdTech world (compare Figure 2.3 for following paragraphs).

Audiovisual Education (1946-1983) Delineated by the invention of the magnetic tape recorder in 1946, when recording audio became much more accessible, the audiovisual education paradigm assumed that auditory and visual enrichment could improve learning outcomes. This built on the precursor paradigm of visual education

[1] This history of the field is only one of many possible perspectives that could be applied. It tries to divide the past into sections, but the edges of these sections are necessarily fuzzy – who is to say when a particular paradigm started or ended? Still, it aims to be broadly true as to which ideas were dominant in the given period, and it should suffice to derive what defines today's EdTech.

(1905-1945), which espoused the value of visual media such as color prints (made possible through high-volume lithography), photography, and later film and television. An added benefit was the broadcasting nature of radio and television, crossing distances and democratizing access to education - without schools or teachers [LW23].

Programmed Technologies (1954-1989) Originating in B. F. Skinner's theories of behaviorism and partly in the ideas of cognitivism (see Section 1.1), programmed technologies put the feedback loop of learning at the center. The observation motivating this approach is that the lecture-and-testing method of teaching is ineffective since feedback is only given once or a few times through test results instead of continuous practice. The earliest example is B.F. Skinner's "teaching machine" based on the theory of reinforcement learning. With this machine, contents were arranged in small steps, and learners would write or select their answer, which was then judged by the machine (giving a reinforcer if correct). Here, the idea of branching learning resources first arose, as Norman Crowder created his own, similar device called the AutoTutor [WC65], which allowed basic individualization simply through learners skipping frames. Variants include Ellson's Programmed Tutoring [Ell68], Lindsley's Precision Teaching [Lin92], Keller's Personalized System of Instruction [Kel74], and Human Performance Technology (HPT). Finally, the ideas of programmed technologies fed into computer-assisted instruction (CAI) [FG95], which an early attempt to apply computers to instruction, first through large-scale projects during the 1970's mainframe era (PLATO [BLE65], TICCIT [MSF80]), later through personal applications in the 1980's [LW23].

Instructional Systems (1967-1983) At the core of instructional systems is the idea of having a formulaic approach to lesson design based on our knowledge of human learning. This goes back to early thinkers such as Pestalozzi (1746-1827) [Hea16] and Herbart (1776-1841) [Ken12], but gained traction in the 1960's, when the Instructional Systems Development project at Michigan State University (1967) published a model and guidelines for so-called instructional systems [Bar67]. The approach frames education as the psychologically rooted design of instructional materials. The most important framework for lesson planning coming out of this movement is: Analyze, design, develop, implement, and evaluate (ADDIE) [Bra+75], referring to the phases of developing instructional systems.

Interactive Multimedia (1977-2000) From cognitivism and by extension programmed technologies, which advocated for a tight feedback loop, and from audiovisual education touting the benefits of audio and video content emerged the idea of interactive multimedia. This new paradigm put immersion in a topic for learner motivation front and center (in contrast to programmed technologies, for example, which sought to remove all distractions from the learning process). 1977 saw the spread of Betamax and VHS videocassette recorders with basic play/pause, fast-forward and "slow-motion" capabilities. [LW23] This is comparable to the programmed technologies ideas of self-paced learning, but would be taken much further, enabled by improving technologies (such as the videodisc) and inspired by the increasing sophistication

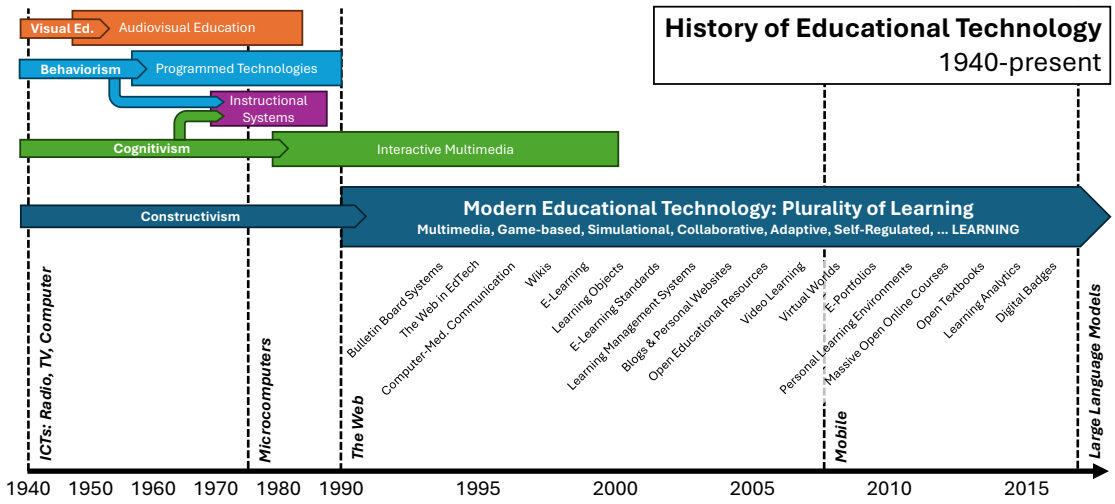


Figure 2.3: History of educational technology in the information age (data: [LW23; Wel20])

of video games (first in arcades and later on personal computers), which came into their own as a medium during this time (think Space Invaders or Pac Man). An early example of educational video games is "The Adventures of Jasper Woodbury", a collection of problem-solving scenarios in mathematics. The 1990's saw the invention of the CD-ROM and DVD, making storage, distribution and playback of interactive multimedia much more accessible and cost-effective. Along with a flourishing commercial video game market, many ground-breaking educational games such as "The Oregon Trail" or "Cluefinders" were published [LW23].

Overall, learning with technologies remained unidirectional and limited to individuals. Three key innovations would change that over time: First, the invention of the internet served to connect educators with learners, starting a first wave of online education through online conferencing. An important technological milestone for this wave is the Electronic Information Exchange System (EIES) of 1978 [Tur78], with early adopters including the Western Behavioral Sciences Institute in La Jolla, California in 1982, and the New School in New York City in 1985. As an early bulletin board implementation, EIES facilitated a social learning environment where participants could post messages including attachments, both in synchronous and asynchronous fashion. The second key innovation is the web. Building on Vannevar Bush's Memex (1945) [Bus45] and Ted Nelson's Xanadu (1960) hypertext concepts [Nel65], Tim Berners-Lee introduced the World Wide Web in 1989 at CERN in Switzerland [Ber89]. The world wide web is based on the hypermedia concept and is defined at the core through a set of interoperating standards: The Hyper Text Markup Language (HTML) and the Hyper Text Transfer Protocol (HTTP), later augmented by Cascading Style Sheets (CSS) and JavaScript (JS). We will elaborate on the web in the next section (Section 2.2), as it is an interesting parallel and often intersecting development to EdTech.

With the three key innovations of the personal computer, the internet, and the web in place, the 90s were an experimental, transitional period for EdTech. The mentioned technologies were on the rise, but were far from the dominance they enjoy today.

The space was dominated by researchers and hobbyists, leading to early innovations including bulletin boards, chat, and wikis which retain their significance as EdTech even today (although, arguably, bulletin boards evolved beyond recognition into social media).

The 2000s finally established EdTech as its own research field, which is marked by the growth of new conferences (ISTE, EC-TEL etc.) and research chairs, and by the rapid formation of new theories around learning with technologies. As we can see in Figure 2.3, EdTech has since been a highly active and innovative space, with frequent new applications (learning management systems, learning analytics, etc.) and emerging technologies in other spaces reforming the field constantly (social media & mobile devices, large language models, etc.).

Modern EdTech mixes ideas of learning psychology, didactics, and technology. It would be difficult to unify the field since then under a single banner. Some researchers suggest a set of modern paradigms such as distributed learning, democratization of media access, inclusion, and emerging technologies [LW23]. A lowest common denominator is the idea of active learning [Fre+14], which suggests that learner-centric activities that require learners to participate actively (e.g. writing an essay) are more effective than passive, receptive forms (e.g. hearing a lecture). This can be understood in the framework of constructivism (see Section 1.1), which suggests that knowledge is not transferred (passive reception) but actively constructed by each learner.

This learner-centric view lends itself to a plurality of learning approaches instead a single, supposedly optimal approach demanded by previous paradigms such as audiovisual education, programmed technologies, instructional systems, and interactive multimedia. These original paradigms are not incompatible with modern EdTech, but live on in modern technologies and theories:

- Audiovisual education's axiom of the benefits of audiovisual content is explained and empirically validated by Mayer's theory of multimedia learning [May20]. Educational videos [Bra16] strongly incorporate multimedia learning ideas, as evidenced by educational YouTube channels such as Crash Course or Khan Academy.
- The behaviorist idea of a tight feedback loop for learners, which may be automatic, today lives on in theories about feedback and more broadly adaptive learning. In EdTech practice, examples include `Khan Academy`'s [Kha08] adaptive task system and `Duolingo`'s [AH11] language learning system.
- The cognitivist idea of instructional systems to optimize learning for the structure of human cognition has developed into the pedagogical approach of instructional design, which, unlike other didactical approaches, is more closely related to educational psychology than to educational philosophy.
- Finally, interactive multimedia paradigm laid the foundations for the fields of serious game [Wou+13] and learning simulation [WAP08]. We can think of PhET's STEM subject simulations as prime examples.

For this section, it suffices to notice that today's constructivist landscape has lead to complex and highly interactive EdTech, a natural consequence of demanding learners take an active role in the learning process.

2.1.2 Taxonomies of Learning Resources

As we aim for expressiveness in our software design pattern and our authoring paradigm (Section 1.3), we should further define what is meant by interactive elements and resources. In research, classification of different interactive content often centers around properties with a unique impact on learning. For example, virtual reality (VR) research forms a cluster at EdTech conferences and in publications, while serious games form another. In practice, both overlap (there are serious games in VR), but for most research questions, this clustering is sensible. For our project however, using such research terms to classify interactive content seems insufficient, since there is no unifying logic behind the assortment of terms – what terms are used is only a snapshot of the current research discourse and may not cover learning resources used in practice. Rather, we want to borrow from several theories in EdTech and media studies for our definitions, which can offer a more complete perspective.

Atomicity: Elements & Resources In our research questions (Section 1.4), we made the distinction between elements and resources. In our context, both are interactive and intended for learning, but while *resources* can be broken down into parts, *elements* by definition cannot be divided further. An example of an interactive element is a physics simulation of a generator (`https://phet.colorado.edu/en/simulations/generator`), while an example of a interactive resource would be a web development book with runnable code examples [Hav19]. Practically, there cannot be "half a simulation", but the book is subdivided into chapters, sections, and finally elements such as interactive code cells and text. However, to muddy the waters a bit, whether something can be subdivided in practice depends on the tools. With source code access and programming expertise, a developer could make a "cut" and create "half a simulation", but since we are viewing the element/resource distinction from the educator's perspective, we can discard this possibility.

Looking at interactive elements, those are programmed using the methods of software engineering. Software can be said to have three distinct facets: The *source*, the *artifact*, and the *runtime*. The software source is usually written higher level abstractions such a general-purpose programming languages and domain-specific technologies, while the artifact is less abstracted and closer to the machine. For example, a serious game may be written in C++ and then compiled to a binary executable or WebAssembly file. The runtime is the system that emerges when the artifact is executed. With some technologies, there is no meaningful distinction between source and artifact, i.e. with interpreted languages such as Python, where the source is run directly by an interpreter.

Intended Audience: Educational Levels and Fields One perspective we can take to understand educational resources in general is that of the intended audience: Which learners is the resource made for? There are two important distinctions institutionalized in almost every educational system across the world: Levels of education and fields of education. Simply put, an interactive element intended for lower secondary learners in a STEM subject (e.g. a simple simulation of photosynthesis) may be quite different from an interactive element element for tertiary learners in the humanities (e.g. a text comprehension and highlighting tool for complex texts). How omnipresent these distinctions are can be seen in the work of the UNESCO Institute for Statistics, which publishes statistics for and comparisons between the different national systems of education worldwide. From their work emerged two standards: The International Standard Classification of Education (ISCED, [UNE12]) distinguishes levels or programs of education, from early childhood education (level 0) to doctorate level programs (level 8). The related standard ISCED Fields (ISCED-F) on the other hand is a taxonomy of different subjects in education, with 11 broad categories (such as "Arts and humanities") and numerous specific topics (the standard lists hundreds of topics and offers detailed descriptions on what does and what does not fit certain categories). Both standards have in common that they can be used at different granularities. For example, we may refer to upper secondary education in general (ISCED 3), or differentiate between general education such as the German *Gymnasiale Oberstufe* (ISCED 34) and the *Berufsschule* (ISCED 35). Classifying elements and resources by educational levels and fields is another axis of expressiveness we consider in our research.

Role: Churchill's Taxonomy For interactive elements which are part of resources especially, Churchill offers a more timeless taxonomy [Chu07], which distinguishes interactive elements (Churchill uses the term "learning object", which we will explore in Section 3.3) by their function or role within a learning experience (a kind of syntax or part-of-speech logic for learning resources). He suggests six types:

- *Presentational objects*: Arrangement of content for a topic (e.g. a slide show)
- *Practical objects*: Exercises with feedback, serious games, etc. that allow practice and learning (e.g. a quiz)
- *Simulational objects*: Representation of some real-life system or process (e.g. a physics sandbox for mechanics)
- *Conceptual objects*: Representation of an abstract concept or multiple related concepts (e.g. `GeoGebra` [Hoh01] for analysis and geometry)
- *Informational objects*: Organized display of information (e.g. a periodic table)
- *Contextual objects*: Live or example data from authentic scenarios (e.g. a historical atlas)

The presented taxonomies are a useful basis for classifying any learning resource, which in turn will help us evaluate the expressiveness of our solutions.

Learning Resource: Any kind of (digital) resource can be used for learning. They can be distinguished by atomicity (elements or resources), intended audience (ISCED educational level and ISCED-F educational field), and role (Churchill's taxonomy).

2.1.3 Multimedia, Interactivity, and Current EdTech

With our research question comes an implicit assumption: That an interactive learning resource, at least one authored by an educator, is beneficial for learning – otherwise, why take the time to create it, at all? While interactivity in itself is a complex topic [Smu09; Str04] and we cannot support the claim that interactive content is always better than non-interactive content, we want to make two more limited claims which support our idea of remixing done by educators. These claims emerge from the modern discourse in educational technology (see below).

Interactivity helps in specific cases. As a first claim, we know that interactive and multimodal content can be highly effective for specific topics. Successful examples include learning videos [Bra16], learning simulations [WAP08; Fin+05], and serious games [Wou+13]. The mentioned research shows that creating interactive content may be worth the effort, as especially complex STEM topics may be made more accessible to learners through interactivity.

Composed, multimedia resources are beneficial. Mayer's Cognitive Theory of Multimedia Learning (CTML) [May20] posits that people learn through two limited-capacity channels (visual–pictorial and auditory–verbal) and that meaningful learning requires active selection, organization, and integration of information. Good multimodal designs distribute processing across channels and support mapping between words and pictures, enabling learners to build coherent mental models – particularly for spatial, causal, or dynamic content that text alone renders abstract. By contrast, non-interactive, text-heavy materials often overload a single channel and make it harder to form integrated representations, reducing transfer. CTML's evidence-based design principles translate this mechanism into practice. When these principles are applied, multimodal materials reliably improve comprehension and transfer relative to text-only presentations, especially for novices tackling complex systems. Benefits are not automatic: misaligned audio–text–graphic combinations can induce redundancy and pacing problems, and advantages diminish with increasing expertise (expertise-reversal).

Both claims together lead us to the argument that multimodal and interactive resources are in fact worth creating, but their value depends on the targeted learner among other contextual factors – educators need to be involved!

Multimedia Axiom: Composed, complex learning resources can be more beneficial for education than disjunct, single-mode content – interactivity may help in specific cases to promote active learning.

Interactive Content From a technological point of view, we can see from our historical overview that learning resources have become more and more capable over time: Each medium encompassing (or merely imitating) the previous one (as suggested by Marshall McLuhan [ML94]), from books containing both text and images, across videos which incorporate images and audio, to the web enveloping everything previously made with the hypermedia concept. We will discuss this idea further in the next chapter when looking at authoring paradigms.

Looking at the current landscape of EdTech in practice, all types of media are likely still in use (from the paper textbook to the learning app). In the world of interactive and connected resources, there is a plethora of different technologies with different emphases: Structured documents [AFQ89] are perhaps the most common type of resource in practice, authored with word processing software such as Microsoft's `Office` [Mic90] suite. This may include worksheets, presentations or other course materials. With the evolution of the web towards multimodal content, self-producing podcasts and learning videos [Bra16] have also gained traction. In STEM, the computational notebook is another form of learning resource, popularized by the `Jupyter` [Jup14] software project but going back to the literate programming concept of Donald E. Knuth [Knu84b]. Also mostly in use in STEM fields, interactive data visualizations can help build data literacy in an increasingly quantitative world.

One especially interesting concept we can build on is that of explorable explanations (explorables) [Vic11a]. This concept by Bret Victor puts an emphasis on active reading [Vic11a]: "An active reader asks questions, considers alternatives, questions assumptions, and even questions the trustworthiness of the author. [...] A typical reading tool, such as a book or website, displays the author's argument, and nothing else. The reader's line of thought remains internal and invisible, vague and speculative. [...] And so, in the end, we blindly trust, or blindly don't, and we miss the deep understanding that comes from dialogue and exploration." Victor suggests three specific measures, which are *reactive documents* (playing with an author's assumptions and seeing the consequences), *explorable examples* (making the abstract concrete in an interactive way), and *contextual information* (allowing the reader to access related material). In practice, explorable explanations mostly follow a classic format, with a running body of text orienting the reader – that body of text is interspersed with interactive content. The concept was elaborated on in the work of Hohman et al. [Hoh+20], which label them "interactive articles", emphasizing use in journalism.

Victor's example implementation is done on the web, which we will explore next (Section 2.2), where we will see how the web platform is a natural fit for interactive documents.

2.2 Hyper Medium – The Web

The previous section (Section 2.1) gave us an idea of the types of content that interactive learning resources may include. In this section, we present the web (and the internet) as the most viable technological basis to express interactive learning resources, and also to remix them.

2.2.1 Beneath the Web: The Internet

To understand both the conceptual roots and the technological basis of the web, we must first look at how the internet developed in the 1970s and 1980s. Today, the internet is a set of interconnected computer networks communicating over the transmissions control protocol / internet protocol (TCP/IP) – a network of networks. It is what makes the web possible, as it is "at once a world-wide broadcasting capability, a mechanism for information dissemination, and a medium for collaboration and interaction between individuals and their computers without regard for geographic location" [Lei+09]. A feeling for the vastness of this network of networks can be gotten from Figure 2.4.

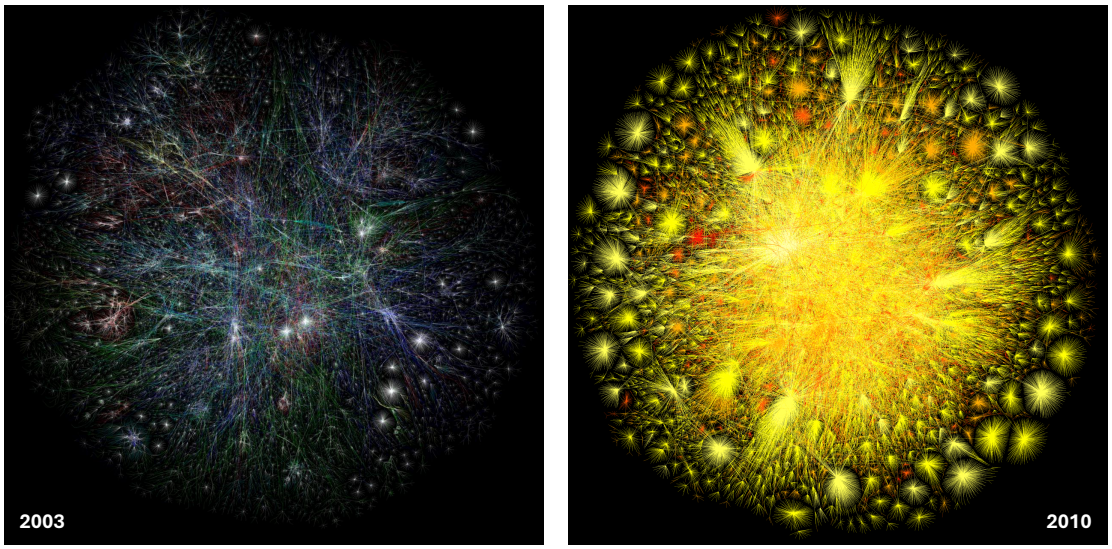


Figure 2.4: The internet visualized as a graph by the OPTE project in 2003 and 2010, with internet hosts as nodes and connections as edges. In the 2003 picture on the left, the colors mark the network region. In the 2010 picture on the right, the color temperature (red to white) expresses the concentration of connections. [Composite of two images from OPTE, CC-BY-NC 4.0]

Going back, much of the foundational early work on the internet as a technology came from the public and military sector. At the US Department of Defense's Advanced Research Project Agency (now DARPA), created in 1958 in response to the "Sputnik Shock", MIT researcher J.C.R. Licklider first proposed the "Galactic Network" concept for globally interconnected computers [LC62]. Leonard Kleinrock then

published the first paper on packet switching, a theory further developed by UK researcher Donald Davies in 1965. This would become the central theory around which the internet protocol suite TCP/IP would be built [Lei+09]. [2]

In 1969, the theoretical work of the pioneers at DARPA came together as the most important early implementation of a computer network – the ARPANET [PS19]. It initially connected four US universities in the western US (UCLA, UCSB and Stanford University in California, and the University of Utah), growing rapidly through the 1970s to include institutions all across the country. Later, there were even connections across the Atlantic, with a satellite link connecting University College London (1971) and the Norwegian Seismic Array (1973) to ARPANET. As a public sector project, this network was used for research and defense. The most significant application running on ARPANET was electronic mail (email), allowing direct, near-instant communication between researchers over any distance. This turned out to be an evergreen application and remains very important in today's internet [Lei+09].

Significantly, the rise of personal computing in the 1970s and the 1980s also meant that, through technologies such as Ethernet (developed at Xerox PARC) [Sch13], local area networks (LANs) could be established. This meant a large number of independently managed networks, somewhat complicating the original idea of "internetworking". To go from net to internet, there needed to be a protocol connecting different network architectures. Bob Kahn and Vint Cerf, important proponents of the ARPANET project, decided to evolve ARPANET's existing protocol, the Network Control Protocol (NCP), into the TCP/IP protocol suite. They applied four design principles [Lei+09]:

- **Architecture-agnosticism**: No changes would have to be made to networks internally to connect them to the Internet, and each distinct network had to work independently.
- **Best-effort**: No assumption of fail-safe packet transmission, rather, packets would be retransmitted if they would not reach their destination.
- **Black box**: No information about packet flows would have to be retained by connecting intermediaries such as gateways and routers.
- **Federation**: No centralized control, instead creating a federation of networks.

These principles made TCP/IP sufficiently fit for real world use: The black box and best-effort principles made the protocol robust in face of failures on the physical layer, such as jamming, interference, and blackouts (to the sponsors of this development, militaries and governments, a highly attractive property). The architecture-agnosticism and federation principles meant that adoption could happen step-by-step, instead of needing worldwide consensus or a central authority asserting power. In

[2] Again, we can only tell one possible narrative of many here – parallel early developments in the UK (NPLNET, X.25), France (CYCLADES), the USSR and the Eastern Bloc (OGAM, Sovam Teleport) are not detailed here. Similarly, we do not look at UUCP/USENET, which developed alongside the internet but later declined. We chose this narrative since the technologies developed for ARPANET would later become the standards of the modern internet, most importantly TCP/IP, turning out to be the most historically significant in retrospect.

this way, the technology perfectly fit the social environment it was conceived in, with the Cold War propelling science and defense concerns, especially in the US. TCP/IP was developed and applied to smaller systems through the 1970s, but introduced to ARPANET proper in 1983. In conjunction, the Domain Name System (DNS) was introduced by Paul Mockapetris, allowing resolution of hierarchical host names into IP addresses, dealing with naming and addressing the many small networks of the ARPANET. [Lei+09]

Parallel to the development in the US public sector, private and grassroots efforts spread much faster globally, such as BITNET ("Because It's There Network"). They showed demand, but were more informal. On a broader scale, one of the first commercial internet offerings that also included actual content was Minitel by France Telecom in 1982 [Ver87], which gave out special-purpose terminals for customers to access services such as online telephone directories and other commercial services, running a kind of predecessor to the web for millions of customers. The European Laboratory for Nuclear Research (CERN from the French acronym) had become an important center of networking, since the amounts of data produced by particle accelerators led to a need for quick and efficient transmission. By the early 1980's, many developed countries had also followed ARPANET's example and built national, state-run networks, including Canada, Germany, Norway, Sweden, Australia, New Zealand, and Japan. In the US, the civilian NSFNET initiative marked a first step towards the commercialization of networks, as the structure of the ARPANET was adopted and its infrastructure shared, offering access to a broader network to smaller institutions and private partners [Abb18].

Through the 1980s and 1990s, TCP/IP was broadly adopted by network builders outside the US. For example, the group European IP Networks (RIPE from the French acronym) created a pan-European network. This was the start of the global growth of the internet. The growth can be characterized by the number of internet hosts in the DNS: An increasing rise through the 1990s (ca. 1.3 million in 1993 to 110 million in 2001), with steady growth and a point of saturation slowly being reached through the 2000s and 2010s, finally stagnating at the start of the 2020s (ca. 730 million in 2010 and 1 billion in 2019). Over time, the 1960s dream of the "Galactic Network" had been realized, creating the developed, global internet we know today [Abb18].

In summary, the internet is the technical foundation for almost every networked application conceived today. Viewed in terms of the OSI layer model, the internet and its infrastructure covers all five lower layers (physical, data link, network, transport, and session), allowing developers of applications to mostly operate on the upper two layers (presentation and application).

Internet Axiom: The Internet is the most relevant system to interoperate interactive learning resources across distances, devices, and different applications.

2.2.2 The Web

As noted, the internet is digital infrastructure, but what could run on it? One of the first broadly successful applications was e-mail, but in the 1980s and 1990s, the growth of the internet meant that another, in fact old concept, might now be implemented at scale: Hypertext.

Hypertext means text that can contain references (hyperlinks) to other texts, which can be instantly accessed through some sort of technical system. After some precursor ideas in literature (Jorge Luis Borges' "The Garden of Forking Paths" [Bor93]) and art (H.G. Wells' "World Brain" [Wel37]), the first proper hypertext system, called the Memex, was proposed by Vannevar Bush in 1945 [Bus45]. He speculated how existing electromechanical microfilm devices could be extended into a system that could show any document from a library, with "trails" connecting documents and even reader annotations captured by a microfilm recorder. While the practical technology turned out differently, we can already extract three principle from this concept that would appear again and again:

1. **Hyperlinks**: Connecting documents through machine-readable references (machine-readable is key, as references as a concept are probably as old as writing itself), which is the defining characteristic of hypertext systems.

2. **Multimedia**: Bush proposed that any document from a library may be read, which implicitly already includes images. As more types of media became common over time (audio, video, etc.), this idea in hypertext became much more broad and was coined "hypermedia".

3. **Interaction**: Even as early as 1945, Bush did not imagine hypertext to be a passive medium, but rather a participatory one, through annotations. As technology continued to develop, especially through the trend towards interactive multimedia in UIs, entertainment and EdTech, interaction could be understood much more broadly.

The 1960s and 1970s saw the formation of proper theory around these principles, with first attempts at implementations. In 1965, philosopher and sociologist Ted Nelson further developed and theorized on Bush's idea, coining the terms of hypertext and hypermedia still in use today [Nel65]. Yet, Nelson's contributions remained largely on the side of theory, as his own, incomplete implementation "Project Xanadu" was published only much later in 1998 [Nel99b]. He however collaborated with researcher Andries van Dam to develop the Hypertext Editing System (HES) [Van88], widely considered the first working prototype of a hypertext system. It could run on commercial equipment (IBM mainframe and terminal computers) and was used in practice for the Apollo Space program's documentation. Notably, it has the first documented inclusion of a "Back" button for hypertext navigation, which has been a staple of browsers since. The HES was later succeeded by the File Retrieval and Editing System (FRESS) [Van88], with improvements to the user interface and better available on commercial hardware. It was also the first implementation of the interaction principle, as content could be edited and annotated. Douglas Engelbart's famous

oN-Line System (NLS) of 1968, the reveal of which was dubbed "The Mother of all Demos", showcased many of the elements of personal computing integrated into a single system (windows, graphics, video conferencing, the mouse, word processing, revision control, etc.), among which was a hypertext implementation [Hin18]. The 1980's largely continued the efforts of the previous two decades by developing more mature solutions aimed at the emergent personal computers. Examples include ENQUIRE by Tim Berners-Lee (1980) [Ber23], Intermedia at Brown University (1984) [Mey86], and HyperCard by Apple Computer (1987) [BT90].

In 1989, internet and hypertext would coalesce into what would become the most significant hypertext system ever made: The web. Researcher and engineer Tim Berners-Lee had become frustrated by how difficult it was to share information between physicists, publishing a memorandum on concept for "a large hypertext database with typed links" [Ber89]. In this web, the hypertextual reference would be an remote link including the network address of another device. He also speculated that audio, video, and images could be included in hypermedia fashion. Outside of his own team, there was little interest in the concept, but Berners-Lee remained undeterred. Encouraged by his manager Mike Sendall and supported by his colleague Robert Cailliau, he specified the three truly essential concepts of the web the next year [Abb18]:

- **URLs**: Uniform resource locators, which are globally unique identifiers for resources on the web building on the DNS system of the internet. This would solve the issue of addressing resources with hyperlinks across network boundaries. It was originally named "universal document identifier (UDI)", but renamed to URL in 1992.

- **HTML**: The HyperText Markup Language (HTML), building on the Standard Generalized Markup Language (SGML) and the Dynatext system. As a markup language, it occupies a middle ground between readability by humans and by machines: Any normal text sequence is valid HTML, but can incrementally be structured using markup, so called "tags", which are then rendered as elements by specialized readers called browsers.

- **HTTP**: The HyperText Transfer Protocol (HTTP), an application-layer protocol on top of the internet's infrastructure, which allows for the exchange of resources based on a request-response scheme, meaning that there are servers providing resources, and clients receiving them (client-server model).

A working browser prototype "WorldWideWeb" and a corresponding server prototype "CERN httpd" enabled the creation of the first website (http://info.cern.ch/). Apart from the major innovation of building on top of the internet, this new "web" was actually less powerful compared to existing hypertext systems. Links were one-way and not typed, and there was neither transclusion nor source tracking. This would inescapably lead to problems, such as link rot (where a linked resource becomes unavailable) – a point of harsh criticism for theorists like Ted Nelson, who said "HTML is precisely what we were trying to PREVENT" [Nel99a]. Similar to the spread of TCP/IP however, this lack of features would prove beneficial for adoption: Implementers of

browsers (web clients) and web servers had far less to worry about, with the first descriptions of HTML, HTTP and URLs being very simple to understand and work with (for example, the first HTML description [Ber91] mentioned only 18 elements, mainly hyperlinks, basic metadata, and some text structure).

The introduction of the first website in 1990 marked the beginning of what would later be called the "Web 1.0" period (roughly to 2004). The Web 1.0 was a web of mostly text-based pages, authored by very few content creators, with the vast majority of users being consumers. Significant events include: First, the formation of the World Wide Web Consortium (W3C) in 1994 led by Tim Berners-Lee, which would work on standardizing web technologies. Second, the rise of the Netscape Mosaic and Microsoft Internet Explorer graphical web browsers, leading to a fierce competition among browser vendors, the so-called "Browser Wars". Browsers rose in importance due to commercialization of the web, in which enthusiasm for the new technology drove massive investment into all kinds of web-related ventures ("dot-com companies"), many of which collapsed during the "dot-com bust". During this period, the web would also pick up a few more of its hallmark technologies. Cascading Style Sheets (CSS), proposed in 1994 and standardized in 1996, split off concerns about presentation from HTML into a separate language. Client-side scripting was also first implemented in 1994 by Brendan Eich, who created the JavaScript (JS) language for the the Mosaic graphical web browser. The language can be summarized as multi-paradigmatic: It combines event-driven, functional, imperative, procedural, and object-oriented elements with a Java-like syntax (just as with the name, coasting off the popularity of Java in the 1990s). Using the trio of HTML for structure, CSS for styling/layout, and JS for interactivity, each as separate parts of the website's architecture, would later be called the "separation of concerns" [Syk17].

After 2004, the web changed its trajectory, evolving into a more participatory medium [Syk17]. Blogs and later `YouTube` [HKC05] pioneered the idea of user-generated content, which was readily picked up and evolved by the early wave of social networks, including Friendster, `Myspace` [DAH03], `Facebook` [Zuc+04], and `Twitter (now X)` [Dor+06]. From a technical point of view, driven by the new and more interactive learning resources of the commercial world, the web evolved more into the direction of platform for applications. This is known as the "Web 2.0", a new vision of the web in which users would create content and engage with each other. For the modern web, it becomes harder to present a single narrative, but we can look at some of the major and recent innovations (similar as in EdTech, see Section 2.1):

- **Multimedia & Interactive Content**: As consumer hardware became more capable each year, and the internet's infrastructure offered much more bandwidth through fiber optic cables and later cellular networks, multimodal content became practical. A 2022 study [AS22] found that, based on empirical data, "[...] the Web is increasingly turning to images, videos and other sophisticated forms of content. Despite these changes, we still find that text dominates in terms of the number of web resources [...]". For example, streaming high-definition video has become a staple across many social media platforms (e.g. `YouTube`, `TikTok` [Byt16]).

- **Mobile**: The introduction of the first iPhone in 2007 started mobile computing, providing many more users to the web (through smartphones and tablets). In terms of technology however, this poses a significant challenge to website creators and even browser developers, as a variety of new operating systems (e.g. Android and iOS), new screen sizes (small and large phones and tablets, different rotations), and even new input methods (touch screens instead of mouse and keyboard) would need to be supported.

The participatory turn of the Web 2.0 meant that there was an increasing commercial interest in offering more and more complex features to consumers through the browser. This provided an incentive for the big players (the FAANG companies of Facebook, Apple, Amazon, Netflix, and Google, also Microsoft) to invest into the development of web browsers and the web platform in general. Especially Google strongly push the idea of web apps nowadays: With `Google Chrome` [Goo08a] (currently the dominant web browser by far), participation in the industry's standard body (the WHATWG), and many open source initiatives (e.g. Angular [Goo16]), the company can wield much influence over the direction of the web. This has allowed Google to develop types of software for the web which were previously thought impossible to realize on the platform, for example the Google `Workspace` [Goo06] apps (competitors to `Microsoft Office` [Mic90]). The new commercial reality is reflected in the standards developed by the WHATWG, in which the interests of all players are balanced. Today's HTML Living Standard [WHA25] is no longer versioned (the last numbered version of HTML being HTML5 in 2014), but continuously extended as features are added. Previously, a lot of a web app's functionality had to be implemented on the server side (e.g., storage or routing) or integrated by plugins (e.g. Flash for multimedia). Today's web standardizes all these features in a single platform, meaning web developers can rely on them to work in a uniform fashion across browsers and to be available without installation or extra steps on the user side.

From the perspective of EdTech, the modern web's potential is vast: A single medium may encompass all the paradigms previously introduced (Section 2.1): Audiovisual content may be included in web pages (although at the beginning, with the hardware limitations of the 90s, this was mostly theoretical). Hyperlinking allows for nonlinear experiences in the spirit of programmed technologies right away, and the later inclusion of scripting would enable automatic feedback. The same scripting can also be used for interaction and combined with the web's audiovisual capabilities, allowing for interactive multimedia. Similarly, instructional systems can be designed for the web. As a decisive benefit, all capabilities are networked over the internet, ignoring distances and adding a collaborative aspect. We will look at modern web development some more when we discuss component systems (Section 3.1), but for now, it suffices to say that the web is arguably the most versatile and popular platform for mixing text-based and interactive content, making it an ideal technological basis for our solution to the reusability paradox for interactive learning resources.

Web Axiom: The web is versatile enough to express both text-based and interactive learning resources.

2.3 Summary: Research Context

In this chapter, we first (Section 2.1) looked at the results of decades of EdTech research: EdTech, while technically complex, can aid learning in the form of interactive content such as games and simulations, by composing multimodal content to leverage the effects of multimedia learning, or by being both adaptive to and customizable by users. Due to the technically diverse nature of EdTech, it is only decomposable through programming, which is of limited use to educators. Next (Section 2.2), we introduced the web as a platform on which a solution for remix can be built.

In theory, the reusability paradox theoretically applies to all learning resources – but research questions encompassing all learning resources and all of EdTech set a scope impossible to empirically investigate. Following the foundations laid in this chapter, we opt to limit ourselves and leave the transfer of the results to future work, should it seem promising. For that reason, we want to delimit our scope as follows: First, we take a *technical and practical approach*. As we will reemphasize when looking at the process of remixing, the challenges surrounding remix are multifaceted. There are legal and organizational questions surrounding this topic which the open educational resources movement (OER) seeks to address. There are also didactical considerations on how to optimally make use of interactive learning resources, the impact on learning outcomes, and so on. Second, we aim at *post-primary learners*, focusing on secondary and tertiary level learning resources, excluding the unique challenges of pre-primary and primary teaching, and assuming literacy and numeracy for learners. Third, we focus on *STEM resources*. A teacher in the field of chemistry may apply vastly different methods of teaching than a Spanish teacher – after all, each field has their own didactics. We choose to focus on STEM subjects, which brings several advantages: Firstly, there are already many EdTech projects in the STEM fields, probably due to the natural adjacency to more technical approaches and solutions. This means that our technical solution can explore supporting and integrating these existing interactive elements. Also, STEM fields lend themselves well to approaches like learning simulations, which the PhET project has shown, for example.

RESEARCH SUMMARY

CT: In the context of a technical solution for web-based, interactive resources to support active learning, aimed at post-primary STEM learners:

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

- **RQ_1**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
- **RQ_2**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?

CT=Context, RQ=Research Question

Chapter 3 Related Work

In this chapter, we relate the current research literature and state-of-the-art implementations to our research questions, arriving at several hypotheses. This is distinct from the research context which we established in the previous chapter, which consists of assumptions we hold since they are well-established or more factual in nature.

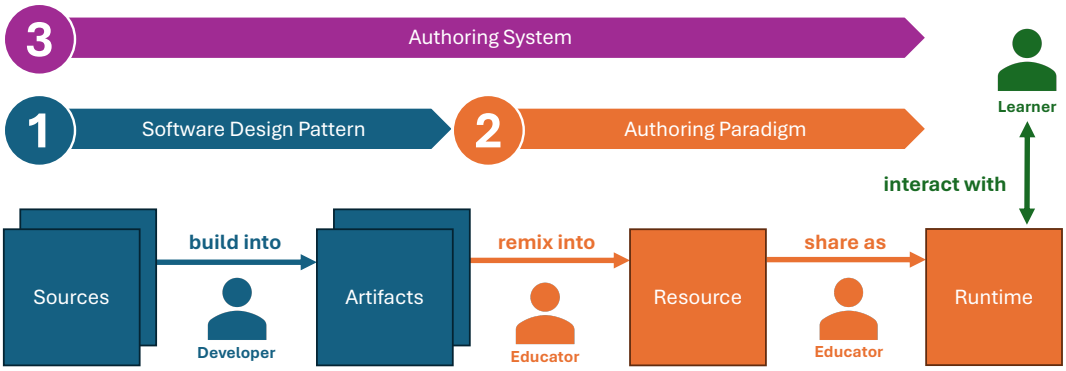


Figure 3.1: The problem space of reusable interactive elements involving remix

The chapter is broken into three sections: First (Section 3.1), we focus on developers, addressing design patterns for interactive elements. Next (Section 3.2), we address educators, covering paradigms for the composition and customization of interactive learning sources. Finally (Section 3.3), we bring together both concepts and go through explorable authoring systems which combine both a design pattern for interactive elements and an authoring paradigm for remixing these elements.

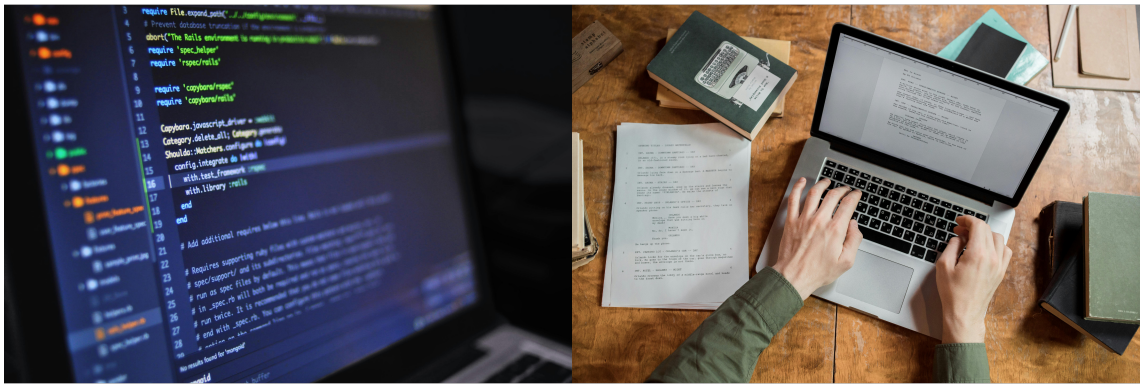


Figure 3.2: Two modes of producing digital content: Programming and authoring

3.1 Design Patterns for Interactive Elements

In the first part of our research (RQ1, see Section 1.4), we want to attain a domain-specific software design pattern that allows developers to create interactive elements in a usable and expressive way which can be remixed by educators.

3.1.1 Interactive Elements and Levels of Reuse

Interactive elements in our scope are parts of digital learning resources which are highly adaptive to the learner's interaction, such as simulations, games, or quizzes (see Section 2.1). This is in opposition to *static elements* such as text or images, where interaction is limited. Of course, this is more of a continuum than a sharp categorization, since for example elements such as audio recordings or videos are more interactive than text, but less so than games. Interactive elements are fundamentally a specific type of software.

With the distinct parts of source, artifact, and runtime (see Section 2.1), we can further distinguish three types of reuse that appear in the literature:

1. **Source-level reuse**: Other developers can import or include the source of the software. For example, an interactive quiz element may be a fork (new project based on another) of an older quiz library. This is also called "white-box reuse" or "grey-box reuse": While there may be available abstractions aiding in the source-level reuse, such as modules or classes, the developer is given full control to "reach inside the box" and change any detail.

2. **Artifact-level reuse**: A software artifact is considered opaque, meaning only parts of it exposed via interfaces can be modified. This is called "black-box reuse". For example, UI components in web development are a black-box reuse idea, where the component's internals are hidden and data is passed to the component via attributes/parameters, while data is emitted with events (i.e. `React` [Wal13]). In EdTech, learning objects are a classic example of the black-box approach.

3. **Runtime-level reuse**: As software is active, rather than passive, it can also communicate with other software during runtime – this is sometimes called interoperating. Normally, this is done via Application Programming Interfaces (APIs), which offer points of access to other software. Prominent examples in the web development domain included the DOM APIs, which allow scripts to manipulate the rendered state of a website, or REST, which is a popular design pattern to offer services via the HTTP protocol. In EdTech, a well-known example is xAPI, which is a protocol that allows EdTech software to record and store so-called statements in databases called learning record stores.

3.1.2 Design Patterns

Design patterns are a measure of abstraction. Abstraction is a way of thinking fundamental to software, and there are a myriad ways of understanding it. One is Krueger's framework [Kru92], in which abstractions are multi-layered: An abstraction is a relationship between a specification and a realization. If we look at more than two layers, one layer may be the specification of the layer below, but the implementation of another layer above. This has been called the "ladder of abstraction" [Vic11b].

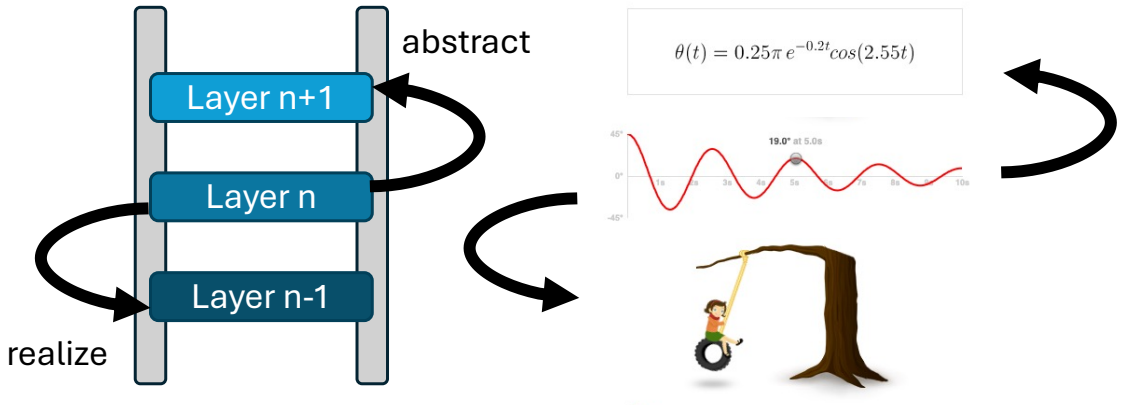


Figure 3.3: Ladder of abstraction metaphor [Vic11b]

As an example of the ladder, consider the program `print("hello world")` in Python. Python source code (`.py`) is an the specification of Python bytecode (`.pyc`). Python bytecode is in turn a specification of machine code. In Python both realization steps ("down the ladder") are done in real time by the Python interpreter. Even further down the ladder, the machine code is executed by the machine. One could continue to the hardware component, physical, and even quantum level. The Python example also shows the value of abstractions: `print("hello world")` is trivial to write for an aspiring programmer. The machine code equivalent is less so, not even mentioning the hardware and physical levels. But what lies further up the ladder? We are intentionally not seeking just a specific implementation of remixable interactive elements. Instead, we are looking for a domain-specific software design pattern, which is a very high level abstraction that may fit into the many platforms and environments of the EdTech world.

Design Patterns The idea of a design pattern goes back to architect Christopher Alexander's 1977 work "A Pattern Language: Towns, Buildings, Construction" [Ale77], which framed architecture and urban design with a language of patterns, which is supposed to help ordinary people (the "end users" of architecture) to participate in the process. We can again see the ladder of abstraction at work in this theory. In Alexander's work, patterns of construction such as "vaulted ceilings" are at the lowest level. Those form building patterns such as "arcades". Finally, buildings compose into town patterns, such as "promenades". In a later work [Ale79], Alexander offers a clear definition of design patterns: "Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution." With the

statement that a pattern depends on the context, Alexander's definition is almost an invitation to transfer his pattern concept beyond architecture. It begs the question: What do design patterns in other contexts look like?

In the 1990s, the design pattern concept experienced a wave of popularity in the software community. Even nowadays, the term "design pattern" is strongly associated with the eponymous 1995 book by Gamma et al [Gam+95]. The book centered around object-oriented programming (OOP) patterns. For the OOP domain, it separates creational patterns (centered around creating objects), structural patterns (for structuring objects in a system), and behavioral patterns (for communication between objects). At the same time, but less widely publicized, the pattern idea was applied to many more parts of software development such as analysis and design [Ker95], or organizational issues [Ser+06]. The idea of finding patterns for specific domains of software also spawned the "Pattern Languages of Programs" (PLOP) conference series [YG23] and its European counterpart EuroPLOP [Plu24]. Recent examples of the pattern idea in specific domains of software include game programming [Nys14] and distributed systems [Bur24].

We can take away that the design pattern idea, especially in Alexander's original structure of context-problem-solution, is a helpful framing for our own research goal. While there are critiques of the idea, it has been widely applied across many different fields of software, so there is a similar potential for EdTech. There have been some scattered attempts at describing all of EdTech in a pattern language [CN23], but we consider this far too ambitious a project. Rather than a pattern language for the whole field, we aim to present a pattern to solve the specific problem of remixing interactive elements (by educators) in EdTech. Starting with a more limited scope seems far more promising, and has been successfully done for some parts of EdTech, e.g. interactive videos [Sei14].

3.1.3 Software Reuse

If we apply Alexander's format of context-problem-solution to our research question, we can see that the context is given by our assumptions established about EdTech and the web from the previous chapter (Chapter 2). As for the problem, we already stated it generally: We want to support remixable interactive elements. To gain a better understanding however, we need to look at the process of reusing software.

The 1968 NATO Software Engineering Conference introduced the term of the "software crisis": The problem of building large-scale software in a reliable way, which was to be the mission of software engineering as a field [Kru92]. As software became more complex, disorganized practices made way for more advanced tools and practices. This is largely a success story: In 2011, Silicon Valley investor Marc Andreessen coined the aphorism "Software is eating the world" [And11]. Software today is used in business, research, entertainment, the military, education.

Software reuse is a core practice and concern of software engineering. With software reuse, we mean "the process of creating software systems from existing software rather than building software from scratch" [Kru92]. There are attempts to formalize software reuse [War93; KF98], but there is no consensus on a single model.

Rather, there are two approaches of understanding software reuse [MMM95]: Either by what is reused (the object), or by how the reuse occurs (the method).

Software reuse can happen through many different activities, such as code scavenging [Kru92] (informal reuse of other code, e.g. copying from StackOverflow), component-based reuse (which is to design parts of software source code for reuse) [McI+68], or even automation (code generation and declarative techniques) [Kru92].

Four Technique Framework Across all activities of software reuse, Biggerstaff and Richter [BR89] identified four techniques of reuse still relevant today: Abstraction, selection, specialization, and integration. *Abstraction* we outlined already above. *Selection* means to pick software artifacts for reuse. Usually, this involves classification and cataloging on the side of the original developer, which in turn allows tools to make software artifacts searchable. *Specialization* means refining a generalized artifact into a more specific one to fit a new context. Typical methods of specialization in software are parameters, transformations, or constraints [Kru92]. Finally, *integration* means combining a collection of software artifacts into a system. Integration can happen through a module interconnection framework that offers import/export functionality. While we consider including "abstraction" as a reuse technique as rather incongruent, the other three techniques are a useful framing still applicable today. For our purposes, we can observe that activities like selection, specialization and integration would need to occur if educators want to reuse interactive elements, so our task is to enable each technique with appropriately designed interactive elements.

FAIR Another interesting perspective on reuse comes from the world of scientific research. Just as in software engineering, reuse is central. It is safe to say that the main mechanism of reuse is referencing another paper, which allows researchers to explicitly connect to other work. Similar to software, this allows for more complex work as researchers can rely on each others theories and results which may encapsulate years of effort. There is also the activity of replication, which means repeating a study to confirm the conclusions reached by the original authors – another instance of reuse. However, there is arguably a blind spot in this framework of reuse: Research data. While reusing specific sections of a paper is fairly standard and well supported with tooling and formats, the data that papers build on is usually less uniform. This makes replication difficult and complicates many kinds of reuse such as running a different analysis on the same data. A useful model to describe what is necessary to make data reusable is the acronym FAIR [Wil+16], meaning "findable, accessible, interoperable, and reusable". Each principle is best understood through the set of requirements originally formulated for FAIR. Note that accessibility is meant literally as access to the data, not in the usual sense of accessibility as inclusive practices regarding disabilities. Also, we employ the term "reuse" to encompass all parts of FAIR, while FAIR associates the term with metadata, licensing and standards only. FAIR is a useful model for three reasons: First, the model does not describe the process of reuse, but rather the properties reusable data needs to have. Clearly, both are highly related, but our own research question is oriented more towards the latter.

Second, through the enumerated principles (F1, F2, etc.), we gain very concrete requirements for a possible design pattern. Third, the FAIR principles are a very recent concept that was suggested with tooling and infrastructure in mind. However, the domain is slightly mismatched: FAIR is intended for research data, not software.

An adaptation of FAIR that considers software is FAIR for Research Software [Chu+22] (FAIR4RS, see Table 3.1). The product of a working group of research software engineers, academics, and practioners, FAIR4RS is a solid model for reusable software. Compared to the original FAIR principles, most are adapted as-is, swapping out "data" for "software" (F2, F3, A1, A2, R1, R2, R3, and R4). Some software-typical concerns are also addressed, such as versioning (F1.2). Other aspects are only rearranged, such as statements about the quality of metadata (F4 vs. I2). There are three substantial differences: First, FAIR4RS introduces an idea of a hierarchy of components inside the software, as it mandates distinct IDs for components (F1.1). Second, it acknowledges that software is an actor onto itself, as the interoperability principles includes the software reading/writing/exchanging data (I1).

F	**Software, and its associated metadata, is easy for both humans and machines to find.**
F1	Software is assigned a globally unique and persistent identifier (ID).
F1.1	Components of software representing levels of granularity are assigned distinct IDs.
F1.2	Different versions of the software are assigned distinct IDs.
F2	Software is described with rich metadata.
F3	Metadata clearly and explicitly include the ID of the software they describe.
F4	Metadata are FAIR, searchable and indexable.
A	**Software, and its metadata, is retrievable via standardized protocols.**
A1	Software is retrievable by its ID using a standardized communications protocol.
A1.1	The protocol is open, free, and universally implementable.
A1.2	The protocol allows for an authentication/authorization procedure, where necessary.
A2	Metadata are accessible, even when the software is no longer available.
I	**Software interoperates with other software by exchanging data and/or metadata, and/or through interaction via APIs, described through standards.**
I1	Software reads/writes/exchanges data with domain-relevant community standards.
I2	Software includes qualified references to other objects.
R	**Software is both usable (can be executed) and reusable (can be understood, modified, built upon, or incorporated into other software).**
R1	Software is described with a plurality of accurate and relevant attributes.
R1.1	Software is given a clear and accessible license.
R1.2	Software is associated with detailed provenance.
R2	Software includes qualified references to other software.
R3	Software meets domain-relevant community standards.

Table 3.1: The FAIR4RS [Chu+22] principles

Overall, FAIR4RS is a promising framework for thinking about reusable software (in the rest of this text, we will use FAIR synonymously with FAIR4RS, as our topic is software – we always mean FAIR4RS). We can map the FAIR principles back onto the

reuse techniques introduced: If software is findable, accessible, and reusable, this enables selection based on metadata, and the software and metadata can be accessed in practice. If it is interoperable, this can enable specialization and integration as the software may offer interfaces such as parameters to modify itself, and there is exchange of standardized data formats. Put shortly: FAIR interactive elements enable selection, specialization, and integration, which allows remix.

Remixable means FAIR: The problem of remixable interactive elements can be framed as fulfilling the FAIR4RS principles, which define reusable software that integrates with tooling.

3.1.4 Informal Reuse Patterns

Informal reuse encompasses all kinds of reuse where the reused abstraction is not explicit, but implicit [Kru92]. For example, a developer may look at the source code of a serious game on an open source platform like Github and copy some part of the implementation into his own, maybe a clever one-liner that solves a programming issue they had. They may also look at some design ideas and use them for their own project, informally reusing a higher level abstraction. There was no ready-made abstraction to be reused: The developers of the original repository did not offer a library, and they did not present an architecture overview document for reuse – still, reuse occurred. The long-standing and successful open source movement, propagating a "bazaar" rather than a "cathedral" approach to software development [Ray99], enables large-scale informal reuse. Q&A Platforms such as StackOverflow are an example of semi-formal reuse: Problems are abstracted as questions, and answers are somewhat generalized from specific contexts. Such platforms have proven highly popular in software development [Wu+19; Che+24; YHL16].

However, informal reuse is not useful to our inquiry, since we assume that the actors reusing our interactive elements are not developers, meaning they cannot reuse code itself. Rather, we need a more formalized pattern of reuse which is compatible with tooling. For this, we can look at some approaches specific to EdTech and connect them back to the wider world of software engineering.

3.1.5 The Package Pattern

Previously, we observed that the principles of findability, accessibility, and reusability (FAR) are closely related, as they allow for the selection of interactive elements. The package *pattern* (see Figure 3.4) is one design pattern appearing in many contexts which may cover the FAR principles (though not interoperability).

Package Pattern: A package is a collection of different resources, hierarchically organized and with metadata attached, requiring certain other packages as dependencies. A package has a unique name and version, which is enforced by a registry, through which packages can be accessed and dependencies resolved. Packages can be accessed either for reuse by other developers, or by end users.

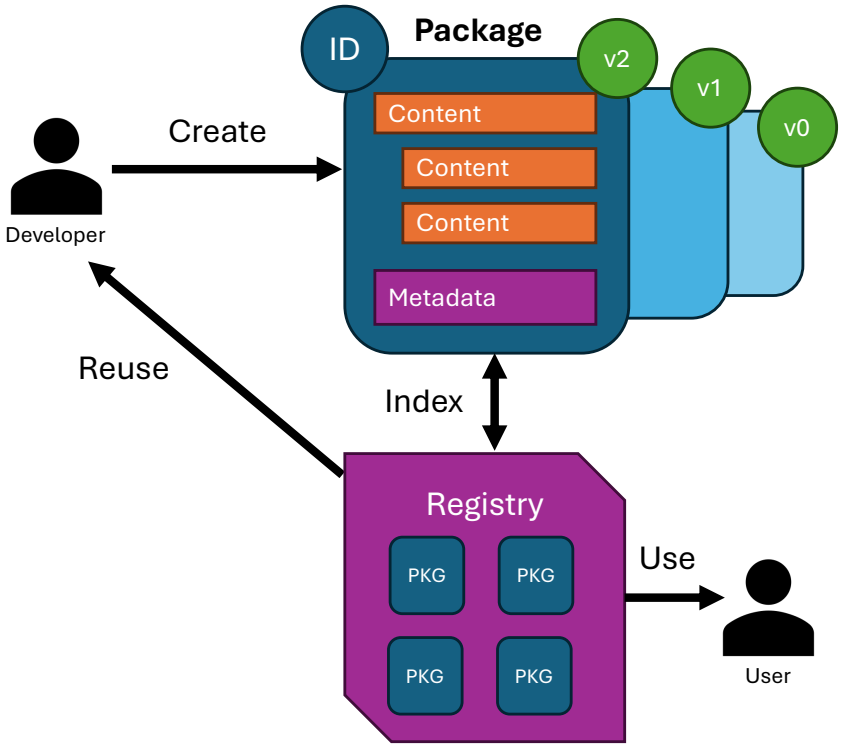


Figure 3.4: The package design pattern

Packages are typically associated with package management systems (PMS). A PMS consists of a registry which stores and indexes packages, and a package manager, which helps manage packages (install, update, remove, etc.) and resolve dependencies. Registries and package managers are decoupled, allowing different package managers to provide access to the same registry, for example.

In theory, packages have several defining characteristics:

- **Naming**: A core part of the package pattern is the idea of unique and persistent naming to avoid name collisions. We can easily see how name collisions (or non-unique names) hamper reuse: How can we depend on other software and have tools download and manage it if they cannot be found? There are many solutions which satisfy uniqueness, such as using UUIDs, but keeping names human-readable is also a priority. Furthermore, we may want to express hierarchies with the name, where components also have unique names that relate to the main entity.

- **Versioning**: As software develops and evolves, there arises a need to not only uniquely identify the software as a whole, but also each step over time – versioning. Put succinctly, a version "represents a state of an evolving item" [CW98]. Software can also depend on other software, meaning that from our perspective of reuse, software should be versioned to enable reuse. For example, a web-based learning simulation may depend upon a specific version of a game engine. As the version is a snapshot of the game engine at a certain point of development, the developer can be sure that certain features will be available.
- **Dependencies**: By specifying dependency relationships precisely with explicit names and versions, the package pattern makes it easy for tools to resolve complex dependency chains. This includes even transitive dependencies (package A depending on B, which depends on C).

Related Concepts to Packages There are a few related concepts to the package pattern. On the source reuse side, there is the code repository. Version control systems such as the historical RCS [Tic85], Subversion [Pil04], or the popular Git [LM09] focus on managing the different diverging and converging versions created by teams of developers. Version control systems offer a sophisticated branching model of reuse and focus on recording text-based changes with high granularity. In web and enterprise development, content repositories are a common part of content management systems (CMS), which help manage digital content (i.e. `WordPress` [LM03]).

Notable Package Systems Next, we will look at a few instances of the package pattern in the real world and frame them in terms of our list of FAIR principles (compare Table 3.1)). Aside from better understanding the pattern, this may also inform our own implementation later. As a detailed example, we will look at the largest package repository in the world [Die+19], which is NPM, housing Node packages for front end and back end web development. But first, we broadly survey historical and existing package systems in different domains. The different package management systems introduced in this part are summarized in Table 3.2.

System	Ecosystem	Year
UNIX Package	PMS/Homebrew	1994
Programming Language Package	PIP/RubyGems/CPAN/CRAN/Cargo/Hex/Packagist/Pub/Haxelib	1995
Game Package	Steam/Uplay/Xbox	2003
Java Module	Maven/Gradle	2004
.NET Package	NuGet	2010
NodeJS/NPM Package	Node, Bundlers, Package Managers	2010

Table 3.2: Overview of common types of package systems within different ecosystems, sorted by year of the pioneering technology

There are some precursor technologies such as IBM's SMIT in 1989 or the Comprehensive TeX Archive Network (CTAN) in 1992, but those lacked a key feature of our pattern: Dependency resolution. The first true package management system according to our definition was the Comprehensive Perl Archive Network (CPAN) in 1995. Inspired by the earlier CTAN, it intended to index and distribute Perl modules

for reuse by other Perl programmers. We can see that packages in this context are a wrapper for reusable modules which can be imported for source-level reuse. Many other programming language communities followed suit with similar tools (PIP for Python, RubyGems for Ruby, CRAN for R, Packagist for PHP, Hex for Erlang, Rust's Cargo, Pub for Dart, Haxelib for Haxe, and many more). There are some ecosystems such as the Java Virtual Machine (JVM) and the .NET Framework which offer cross-language packages (through Maven/Gradle and NuGet, respectively). Nowadays, using external dependencies through packages can be considered the default way of programming, as this kind of reuse enables much more complex applications "on the shoulders of giants". Packages also became relevant in the world of end users through system package managers (APK) and later through app stores (APK for Android and Apple's App Store). A special case are video games, which often use their own package management systems with attached distribution platforms such as Steam, UPlay, or XBOX. In the commercial application development world, app stores and conventional package managers of operating systems are two other examples of the pattern. All these solutions show the widespread nature and broad applicability of the pattern, which means it may also be useful to our research.

Node Packages Node packages originate from the server-side ecosystem of NodeJS, which is a popular runtime for JS. There is no true standard, but the implementation in NodeJS and the related documentation are the de-facto standard used [Fou25]. Simply put, a Node package is a directory of files with a `package.json` file that conforms to the de-facto standard. The `package.json` is rather simple, with only the `name` and `version` fields being required. Other parts of the `package.json` revolve around interoperability of packages, as module `exports` can be defined, as well as the module `type` (standardized ES modules or legacy CommonJS modules). Extra fields not explicitly defined by the de-facto standard are allowed.

This possibility of extra fields is used by the Node Package Manager (NPM) registry, extending the format with many optional properties which we will look at later. The NPM registry is remarkable in that it is the largest package repository (3.6 million packages) and the most used one (370 billion downloads in a single month) in the world, spanning both front end and back end web development. Accessing and even publishing packages is free and unlimited (within terms and conditions), with only private package as a paid feature as of 2025.

Around Node packages, a vast ecosystem of reuse has sprung up: NPM serves as the central backbone of the ecosystem where packages can be published and accessed. Through the `npm` command line utility, Node packages can be installed on NodeJS servers easily, allowing for source-level and artifact-level reuse on the server side. For example, a Node project may want to offer a web server capabilities, for which the `express` package can be used. There is also source-level reuse on the client side through bundlers such as `webpack` or `esbuild`, which allow developers to use JS source files provided through Node packages as dependencies, which are resolved and bundled into a single JS artifact for reasons of size and performance. Finally, there are also content delivery networks (CDNs) such as `jsdelivr`, which mirror the packages hosted on NPM and allow granular access to files in packages,

enabling artifact-level reuse on the client side. For example, a website could include an analytics package via a CDN which then analyzes traffic to the page.

3.1.6 The Component Pattern

Packages are a promising pattern to cover findability, accessibility, and reusability. The "package" metaphor is spot on: They are a digital box with a standard shape and size, and a label/manifest describing what is inside. Clearly, though, this does not address interoperability at all: Once the educator has found, accessed and reused a bunch of packages, how do they put the content together? Or from the developer's perspective: How does the package content have to be structured so that educators can put them together, providing interoperability? A possible answer is the component pattern [SGM02].

> **Component Pattern**: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties.

This definition breaks down into several parts (compare Figure 3.5):

- **Unit of composition & subject to composition by third parties**: This expresses the intention of software reuse, which we discussed above. Specifically, in the four techniques model [BR89], it is the integration of components into a new system. Reuse through composition had long been pursued in software – possibly the first suggestion of this idea in the software field occurred in 1968 [McI+68] under the label of "mass-produced software components". The "mass-produced" label evokes industrialization, and the cited essay indeed suggests that at the time, the software industry was not actually an industry – more of an association of craftspeople creating software with "backwards techniques" [McI+68]. As a solution, McIlroy suggests to fabricate and sell software components, likening the idea to integrated circuits in electronics. This has continued to live on as the idea of component markets, of which many became popular around the 2000s, mainly in the enterprise software world [SGM02].
- **Explicit context dependencies & contractually specified interfaces**: As components are a solution for interoperability, there needs to be a mechanism to connect components with each other and to modify them dynamically. That is an important distinction: Components are not simply bricks that can be composed into a larger system (integration according to the four technique framework [BR89]). Components also provide a mechanism of specialization, which is their interface. As such, components enable so-called blackbox reuse [SGM02; RR03]: Reuse is possible without considering implementation details, as they are encapsulated in the component and hidden behind the interface. This minimizes the dependency of components on each other: In the software life cycle of patches and updates, a component can then be changed internally without breaking other components that rely on it. This is distinct from whitebox reuse

[SGM02; RR03], where reuse occurs with a full understanding and dependence on the inner working of the reused software (source-level reuse in our terminology). Blackbox reuse is commonly considered the more realistic approach for third-party reuse [RR03; Hen+07]. As for dependencies, a component may rely on other components to be deployed in the same environment. Also, there are aspects that the environment itself is expected to provide (for example, a file system interface) for the component to function. Here, the component pattern also overlaps with the package pattern in that dependencies are explicit.

- **Deployed independently**: This part of the definition implies that components are actors of their own and can function independently when deployed. As such, components actually are a pattern of distributed computing with components fitting the actor model [HBS73], since they can make local decisions, deploy sub components, and interface with other components through the context system.

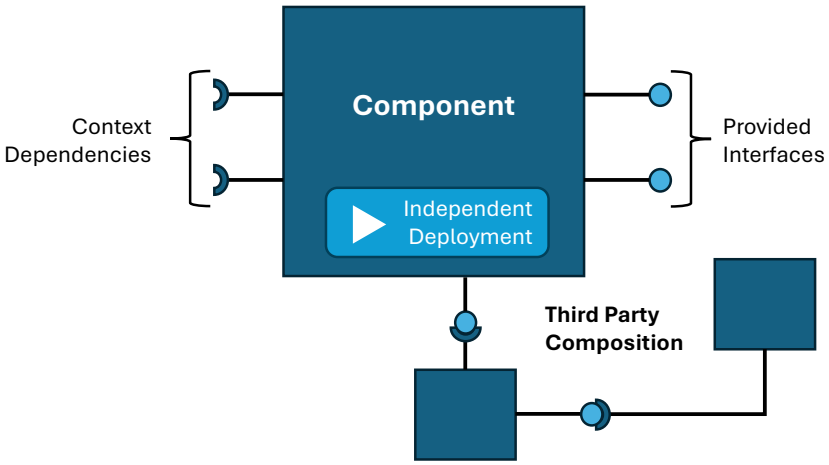


Figure 3.5: Structure of the component pattern according to our definition with context dependencies, provided interfaces, independent deployment, and third-party composition [SGM02]

Related Concepts to Components Below the level of components, there are more granular units of composition. For example, we would consider objects in object-oriented systems as another unit of composition, which satisfies some, but not all of our definition. One way to define objects is to say they are a unit of instantiation with a unique identity, which may have a state that can be externally observable, and that encapsulates its state and behavior [SGM02]. Objects are a unit of composition (objects can reference and instantiate other objects), which has contractually specified interfaces (properties and methods), and explicit context dependencies (the programming language, e.g. Java). Classes are instantiated into objects (or "specialized" in the four technique framework [BR89]), or may be part of a class hierarchy through the mechanism of inheritance (another ladder of abstraction). Critically, they are not deployed independently, though – a set of objects can express a software system, but it is a source-level reuse technique. Objects themselves can only be deployed as part of the expressed software system, not independently. Of course, components and objects do no exclude each other, either: A component may be realized using classes

and objects in an object-oriented language [Szy99]. As such, objects operate at a lower level of abstraction than components. Another related concept are modules, which are a first step towards components: Through imports and exports, there is a mechanism of composition and a specified interface. However, a module is only a set of related entities – in a language such as JS, a module could export constants, functions, and classes. For example, a math module may export π as a constant, an implementation of the sine function, and a class for fractions. This loose affiliation of entities is closer to the package pattern, as there is no mechanism of independent deployment – rather, modules are passive and meant to support source-level reuse. Some authors consider objects and modules as the two lower tiers of component systems beneath "standard components", envisioning a whole hierarchy [Aßm03], but this requires a wider definition than we allowed previously and is outside our scope.

Notable Component Systems Component systems have a long and rich history both in research with component based software engineering, and in practice with several thriving ecosystems of components. In the real world, component systems appear under many different names such as plugins, extensions, objects, applications, and so on (see Table 3.3 for examples). We will discuss the most recent evolution of components on the web, web components, in depth, but will first take a broad sweep.

In many ways, *applications on modern operating systems* were the first successful component systems [SGM02]. In the 1980s and early 1990s, the ancestors of the desktop operating systems still popular today were conceived, starting with the IBM personal computer in 1981, continuing with Mac OS System 1 in 1984, then Windows 1.0 in 1985, and finally Linux with the added X Window System in 1992 [ORe21]. The new hardware created a demand for consumer applications that would make use of the resources the system provided, for example interfacing with the file system [ORe21]. In this sense, users would compose applications within the environment of their operating system, and applications could depend on external resources or expose interfaces [SGM02]. Mobile applications on iOS (2007) and Android (2008) similarly have successful app/component systems – mobile computing provides access to specific hardware to apps such as location and camera access. Each operating system started offering their own marketplace over time (`Microsoft Store` [Mic12], `App Store` [App08], `Play Store` [Goo08b], etc.), which allows developers to distribute apps for third-party reuse. Compared to package systems, apps are distributed as bundled artifacts without dependency resolution, however.

The 1990s saw the rise of several *enterprise component systems* and is intertwined with the boom of object-oriented programming. A standardized, language-independent approach was taken by the large consortium of the Object Management Group (OMG). Founded in 1989, efforts of the OMG focused on standardizing object-oriented systems, especially distributed ones. The OMG specified the Common Object Request Broker Architecture (CORBA) first in 1991 with CORBA 1.1 [OMG91], with the current version being CORBA 3.4 from 2021 [OMG21]. CORBA revolves around the idea of an object request broker (ORB), which can be understood as a middleware that allows objects of one software to communicate with other objects, independent of software and hardware differences. As such, it is a solution for remote

procedure calls (RPCs) [SGM02]. As part of the CORBA specification set, the CORBA Component Model (CCM) [OMG06] describes a very complex structure for components (compare Figure 3.6). In the CCM, there are multiple types of input ports (receptacles, event sinks), output ports (provided interfaces, event sources), configuration/state (attributes), and also the special equivalent interfaces, which maps different internal facets [OMG06]. Component instances are then managed by so-called component homes [OMG06]. Suffice it to say that this model is a maximalist approach to component design, operating at a high level of abstraction with a way to express all imaginable types of interoperation. This enterprise-level solution is far too complex for our use case of EdTech developers creating components, and the overhead in usability would be too great – simply put: FAIRness can likely be achieved with a less complex design. It is worth mentioning that there are similar enterprise component systems in Microsoft's ecosystem (COM, OLE, ActiveX, COM+, .NET Framework) and in the Java ecosystem (JavaBeans, EJB) [SGM02]. They are too limited in scope for our purposes (being intended for Windows or the JVM respectively), and suffer a similar design complexity. However, it is informative that highly successful component systems in the enterprise world share several aspects [SGM02]: First, there is a rich ecosystem providing many foundational functions so that the component implementation becomes easier and component artifacts are fairly lightweight. Second, composability is not guaranteed, but that is sufficient – not all components work with each other as expected, but they can largely coexist. Third, there is a thriving third-party reuse economy with independent component providers and marketplaces. Fourth, components are on a level of abstraction that is meaningful to the client doing the composition, often by providing a direct visual representation (e.g. in Visual Basic).

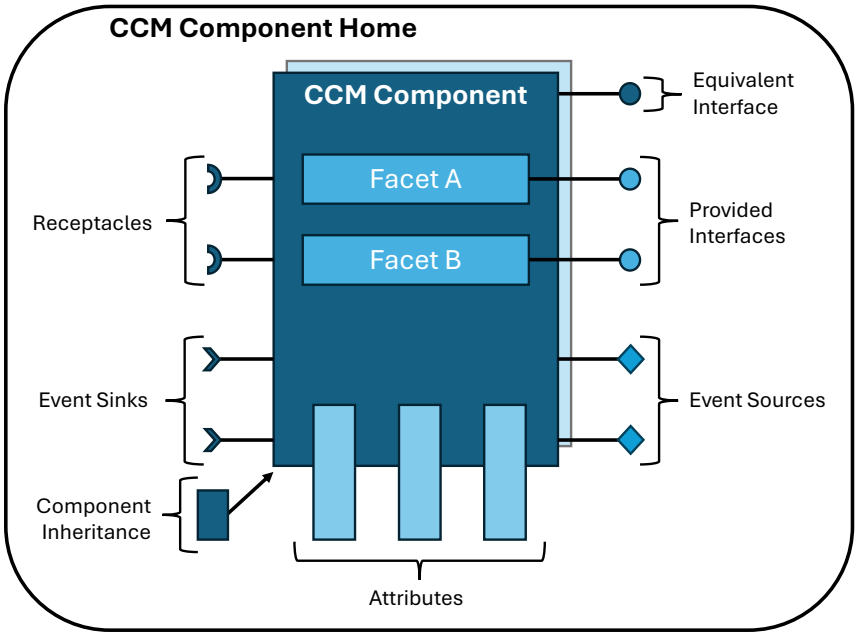


Figure 3.6: Structure of a CCM component in the CORBA ecosystem, with facets, types of ports, inheritance, and management in component homes shown [OMG06]

There are several miscellaneous component systems, the diversity of which shows the adaptability of the component idea to different contexts. Container runtimes such as `Docker` [Doc13] or `Podman` [Red19] offer an abstraction for particularly heavy, networked components. Technologies such as `Docker Compose`, `Docker Swarm` or `Kubernetes` use the container abstraction to provide a component system for web services. For example, a to-do list web application may be separated into three containers: A container running a database such as `Postgres` [Pos96], another running a simple `API` with a web framework such as `Flask` [Ron10], and finally a container running a reverse proxy such as `nginx` [Sys04] to regulate access and deliver a web application as the user interface. Game engines such as the popular `UnrealEngine` [EPI95] and `Unity Engine` [Uni05] offer their own systems and marketplaces for *game* components. In browsers, there is also a large economy of *browser extensions*, where the dominance of `Google Chrome` [Goo08a] established its extension format as the de-facto standard. Well-known types of extensions include ad blockers (e.g. `uBlock Origin` [Hil08]) and user tools such as for grammar checking (e.g. `Grammarly` [SLL09]).

System	Ecosystem	Year
Desktop Application	Windows/Mac/Linux	1984
COM/OLE/ActiveX/COM+/.NET	Windows	1990
CORBA/CCM	CORBA-compatible systems	1991
Java/JavaBeans/EJB	Java	1995
Game Object	UnrealEngine/Unity	1995
Browser Extension	Web Browsers	1997
Mobile App	iOS/Android	2007
Docker Container	Docker/Podman	2013
Frontend Component	React/Vue/Angular	2013
Web Component	Web Documents	2016

Table 3.3: Overview of common types of component systems within different ecosystems, sorted by year

Frontend Frameworks & Web Components A fairly recent, but very powerful addition to the web platform are the web component standards – they are part of an ongoing discussion about technology in EdTech, for instance [Kau+17]. As the name suggests, they are a realization of the component pattern for the web. Taking a step back: The base technologies of HTML, CSS, and JS are used to create web documents, and HTTP is used to transmit them to clients with a browser. The browser renders a runtime representation of that document, also providing access to that runtime representation via the DOM. JS scripts can use the DOM to manipulate the page at runtime, forming the basis for interactivity on the web – the user interacts, and the page is changed.

With this ad-hoc manipulation, web developers could build abstractions in the form of libraries other developers include in their pages, for example for component-based apps. As elaborated, components had been a popular approach for user interfaces for a long time, and as complexity grew, web developers desired to build their own, custom components for their apps. While there are precursors, the first

library to popularize components on the web was Meta's `React` [Met13] in 2013. With HTML, there is a fixed set of built-in elements, and traditionally, developers could only realize their own elements as instances of the existing ones. For example, a custom dialog may be represented as a `div` with a specific class, e.g. `<div class="my-dialog">...</div>`. `React` [Met13] provides a language within JS to allow specifying component templates in a declarative way (JSX). It also provides a data model where `React` keeps a so-called "virtual DOM" as separate data structure from the real DOM and efficiently and automatically updates the real DOM when changes are made (reactivity, the namesake of the framework). The latest StackOverflow Developer Survey [Sta25], which contains "49,000+ responses from 177 countries across 62 questions focused on 314 different technologies" provides a good impression about the popularity of `React`, which experienced a meteoric rise and is still strong today: 46.9% of professional developers surveyed had extensively worked with `React` during 2025. The other major UI component libraries, Google's `Angular` [Goo16] from 2016 and Evan You's `Vue` [You14] from 2014 follow far behind at 19.8% and 18.4% respectively. `Angular` had existed as the framework `AngularJS` before (starting in 2010), but rebranded to `Angular` and adopted the component pattern in 2016. `Angular` is a fairly heavy library which offers a large set of development patterns (controllers, directives, services, etc.) to solve common problems in web application development. `Vue` is a more lightweight approach explicitly developed as a reduced, more developer-friendly version of `Angular`. While the new abstraction of components was a large leap for the web developer experience, there were some drawbacks in practice: The component approach quickly led to the "div soup" problem, where a document's HTML was populated with meaningless div elements, because developers build highly nested structures of components with meaning in the code (e.g. dialogs containing buttons, containing icons), but no discernible semantics in the HTML (`div`, `div`, `div`...). This posed issues, especially to accessibility, since tools such as screen readers could not extract meaning from the div soup the same way that users without limitations could from the rendered document. With the arrival of LLMs, machine-readability gains importance outside of accessibility: A semantically structured web app is likely easier to interact with and process than a "div soup". Also, with libraries such as React, optimizing the performance could prove challenging since JS had to perform frequent and heavy computation to synchronize the virtual and real DOM.

It is interesting to note that by 2016, all major frontend frameworks (except perhaps the classic `jQuery` [Ope06], which however does not quite fit the category) had adopted `React`'s component pattern. The same companies developing these frameworks were by now in joint control of the web platform through the WHATWG standards body (and in Google's case, through the dominant `Chrome` web browser), which led to a push to standardize the component pattern. This resulted in the web component standards (compare Figure 3.7), which take the component pattern and integrate it with the core technologies of HTML, CSS, JS, and the DOM, which may help alleviate the limitations mentioned. In a nutshell, the three standards, now incorporated

in the HTML Living Standard [WHA25], define that web components are custom elements – JS classes extending DOM elements with custom behavior (1), whose structure can be declared with HTML Templates including nesting of child elements (2), and which can optionally hide internal structure (the "div soup") inside the shadow DOM (3). Web components can be structured and composed with HTML, as they can be used in the same way normal HTML elements can including attributes and children. Functionality can be implemented with JS in an object-oriented way that fits into the DOM, where custom elements are classes extending `HTMLElement` that can specify attributes, properties, methods, event listeners & emitters, and lifecycle callbacks. Additionally, they can provide a static interface on the constructor with its own properties and methods. As such, they can also be accessed during runtime as DOM elements. DOM interactions work as expected, with custom elements also hooking into features such as the event system (a custom progress bar element could fire a `completed` event, for example). Finally, they can be styled normally using CSS, including the possibility to expose parts of the shadow DOM for styling [WHA25].

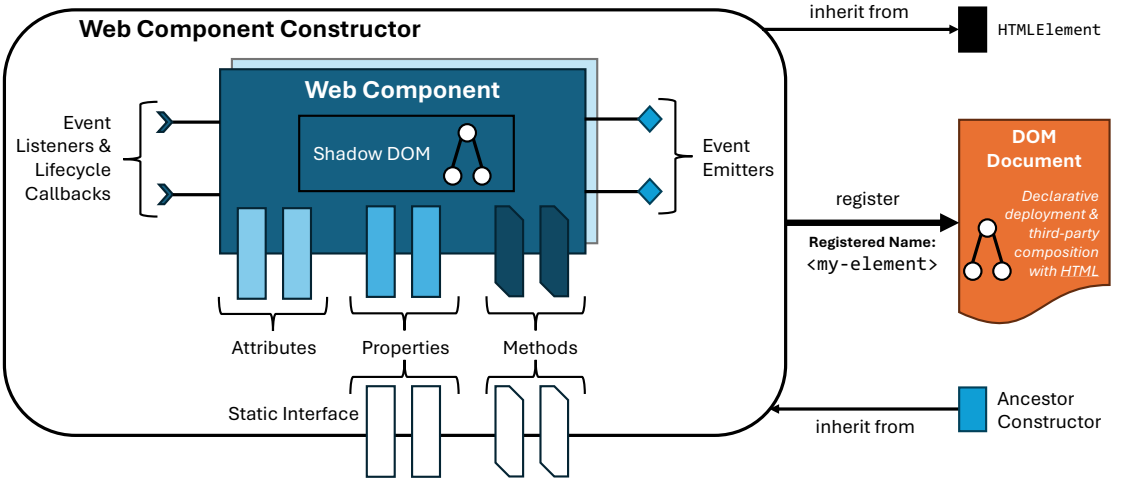


Figure 3.7: Structure of web components on the web platform with the types of ports supported, the static interface, inheritance chain, and the deployment mechanism shown [WHA25]

Web components represent a consensus and are limited in several ways compared to the frontend components of `React`, `Angular`, and `Vue`. Notably, while web components standardize the component pattern, there is no standard for reactivity – managing the application's state and synchronizing it to the DOM is left to frameworks and developers. This limitation is most likely intentional and fits the modular "building block" nature of web standards, where reactivity would have to be decoupled from custom elements – web standards must support current and even future use cases as broadly as possible. In practice, there are also limitations in certain technical details: Custom elements cannot be redefined, but must have unique names, leading to hard-to-resolve name collisions (e.g. importing two UI libraries which each export an `icon-button` component would lead to the second import crashing with an error). Also, the shadow DOM is commonly used when implementing complex web components, but compared to React and other frameworks, there is no declarative way of

specifying the shadow DOM's HTML template. However, these can be seen more as oversights in the specification and are bound to be resolved by future versions of the standard.

3.2 Authoring Paradigms for Interactive Resources

In this section, we will look at what approaches to the second part of our research (RQ2, see Section 1.4) could look like – in short: How can interactive multimedia resources be authored? First, we will look at how learning resources are used and reused in the domain of education, since this gives us a framework to understand what it means to author learning resources. This is important, as any authoring paradigm or concrete tooling would need to cover the requirements set by this workflow. Next, give an overview of different types of learning resources to differentiate what we mean by interactive resources. After that, we will detail some examples of interactive resources to see what needs to be authored. Finally, we evaluate different paradigms of authoring interactive content and highlight a few promising options which will form the basis of our own solution.

3.2.1 Reuse in Education

In education, use and reuse of resources is inherently multi-layered. Barring autodidactic settings where a learner is their own teacher, formal education occurs with an *educator* and one or more (usually many more) *learners*. As outlined in the introduction (Section 1.2), since educators usually do not create learning resources from scratch, but compose them out of existing learning and non-learning sources, there is also a layer of original *authors*.

Models of Reuse There are many models of framing the use and reuse of learning resources, most originating in open educational resource (OER) research. A maximalist approach is offered by van Assche & Vuorikari [AV06], which suggest no less than 11 steps for the lifecycle of a learning resource: Creation, description, approval, publication, discovery, evaluation, resolution, obtainment, reuse, integration, and use (additional phases which end the cycle are retraction and local deletion). While their model offers a complete picture, it includes steps such as approval and evaluation, which are of limited importance for our technically oriented research question. Building on earlier ideas [Gur08], a more reduced model is provided by the widely adapted OER life cycle [CP12], which models the reuse process from the perspective of teachers: Search, evaluate, adapt, use, and share. It puts composition and remix front and center, and the first steps are to search and evaluate existing resources, then adapting them. Another popular model is Wiley's five rights (5R) [WH18]: Retain, reuse, revise, remix, and redistribute. While this confirms the centrality of composing (remixing) and changing (revising) learning resources in practice, it is more of a legal lens and concerned with how to enable reuse normatively, not how to accomplish it technically. Overall, all models fail to make the layer of original authors explicit, who may create content that is not intended for education, but that is still used as such. For example,

famous historical pictures such as from the moon landing are commonly used when discussing the topic in history lessons. Also, the layer of learners is not given much consideration in most theories, although there is likely a flourishing world of use and reuse with photocopies, notes, and digital instances, as well. Any technical solution would need to consider these layers, as well.

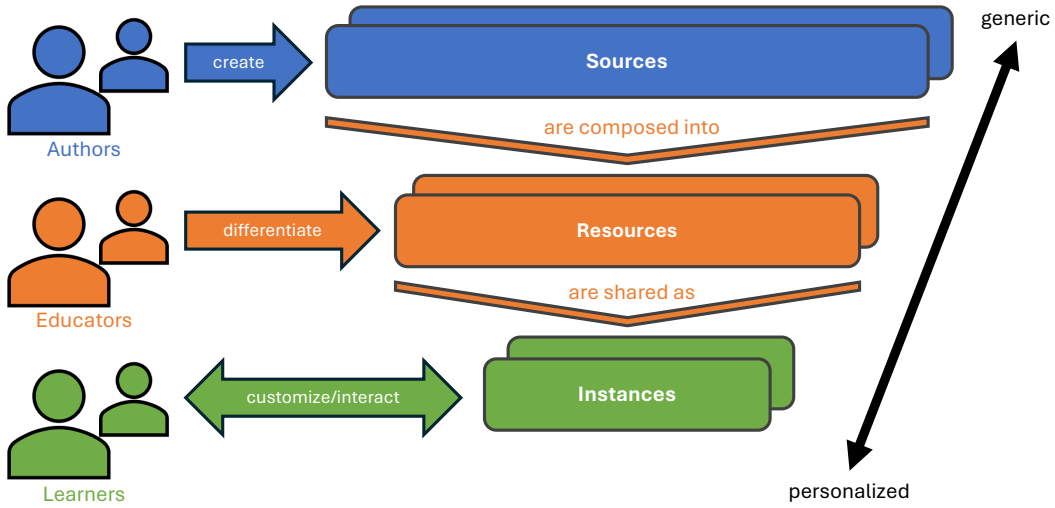


Figure 3.8: The remix workflow, based on the OER lifecycle [CP12] and the idea of personalized learning [SS20]

Remix Workflow To analyze existing types of learning resources and frame our own work, we can consolidate the previously outlined ideas into one working model (compare Figure 3.8) that considers the three groups of actors involved in learning resource use and reuse: Authors, educators, and learners. Our model focuses on three layers, each corresponding to a group: *Sources* are created by authors, possibly for non-education purposes. These are composed into *resources* by educators, who may differentiate them further with their learners in mind. Finally, resources are shared with learners who interact with them. Borrowing from software, we can call the documents the learners interact with *instances* – as there are usually multiple learners, most workflows, be they digital or analogue, involve a copying/instantiation step. As an example, we can again consider the history teacher from our introduction (Section 1.2) creating a lesson on medieval France: They use websites and even the knowledge of colleagues as sources (and could also use textbooks) to compose their lesson, which may be formalized as resources (worksheet, digital slides, etc.). Finally, the lesson is held, with each learner receiving their own instances (e.g. worksheet copies) of the educator's resources. Overall, our framing is compatible with the OER lifecycle and can be seen as more of an elaboration of that existing work: Searching and evaluating occurs in the step of composing sources into resources, while the step of adapting is split into composing the resource and differentiating it. Using the resource means sharing it with learners, and sharing with other educators is implicit in the model, since resources can become sources in other contexts.

Practices of Reuse In the remix workflow, the distinction between source and resource is a matter of perspective: In a different context, a resource may become a source, as well. One teacher's worksheet may become the basis of another worksheet, for example. This idea of educators exchanging resources is assumed to be a common practice, but largely hidden to researchers – *dark reuse* [Wil09]. A recent study offers an empirical confirmation of the practice and can be seen as a validation of the OER workflow and by extension, the remix workflow [Bea18].

We conclude that reuse in education can be summarized as a multi-step process of remixing which we consider universal across different media, be they analog or digital. This model forms the basis of any authoring paradigm.

Remix Workflow: The remix of learning resources involves original authors, educators, and learners. Authors create sources, which are composed into resources by educators and further differentiated. They are then shared as instances with learners, who interact with them, creating a learning experience.

3.2.2 The Structure of Learning Resources

Clearly, there are different types of learning resources: A textbook is different from a worksheet, which is different from a slide presentation. An exercise is different from an informational text. A paper photocopy is different from a website. We already provided a few fundamental distinctions (see Section 2.1) such as the targeted educational levels and fields, and the role of a resource in a wider context, e.g. an exercise.

A helpful framing, which we can borrow from textbook research [Val02], is to view learning resources as having their own language [Par+19]: They are made to support an intended learning outcome for a specific audience (pragmatics and semantics), and are composed (syntax) out of a lexicon of smaller units (morphology). For example, to teach addition and subtraction to first graders (pragmatics/semantics), an educator may compose a worksheet with two distinct parts (syntax) – an informational image explaining the concept and exercises for the pupils to complete (morphology).

Pragmatics of Learning Resources At the highest level, there are several taxonomies that can be used to describe who and what a learning resource is intended for. One common taxonomy applied is the one established by Bloom [Blo+64] and revised by Anderson and Krathwohl [Kra02]. It is a framework for learning objectives which arranges goals into a pyramid spanning from a base of cognitively simple tasks (remember) across increasingly complex tasks (understand, apply, analyze, evaluate) to the tasks considered most advanced (create) at the top. It is used in the design of lessons and learning activities, and has long found its way into educational practice through curricula and teacher qualification. While popular, this taxonomy is rather orthogonal to our research question: Interactivity is a property of the form or mediality of the learning resource, not of the objective. Simply put, interactivity can aid

of the goals mentioned, from flashcard programs (remember) to digital art tools (create). Similarly, other taxonomies and classifications can illuminate different aspects of the intended audience, and can serve as metadata: UNESCO offers exhaustive taxonomies for education levels (International Standard Classification of Education, ISCED) [UNE12] and fields of education (ISCED-F) [UNE14]. Since they are regularly used for worldwide statistics on education, they are an excellent way to describe subjects and education levels across the many education systems worldwide.

Syntax of Learning Resources Getting more granular, taxonomies about the structure of a learning resource are rare, though many distinctions on this level would likely be considered common sense. For instance, giving an example is a universal tool in education and communication in general – a common syntactical structure for learning resources would be to pair a definition with an example. The niche field of research into textbooks offers a syntactical taxonomy for the structure of science textbooks worldwide based on empirical data from the TIMMS survey [Val02]: It distinguishes the block types of narrative, related narrative, unrelated instructional narrative, related graphic, unrelated graphic, exercise set, unrelated exercise set, activity, and worked example. Furthermore, it observes three approaches to content sequencing: A single dominant content theme, a sequence of themes, or no real theme (fragmentation). The taxonomy is not intended to be exhaustive, but gives a useful look into what a syntax of learning resources could look like. Another, more general-purpose model comes from the area of learning objects: Churchill distinguishes six types of learning resources which he identifies based on literature in education, EdTech, and media theory [Chu07]: Presentation, practice, simulation, concept, information, and context representation (see Section 2.1). There are other predecessor taxonomies to Churchill's such as Wiley's taxonomy [Wil00] of fundamental, combined-closed, combined-open, generative-presentation and generative-instructional resources – in stemming from learning object research however, the taxonomy includes the tools that create learning resources (tools are considered learning objects, as well), which is out of our scope.

Morphology of Learning Resources At this most granular level, models focus on the properties of the parts or building blocks of learning resources. One taxonomy on the morphological level is the four-level model by Honegger, Hielscher, and Hartmann [Leh18]: Learning resources can be primarily (level 1) and offered in printed form, static with digital supplements (level 2), fully digitally made and used (level 3), or digitally made, used, and interconnected (level 4). This suggests an iterative relationship between the different types: Each level has more capabilities than the previous one. In web technologies, the standard of media types [MMK25] offers a limited morphology of digital content: Text, image, audio, video, model, font, and application (and miscellaneous types such as multipart and message) – that standard is however technical in nature and of limited use for theories in education. The areas of EdTech we established previously offer another implicit taxonomy based on the

historical development, where technological progress enabling new and more complex morphologies: From *informal resources* (lessons) across *static resources* (textbooks, worksheets), hypermodal *resources* (web articles, digital slides), interactive *resources* (code notebooks, interactive articles) to prospective connected *resources*.

As in natural language, each layer (pragmatics/semantics, syntax, and morphology) is related, but different. For instance, the morphology of a learning resource is quite distinct from its syntactical role: An image can be used as an example (e.g. a photo of a cow as an example of a mammal), or as part of an exercise (e.g. the same cow photo could be used in a memory game). Going up a layer, that exercise could be useful for any number of learning objectives. This layered relationship allows us to better formulate what an authoring paradigm should do: It should allow educators to express any semantics/pragmatics they desire, by providing a broad set of elements (morphemes) to compose in different ways (syntax).

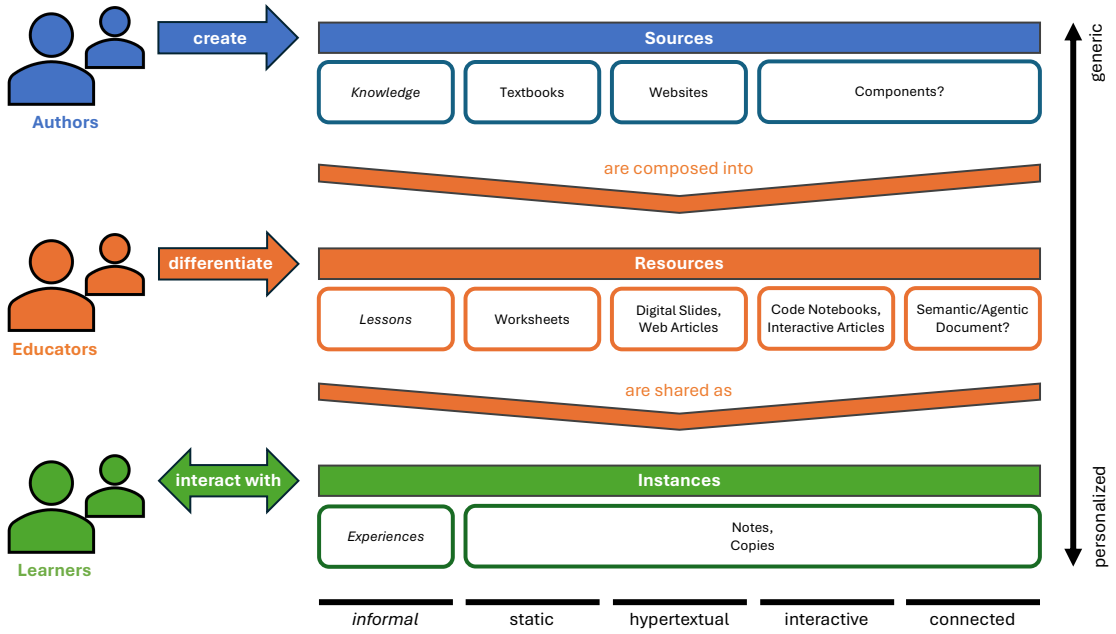


Figure 3.9: The multi-level remix workflow with examples, adding the dimension of different morphologies of learning resources in increasing complexity (x-axis).

We can combine all these insights with the remix workflow above into in a single model, which is similar to the four-level model introduced before (compare Figure 3.9). This model allows us to describe complex remix practices and puts interactive resources into context. In the model, there are sources which come in different morphological forms (informal, static, hypermodal, interactive, and connected), which are fully or partially composed into resources (syntax) by educators to achieve learning objectives (pragmatics). Resources themselves can again have corresponding morphologies. Finally, resources are shared as instances with learners (again with different morphologies). Note that the model does not imply that creating learning resources is only derivative: Each of these three layers involves original work (creating sources, differentiating resources, and interacting with instances).

Overall, the model introduced is a summary of the existing research on reuse in education. It allows us to clearly articulate that editing has three distinct levels, which an authoring paradigm needs to support.

> **Authoring Paradigm for Interactive Resources**: A paradigm must support the remix workflow. This includes modifying elements (morphology), rearranging static, hypermodal, and interactive elements into a composite resource (syntax), and annotating the resource with metadata (pragmatics).

3.2.3 Interaction Styles

Authoring paradigms and the tools that implement them may use different interaction styles to accomplish the task of authoring. As we survey different interaction styles, we better review related authoring tools (see Section 3.3) and conceptualize our own authoring paradigm. An interaction style defines how a user interacts with a system. Classic UX textbooks identify four different types [Soe25]:

Command Languages Command languages are the earliest interaction style, in use even in the early mainframe-client era of computing: Expert users type in prompts with parameters to instruct the system. This is usually done in a command line interface (CLI) or terminal, where previous commands and outputs are shown in sequence, with a command prompt at the bottom. In operating systems, a command language usually interoperates with the file system in that commands are executed at a certain file path. This interaction style is flexible and powerful for expert users, as it blurs the line to programming with user-defined scripts and macros. It also functions easily across network boundaries, as only plain text needs to be transmitted. These advantages make command languages still useful today for programmers, system administrators, and similar experts. However, the learnability and retention of commands is considered very poor, leading to high error rates. Also, the expressive power useful to experts also means it is harder to provide assistance to novice users, as the space where problems can occur is much larger. For these reasons, command languages have largely been abandoned outside expert circles.

Form Fill-in & Menu Selection This pair of interaction styles is a simplification of commands (menu selection) and parameters (form fill-in) respectively. With menu selection, available commands are labeled, grouped and shown to the user (for example in a menu), who can select them and observe a change in the system. Commonly, this is coupled with keyboard shortcuts to execute commands quickly. Form fill-in means presenting a user with a set of predefined form fields to fill out. Usually, digital forms consist of a set of input fields with fixed labels. Input fields can be different depending on which type of data should be entered: For example, if an American user needs to enter their address, a dropdown could be shown for the 50 US states. There is ample research on forms and menus, for example how to optimally design and arrange them [Shn+17]. From a UX perspective, it is advantageous that the user only needs

to recognize the field or command in question, rather than remembering its syntax. The menu or form also guides the user through the experience, as labels provide discoverable information. Too many menus or forms however can lead to information overload, and the interaction style is considered fairly slow for experienced users. The screen space taken up is also significant, which can be a problem with smaller displays. Form fill-in and menu selection are still used in modern tools, although they are usually paired with other interaction styles, such as direct manipulation.

Direct Manipulation This interaction style is the most complex out of the ones presented. Direct manipulation means an interface in which objects of interest have visual representations, and those representations can be rapidly and incrementally changed through affordances that are shown as part of the representation [Shn83]. Direct manipulation can be summarized with the famous acronym WYSIWYG: What you see is what you get – users can always see the current state of the system through the visual representations. A well-known example is drag n' drop in file explorers, where clicking selects a file, and dragging can be be used to move the file to another folder: The user can incrementally move the file closer to its destination, or easily cancel the operation, as well. Direct manipulation is usually paired with an undo/redo functionality so users can rapidly manipulate objects and simply reverse their actions if necessary. Most modern operating systems and software applications are primarily direct manipulation-based, with form fill-in and menu selection for specific parts. Direct manipulation is considered easy to learn, as feedback is instant, making errors more avoidable and also reversible. It also encourages exploration and recognition of patterns. However, it is usually harder to implement for programmers, can take up a lot of screen space, and not every concept lends itself well to visual representations.

Other Interaction Styles Modern research also accounts for new technologies such as VR/AR and chat/voice interfaces. For example, Rogers et al. suggest a more abstract taxonomy of interaction styles [RSP23]: Instructing, conversing, manipulating, exploring, and responding. Especially the conversing interaction style is worth accounting for in 2025, as the development large language models (LLMs) makes conversational user interfaces (CUIs) far more viable than previously. Users are no longer forced to learn command languages and may now employ natural language to interact with a system. For now, LLMs are commonly integrated as agents within direct manipulation systems (e.g. `Microsoft Copilot` or `Apple Intelligence`).

3.2.4 Authoring Paradigms & Tools

There is a large landscape of different authoring tools that are used to create learning resources. Most of the tools we discuss here are general-purpose and not limited to education – just as with design patterns for interactive elements, there is a large overlap with other fields. We can observe that the tools follow different paradigms to solve morphological, syntactical, and pragmatic editing. Each paradigm may be more or less suited to a specific layer, and we may apply our categories of expressiveness and usability to analyze them. What all tools presented here have in common is that

they do not present complete ecosystems that connect a design pattern for interactive elements with an authoring paradigm – we will discuss complete solutions in the next section (Section 3.3).

Related Concepts to Authoring Paradigms There are some notable technologies we consider distinct and not applicable to the search for an authoring paradigm. First, there are special-purpose tools and markup languages such as MathML and SVG for mathematics and vector graphics, respectively. There are corresponding tools such as `Inkscape` for vector graphics. While those languages and tools allow authors to express specific morphologies, they provide no solution for expressing more complex syntax structures that mix different morphologies. Second, there are semantic document approaches [Nor89] such as the text encoding initiative XML format (TEI-XML) – it is mostly suitable to goals such as encoding existing texts in the humanities, as it offers no control over the presentation of text, limiting both expressiveness and usability. In general, we will look at data models rather than serialization formats, about which there is also ample research [Jol89], as the runtime data model is more closely tied to the authoring process we are interested in.

Morphology	**Paradigm**	**Modern Tools**	**Pioneer**	**Year**
Static	Markup	LaTeX — *via text editors, Overleaf*	TeX	1978
	Rich Text	Microsoft Office, LibreOffice	WordStar	1979
Hypertextual	Markup	LiaScript, Markdown, BBCode, Mavo, TangleDown, Visdown, AML — *via text editors*	HTML	1989
	Rich Text	TinyMCE, CkEditor	WebMagic	1995
Interactive	Markup	Jupyter, Quarto, Observable	Jupyter	2011
	Rich Text (limited)	Formative, Fobizz, Articulate	Adobe Captivate	2002

Table 3.4: Authoring paradigms and tools for different morphologies of content, sorted by year of pioneering tool

This first survey of authoring tools will allow us to map out which paradigms are available and could be combined into a new authoring paradigm for interactive learning resources. All authoring tools that will be discussed are listed by morphology and paradigm in Table 3.4. We will discount historical, non-software approaches of typography [WP25] and focus on authoring paradigms still in use in modern tools. It is worth a short mention that *programming* would be strictly the first digital approach to creating documents, established with the first machines in the von Neumann architecture in the late 1940s and early 1950s [Von93]. Simply put, a Turing complete programming language could make a computer do anything computable [Tur37], including producing documents of any morphology (static, hypermodal, interactive, and

even connected) – however, we are more interested in approaches specific to authoring documents.

Authoring Static Content The first specialized approaches for authoring content digitally were *markup-based*. Markup means including additional, machine-readable information interspersed with text content ("marking up the text"). Three types of markup can be distinguished: Presentational markup is concerned with the display of content. Procedural markup consists of instructions to process the text, such as loops to produce text iteratively. Finally, descriptive markup is used to label structural parts of the document to achieve a separation of content and presentation. Static content includes formatted text with rich content such as images and mathematical formulas. Looking back at interaction styles, markup-based approaches are similar to command languages: The author needs to remember a markup syntax to produce a marked up document, which serves as an input for a processing system that produces the final content. First predecessors for markup-based authoring of content included IBM's GML in 1971 and GNU's groff in 1973 [Fur89]. The pioneering work popularizing the whole approach came into 1978: After frustrations with the hardware-based Monotype system when typesetting his own book, computer scientist Donald E. Knuth developed a software system for typesetting called TeX (tau-epsilon-chi in Greek, abbreviating *technē* meaning art or craft) [Knu78; Knu84a]. TeX and its later, today mostly synonymous extension of LaTeX by Leslie Lamport [Lam94] separates content and presentation: The LaTeX language uses descriptive markup such as chapter, section, title, and so on to express documents, which are then laid out by LaTeX macros. Layout can be separately adjusted by extending the LaTeX system with different macros, as LaTeX is technically only a set of commands on top of TeX to make common document typesetting tasks easier [Lam94].

Authoring Hypermedia Content Hypermedia (static content enhanced with hyperlinks and multimedia) started out with a similar markup approach, as the base language of the web, HTML, is also a markup language with mostly descriptive markup. Over time, as HTML tends to be fairly verbose and needed to be fairly general-purpose to serve the whole web, many special-purpose web markup languages with different design tradeoffs were established. One of the earliest examples is Bulletin Board Code (BBCode), which offers a slightly simplified syntax for HTML, especially for complex elements such as lists and tables. A similar custom language is Wikitext, the markup language used by MediaWiki/Wikipedia, which is more terse than BBCode. The most popular example is Markdown, which provides perhaps the most terse syntax for the more commonly used elements in HTML (headings, paragraphs, bold/italic, etc.), with an escape hatch of writing HTML directly if the subset offered by Markdown does not suffice.

Authoring Interactive Content For interactive content, a similar markup-like approach became popular in the Web 2.0 era of the 2010s, where IPython (later named `Jupyter` [Jup14]) pioneered the approach of mixing static and hypermedia content with executable code providing interactivity. `Jupyter`'s offering is mostly focused on

data science and research in general, although the technology has gained some prominence in computing education, as well. Based on earlier ideas and prototypes revolving around the literate programming concept by Donald E. Knuth, `Jupyter` offers a single-file JSON document that can viewed and edited in a Jupyter environment. The document is a flat sequence of blocks, where each block is either a Markdown-based rich text block, or an executable code block (there are also so-called "raw NBConvert" blocks for technical reasons). The `Jupyter` environment is modular in nature, where the kernel executing the code block can swapped out – by default, there is a Python kernel, but through this swapping mechanism, different programming environments can be supported. Recently, there has also been work to formalize markup languages in a document calculus, which is a reaction to the plethora of similar markup languages available nowadays [CK24].

Out of the paradigms mentioned (programming, markup, and rich text editing), we will only evaluate rich text in more detail, since programming and markup are of limited use for our research. Using general-purpose programming languages for documents places the author squarely in the Turing tar-pit (see Section 1.3): Everything is possible but nothing of interest is easy, meaning the authoring paradigm is as expressive as it can be, but not usable at all to the average educator. Markup languages are an improvement and are successful among technically-oriented authors (programmers, researchers) for static, hypermodal, and even interactive content, but either limit expressiveness severely (e.g. `LiaScript` [Die19]), or have deficiencies in usability as they opt for programming as an escape hatch for interactivity (e.g. `Jupyter`).

We previously established that authoring paradigm for interactive content needs to support changes on the morphological, syntactical, and pragmatic level.

3.2.5 Rich Text Editing Paradigm

A modern paradigm which shines at manipulating the syntactical level is rich text:

> **Rich Text Editing Paradigm**: Direct manipulation of structured documents by writing, selecting, formatting, copying & pasting, allowing for interspersed blocks of rich content such as images.

The operative parts of our definition are structured documents and direct manipulation, which we will elaborate on in the next paragraphs.

Structured & Active Documents At the root of rich text editing is the idea that most documents are not simply a sequence of characters, but a tree of objects, which are in turn made up of simpler objects [AFQ89]. There were originally other document models for structured content [Fur89] such as a linear model in Bravo (sequence of paragraphs with associated styles), the inline model in OPS (sequence of paragraphs and inline objects), or the environmental model in Scribe (sequence of paragraphs that can be grouped in nestable environments). A full tree-like model was perhaps first realized with Tioga at Xerox PARC (tree of paragraphs with associated styles),

while the model commonly used by modern tools is Furuta's "strict tree — not strict tree (tnt)" (ordered tree of different type objects) [Fur89]. The different objects in the tree, apart from paragraphs, could be mathematical equations, tables, graphs, charts, drawings, and so on [Fur89]. The web uses this exact data structure for web documents with HTML and the DOM. The same goes for the standardized XML-based document format shared by word processors, Office Open XML, which is again an ordered tree of objects. We can already see that this data structure for documents is flexible enough to not only represent static (word processing) and hypermodal (web) documents, but also interactive content – since the ordered tree can contain any kind of object, why not interactive nodes? This idea was originally put forward with the term active/dynamic documents [Spi88; TB90], thought of as an extension of structured documents [QV94]. Researchers imagined simple node transformation activities, such as an automatically generated table of contents, but also features such as fetching data from a database for an online telephone directory [TB90]. Applications of active documents in research (reproducing results automatically, a niche now filled by computational notebooks) and education (interactive textbooks) were suggested [Cla90]. An overall definition that links active documents with components is given by Aßmann [Aßm05]: "An active document is a component-based document with a set of derived components that is computed from a set of base components. To this end, it contains or is tightly associated with software." We can see that components are a type of object in the document that is instantiated ("computed from a set of base components"), which matches the ideas of software reuse we discussed in the last section (Section 3.1).

Direct Manipulation of Structured Documents Rich text editing systems are usually direct manipulation systems coupled with form fill-in and menu selection for certain interactions. Rich text editing mostly operates on the level of syntax (rearranging elements inside the ordered tree): Through the notion of a selection with an anchor and a head (the selection's start and end point), objects in the tree can quickly be grouped and actions performed on that group. With selection, copy & paste alone, syntactical changes can be performed at any granularity. It is also able to express text formatting, for example through wrapping a part of text in an *italic* node, although this can lead to issues such as when handling overlapping formats (e.g. "hello **wo*r*ld***" with two overlapping text formatting options: Which is the correct tree to represent this?). For some actions, a form fill-in is more suitable: A classic example is marking a part of text as a hyperlink, where the author needs to specify both the part of text to anchor the hyperlink on, and the hyperlink itself – most systems simply offer up a dialog and form field for entering the link. For entering metadata such as an author and a license, form fill-ins are almost always used, since the metadata that could be added is fairly well-defined and limited, so the form cannot grow too complex. Most authoring tools also opt to offer custom direct manipulation approaches for specific types of object in the structured document tree (e.g. affordances to scale, translate, rotate, and crop images).

Discussion of Rich Text The rich text editing paradigm as presented here is strong both for syntactical changes (rearranging the document tree) and pragmatic changes (modifying metadata), which is perhaps why it is used in the most popular authoring tools to date. However, as we move from hypermodal to interactive morphologies, the authoring tool can no longer make many assumptions about the content: While elements such as tables or images are well-defined and an authoring tool can build specialized direct manipulation interfaces to handle them or fall back to form fill-in and menu selection approaches, no such general interface is possible for interactive content.

3.3 Explorable Authoring Systems

In the previous sections, we presented the issues of finding a design pattern for interactive elements (Section 3.1) and of finding an authoring paradigm (Section 3.2) as separate. In practice, they would need to be integrated in a single system, as there needs to be an interface between the authoring tool (implementing the authoring paradigm) and the interactive element (implementing the design pattern). This kind of system we call an explorable authoring system, as it is a complete solution for the remix of interactive, hypermodal and static content into a structured document.

3.3.1 Integrating Interactive Elements and Authoring Tools

In an explorable authoring system, the authoring tool and the interactive elements (components inside packages) need to interface in multiple ways along the lines of FAIRness: (see Table 3.1):

- **Package management (find, reuse, access)**: Authors need to be able to search packages based on their metadata, as well as install, update, and uninstall them.
- **Editing interface (interoperate)**: Authors should be able to edit components using an interface.

For the systems presented in the following sections, we will discuss both the package management and editing interface, which in turn can influence our own implementation later.

3.3.2 SCORM Learning Objects

The earliest full explorable authoring system is centered around the idea of learning objects and the *Sharable Content Object Reference Model* (SCORM) standards (see Figure 3.10 for a SCORM tool). Learning objects are an attempt to transfer the object-oriented paradigm to education. How learning objects could be defined in theory was contentious at the time: According to Wiley [Wil08], an often-referenced definition is given by the IEEE: "Learning Objects are defined here as any entity, digital or non-digital, which can be used, re-used or referenced during technology supported learning. Examples of Learning Objects include: Multimedia content, instructional content,

learning objectives, instructional software and software tools, and persons, organizations or events referenced during technology supported learning." This definition is so broad as to be practically useless (as exemplified by the sarcastic remark "My left big toe is a learning object" [Wil08]). Wiley's own definition "any digital resource that can be reused to support learning" is more focused, but can still include many different entities. Due to this muddled state of learning object theory, we will limit ourselves to the kind of learning objects practically implemented – for the theoretical framing, we stick to the more well-defined ideas of packages and components introduced in the last section (Section 3.2). The Sharable Content Object Reference Model (SCORM) standards facilitate the interoperation of learning management systems (LMS) and interactive elements. Originating from the US military-industrial complex, SCORM offers a fairly rigid, sequential model of learning. The earliest SCORM standard receiving wider adoption was SCORM 1.2 [Dod01] in 2001. This was later extended in SCORM 2004, of which the fourth and final edition was released in 2009 [Jes09].

Implementation in SCORM SCORM's unit of reuse is called the shareable content object (SCO). In theory, any granularity of content is supported, from a small page to a whole course. Multiple SCOs can be bundled in a single package. SCORM packages fit the design pattern we introduced before, as they aggregate content and associate it with metadata, such that it can be accessed and reused by LMSs, or found through learning object repositories. One part of the specification, called the Content Aggregation Model, defines the permissible metadata and packaging: A SCORM package should be a ZIP archive with an XML file `imsmanifest.xml` at the root, containing the package's metadata. The other part of the specification fits the component pattern: SCORM's runtime environment describes the behavior of content running in an LMS and defines an interface a compliant LMS needs to satisfy so that the running content can call it via a JS adapter. The API consists of 8 functions and offers logic to initialize and finish the content (there is the notion that content is sequential and played like a video), getting/setting a value for the state of the model, and handling errors. In practice, SCORM content would usually be run in a separate frame inside the LMS: The LMS would simply open the content, causing scripts to be executed which communicate with the LMS via the runtime API (e.g. calling `LMSInitialize()` when started to indicate loading is complete, and `LMSFinish()` when done). SCORM 2004 adds a more powerful features for LMS-and-content interoperability, such as a specific data model for learning objectives and for assessment. Also, there is a very complex sequencing specification which allows SCORM creators to describe in what sequence SCOs are supposed to appear and what navigational controls are afforded to the learner – essentially baking hypertext-based adaptive learning into the standard. Overall, the described architecture means that SCORM is only a container for arbitrary web content, and as such, development of components can be done with any tool that produces web-deployable content (HTML/CSS/JS).

Authoring in SCORM In SCORM, there is no distinction between authored resources and programmed component – everything is content. As such, authoring only means using an authoring tool to produce web-based content, which is exported as a

SCORM content package. In the public sector and in corporate training settings, dedicated authoring tools became popular. There are too many such authoring tools to enumerate, but some examples include `Adobe Captivate` [Ado02], `Elucidat` [Lea13], `Articulate Rise 360` [Art16], and `Gomo Learning` [Lea10]. In each tool, authoring may take its own shape, but most solutions revolve around the rich text editing paradigm, often creating slide presentations with interactive content.

Discussion of SCORM In the learning objects literature, there are many criticisms leveled at the concept: For instance, Learning objects are ill-defined, and the SCORM standards are of limited pedagogical use due to their narrow design [Fri04]. Both points stem from the design of SCORM, a black-box approach in terms of content that is a strength and a weakness at the same time: On one hand, SCORM is very flexible in that developers may use any technology, and authors may use any tools to produce content. Both a twenty part lecture series and a single interactive worksheet could be packaged with SCORM. In essence, SCORM is simply content packaging with not much of a component model. However, the lack of a proper component model is the decisive limitation that makes remix very difficult on the technical side. As Wiley put it [Wil08]: "Learning objects can be aggregated, but not adapted". This is not a limitation that SCORM or future learning object standards could overcome, since the black box is fundamental: The standards represent an ill-defined concept. In turn, a black box with no discernible interface cannot be remixed, since only composition is not enough. Put shortly, learning objects are so reusable as to be useless – learning objects fall into the reusability paradox with no possible escape through remixing.

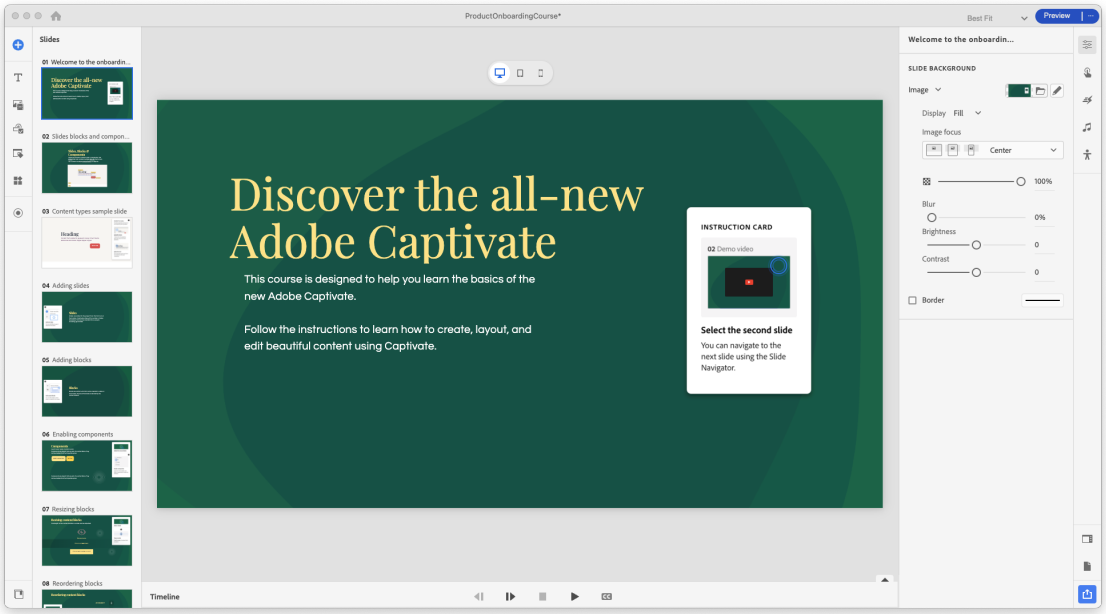


Figure 3.10: Interface of Adobe Captivate, a modern tool revolving around learning objects

3.3.3 H5P

H5P [H5P13] is likely the most popular solution for interactive content among educators. Started in 2013, H5P (originally an abbreviation for HTML 5 Packages) is a project with several parts [Mag21]: An editor allowing authors to sequence and nest interactive content types, integrations for common content and LMSs (e.g. WordPress [LM03] and Moodle [Dou02]), and a file format to express both content and content types. H5P is open core, where the components described are provided under open source licenses, but there is a commercial SaaS offering (H5P.com) including advanced content types, analytics, and hosting services. H5P is especially popular in tertiary education, as is shown by the many studies using H5P in a variety of subjects [UP22; RFA24; SSR21; MAS23; Wic+21; SC20; JC24].

Implementation in H5P H5P realizes a plugin-based architecture: H5P content is structured document tree of content blocks, each of a certain type. Content types are defined in H5P libraries created by developers. A library is a directory with several files (another implementation of the package pattern): A H5P-specific `library.json` contains metadata, versioning information, and entry points for executing the library. Another file called `semantics.json` specifies the input data format, out of which a configuration form will be generated. The functionality is contained in JS files, e.g. `mytype.js` (referenced in `library.json`), which are scripts that define content types as properties of the global H5P object. H5P provides a fixed `jQuery` [Ope06] version as the recommended way of implementing content types. Finally, styling can be added with CSS files, e.g. `mytype.css` (again referenced in `library.json`). Libraries can also depend on other libraries, including on libraries of helper functions. The directory can be shared as a `.h5p` file, which is a ZIP archive.

Authoring in H5P H5P content is usually authored in a CMS environment such as Moodle, where an editor is provided within the H5P plugin. An alternative is the standalone tool Lumi [SR20], which is a desktop application allowing a file-based workflow producing `.h5p` or `.html` files. Content types can be found and accessed through a search interface, installed and added to the document. To edit blocks of these types, the H5P editor reads each `semantics.json` and generates an input form for it. Once all required form fields are filled out (form fill-in), the content can be previewed or saved. For content types which can contain other content types (e.g. "Course Presentation" for interactive slides), nested forms are generated. Authors can also attach metadata such as licensing to each block.

Discussion of H5P The explorable authoring system provided by H5P has some significant strengths: Finding and managing content types (package management) is made very simple, and the platform offers a well-curated set of content types with rich metadata for authors to choose from. The editing interface using purely form fill-in also has a rather low barrier of entry. Still, the usability of H5P is arguably poor (an assumption we will confirm in our requirements study, see Section 5.2), both for

developers and for authors. Developers need to use a custom package format and syntax to create content types (components), which largely parallels established formats (`library.json` being very similar to `package.json`, for example). This reinvention of the wheel forces developers to learn a custom workflow just for H5P. Authors are confronted with many forms to fill out – this likely works well for small documents, but breaks down with more complex and nested document, a typical limitation of a form fill-in interaction style. Also, basic usability features such as undo/redo and a side-by-side, automatically updating preview are not available in H5P. Overall, we can look at H5P as an important predecessor which already solved some aspects, such as package management.

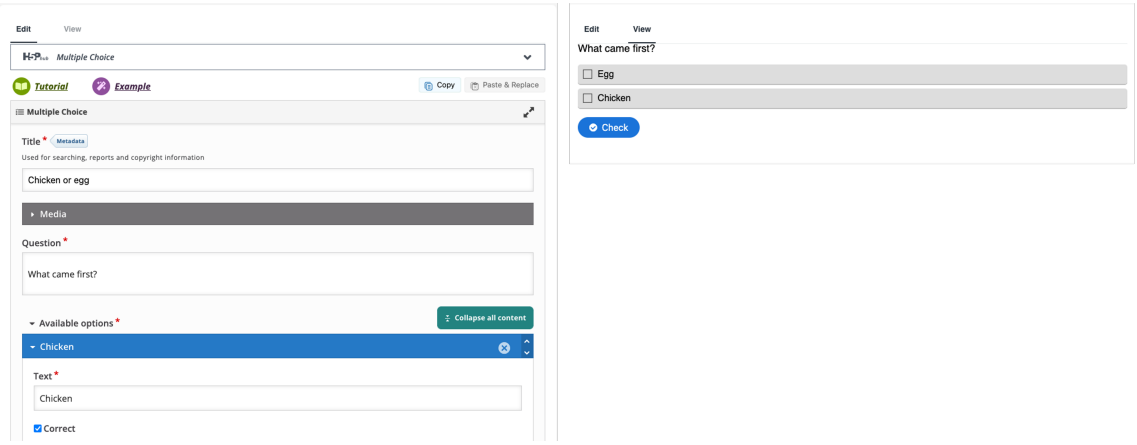


Figure 3.11: Interface of Lumi, a standalone H5P authoring tool, with the input form (left) and the final output preview (right). Note that both were combined for the figure as there is no side-by-side or live preview.

3.3.4 MDX

A recent and minimal solution for an explorable authoring system is MDX [Ver17] (see Figure 3.12). Started in 2019, MDX is a markup-based approach mixing two technologies (Markdown and JSX) in a language for interactive content that compiles to HTML.

Implementation in MDX In MDX, components are compiled and bundled into a document. How this compilation is done is determined by the context chosen by the user. For example, there are official contexts for React and Preact, and also Vue. If an MDX document is compiled in the React context, all imported components must be React components. With the MDX syntax, they can be arranged in any syntactical form inside the document, and passed props can be used to configure the component. For developers, this means that components can be created with different web technologies, although this needs to match the author's choice of context.

Authoring in MDX The authoring process in MDX is similar to Markdown: Authors can use a text editor to write structured documents in the MDX markup language with the aid of syntax highlighting and a preview. MDX documents are standalone text files with a preamble in JS (i.e. to import components) and a main part in Markdown syntax. A compile step is necessary to produce a HTML document to display in browsers, although there are integrations available in CMSs such as MDX.

Discussion of MDX MDX is elegant in that it is not coupled to a specific component technology – components can be implemented in React [Wal13], but also in Vue [You14], for example. As JSX forms an interface between component and structured document, MDX documents remain declarative. This decoupling is desirable, especially in the fast-changing world of web development. Still, having to choose a single context means that authors cannot mix components of different provenance (e.g. a React and a Vue component) in a single document. Also, as established, the markup paradigm does not offer the same degree of usability to non-technical users as the rich text editing and paradigm. Finally, there is no solution for package management: Authors would need to peruse public NPM packages intended for developers in search for useful components, then install them with a package manager and bundle them into the MDX file – in short, a programmer's job. These limitations mean MDX is more suited for developers creating mostly hypermodal web content with some interactivity sprinkled in, and less so for educators.

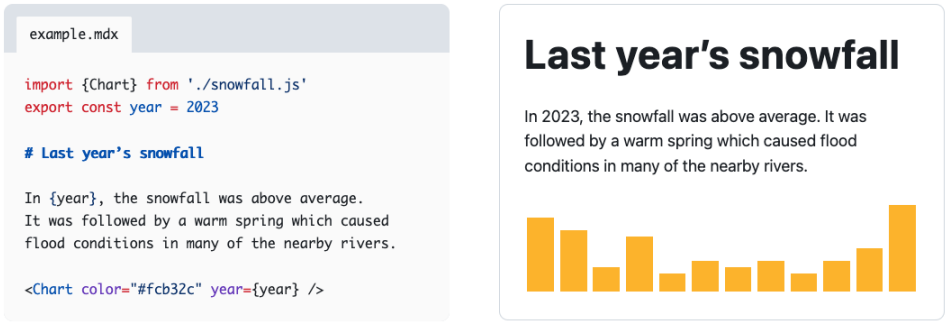


Figure 3.12: MDX example from the MDX website, with markup language on the left and output on the right

3.3.5 Idyll

The most recent entry in the explorable authoring system space is Idyll. Started in 2018 with the Idyll markup language [CH18], the system was completed with the release of the associated authoring tool Idyll Studio in 2021 [Con+21]. Unlike MDX and H5P, it is primarily an academic project. It provides the markup language (Idyll) which compiles to HTML, as well as the mentioned authoring tool Idyll Studio to edit documents. Idyll is intended as a general-purpose solution for interactive content, mainly addressed at journalists publishing interactive articles, but also at educators.

Implementation in Idyll `Idyll` offers a plugin system through components, which encapsulate interactive content that can be used alongside the hypermodal content expressible through `Idyll` markup. Components must be implemented in `React`. The state of the component in the document is determined by its `React` props, which can be passed through `Idyll`'s markup syntax (e.g. `[Counter value:1]` would pass the number value `1` to the `Counter` component instance). Also, components may have children, which enables more complex, nested structures such as different layouts. Through `React`, logic components such as conditionals or loops are also possible to implement. [CH18]

Authoring in Idyll `Idyll` supports two authoring paradigms. Either, authors can use the Idyll markup language directly [CH18], or they use `Idyll Studio`, which provides an interface combining rich text editing and markup to author documents [Con+21]. The markup syntax is similar to Markdown (and `MDX`), with some unique features for components (using a HTML-like syntax) and data (reactive data expressions and data sets). Built-in reactivity (if a value changes, dependent expressions, properties, and so on are also updated, similar to spreadsheets) is the main differentiator between `Idyll` and MDX, otherwise, the differences are only in syntax. The UI-based alternative to writing `Idyll` markup directly, `Idyll Studio` is, as far as we can tell, the only authoring tool combining rich text editing and markup for interactive content. `Idyll Studio` is a hybrid between both authoring paradigms, as the document is displayed in its final rendering (no preview, WYSIWYG), but editing blocks still occurs by changing the `Idyll` markup. When a text block with some content is clicked, the editor shows the corresponding `Idyll` markup in place as a text area to be edited. There are some direct manipulation style affordances for components, such as a palette to drag and drop components from. Components can also be nested through `Idyll Studio`, either by dragging and dropping or through editing the markup. Components are mostly limited to the built-in ones available: There is a mechanism to include custom components from external JS files (likely for developers creating components), but no way to find or access published components. Components themselves can be manipulated by form fill-in, as a property editing form based on the component is generated.

Discussion of Idyll We can observe that `Idyll` takes a step towards applying the rich text editing paradigm to interactive content. Many of the ideas it presents are very strong, such as the tie-in to the existing web development ecosystem via `React` with only a minimal required interface, which allows developers to use preexisting knowledge and cooperate within the web development community. The editing experience is in many ways a step up from previous solutions such as `H5P`, as the rich text editing interface reduces cognitive load, and components can be configured in-context instead of filling out a large set of forms in a separate step. However, if we aim for a system to author interactive content, which supports the remix workflow, it is still slightly incomplete: There is a concept of components, but not of packages, meaning that in terms of FAIRness, interoperability is solved, but not findability, accessibility, and reusability. Aside from that, tightly coupling the system to `React` is

an imperfect solution, as there are many other popular approaches for components on the web developers may want to use, and a more generalized solution utilizing web components seems achievable technically. For authors, the WYSIWYG interface is fairly usable, although a full rich text editing approach that removes the requirement to learn `Idyll`'s syntax may be more successful. This is arguable, since there are fairly popular technologies such as LaTeX and Markdown which work with purely a markup approach – still, their popularity outside of programming and academia is far eclipsed by rich text editing software such as the Office suite.

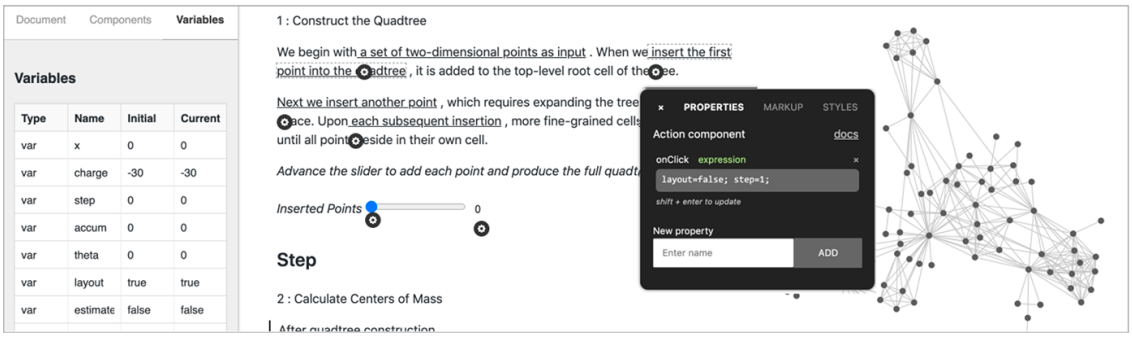


Figure 3.13: Interface of Idyll Studio, reproduced from [Con+21]

3.4 Summary: Hypotheses

In this chapter, we presented work both broadly and closely related to each of our research questions.

For our first research question regarding a design pattern for interactive elements (Section 3.1): We established the general idea of abstraction, arriving at the concept of design patterns, which gives us a language to describe possible answers to our research question. Discussing software reuse in general, we arrived at the FAIR4RS principles (Findable, accessible, interoperable, and reusable research software) as a theoretical framework to understand the reusability of interactive elements: FAIR interactive elements are remixable. To achieve FAIRness, we presented two design patterns, offering a theoretical framing, related concepts, and a history of implementations for each. The package pattern solves findability, accessibility, and reusability by bundling associated resources and attaching metadata. The component pattern solves interoperability through decoupling and well-defined interfaces.

For our second research question regarding an authoring paradigm for interactive content (Section 3.2): Analogous to the previous section (Section 3.1), we first discussed reuse in education and theoretical models thereof, integrating prior work into the remix workflow, which describes how educators compose resources out of existing sources by original authors, and share them as instances with learners. We then looked at the process of authoring in detail, arriving at a multi-level model where content of different types (static, hypermodal, interactive, and connected) may be changed on different levels (morphology, syntax, and pragmatics). We conclude that to support authoring in the remix workflow, an authoring paradigm should allow changing content of any type on each level. We surveyed paradigms identifiable in

research and practice (programming, markup, rich text editing) and analyzed that building on rich text editing is especially promising for interactive content.

Integrating both parts (Section 3.3), we analyzed explorable authoring systems, which pair a design pattern for FAIR interactive elements with an authoring paradigm. We inferred that some additional interfacing between the interactive elements and the authoring tool is needed for any such system to function. We then presented in-depth looks at the most important explorable authoring systems, showing how developing components and authoring content work in each system and discussing it: SCORM learning objects, `H5P`, `MDX`, and `Idyll`. We observed that markup approaches such as `MDX` are expressive, but not usable to most educators. Form fill-in solutions such as `H5P` are limited in expressiveness and their usability deteriorates for bigger documents. We concluded that `IdyllStudio` is the closest to a full rich text editing solution for interactive content, but falls back on difficult-to-use markup for editing interactive elements.

The results of this chapter allow us to formulate working hypotheses as proposed answers for our research questions (see below). Regarding RQ1, our hypothesis suggests that combining components and packages would lead to FAIR interactive elements, which are then remixable. For RQ2, we propose to extend rich text editing with a new solution for morphological changes of interactive content to fully support the remix workflow.

RESEARCH SUMMARY

CT: In the context of a technical solution for web-based, interactive resources to support active learning, aimed at post-primary STEM learners:

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

- **RQ$_1$**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
  - ◇ **HY$_1$**: Interactive elements are remixable if they are findable, accessible, interoperable, and reusable (FAIR), which is possible by applying the component and package patterns and adding a domain-specific API.
- **RQ$_2$**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?
  - ◇ **HY$_2$**: Authoring should support the remix workflow for educators by extending rich text editing with a new paradigm to cover morphological changes to interactive content.

CT=Context, RQ=Research Question, ME=Method, HY=Hypothesis

Chapter 4 Methodology

In this chapter, we will discuss ways in which the research questions could be answered methodologically. We will show an overall research design, while specifics are shown in later chapters.

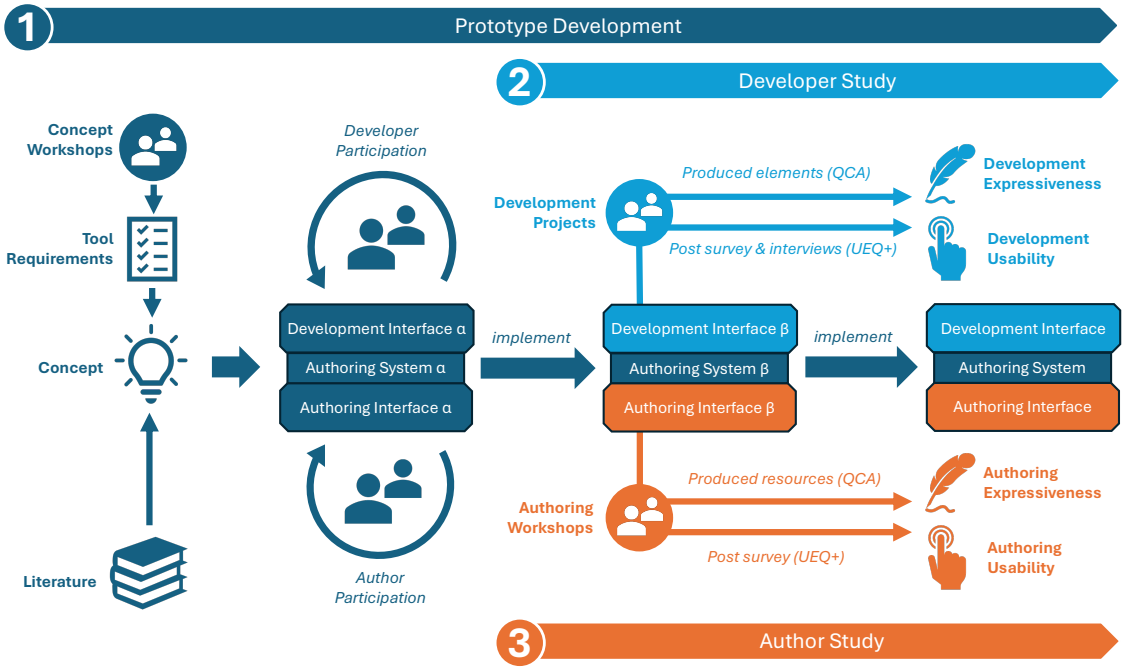


Figure 4.1: Research design of this project including prototype development and evaluations

There are two main objectives to our research (see Section 1.4), namely a design pattern for interactive elements and an authoring paradigm for interactive content. Both objectives are abstract concepts that must be operationalized to be evaluated empirically. We can answer our research questions with a research design, which we can choose from several well-established approaches.[1]

We seek to produce theories (a design pattern for RQ1 and an authoring paradigm for RQ2) which satisfy the properties of usability and expressiveness. There is a also a third property at the heart of this research: Remixability, that is, the degree to which

[1] In this chapter, we use methodological terms common in all empirical research (e.g. theory, evaluation, etc.). For a comprehensive overview with definitions, see the standard textbook of Cohen et al. on research methods in education [CMM17].

the structure of an interactive element (or interactive learning resource for RQ2) allows it to be reused. This framing already nudges us towards the empirical/positivist paradigm: We seek to test our theories by attempting to falsify our hypotheses. Note that we do not intend to prove the hypothesis as true – we try to falsify it and consider it valid if it holds true in the face of our attempts to prove it wrong (Popper's principle of falsifiability [Pop63]). Interpretive approaches which focus on understanding the individual [CMM17], or critical theory approaches [CMM17], which focus on critiques of power structures and ideology, are certainly valuable for other questions, but do not fit our investigation since we seek a solution to a technical and fairly universal problem. In our case, falsifying our hypotheses would mean showing that our design pattern (RQ1) or our authoring paradigm (RQ2) is not usable, not expressive, or not remixable.

Overall, several methodologies popular in educational research are worth considering [CMM17]. Many are clearly not applicable to our questions (e.g. historical/documentary research, netography, longitudinal studies). We argue that a meta analysis/review is also not sufficient based on the research gap shown in the last chapter (Chapter 3) – a comprehensive solution does not exist but needs to be developed and tested. Similarly, case studies belong more in the interpretive world and do not fit our positivist approach, as they would not do much to falsify our hypothesis – isolated examples are insufficient for falsification. Another approach is action research, in which actors in a system research their own actions to improve established practices – while involving educators and developers in the development process is valuable and the researchers of this project are personally familiar with both groups, we seek to craft and test a theory from the outside rather than improve practice from the inside.

Considering Design-based Research One way to pragmatically address the specific challenges of this project (novel and complex implementation, multiple target groups, many confounding factors) is an approach popular in EdTech: Design-based research (DBR). There are several variants of design-based research (also called design-research and development research) [AS12], and in practice, the method has evolved since the earliest suggestions by Collins [Col92] and Brown [Bro92]. We base our understanding on the contemporary literature, which identifies six main properties [AS12]: First, the research should be situated in a real educational context (similar to field studies). Second, it should focus on the design and testing of a significant intervention. Third, it usually employs mixed methods in data gathering and analysis. Fourth, it involves multiple iterations (similar to conventional design processes such as in product or software development). Fifth, it involves a collaborative partnership between researchers and practitioners. And sixth, the design principles should evolve together with the developed intervention. From these properties, we can understand that design-based research is focused on creating a practical impact by iteratively developing an intervention, and abstracting findings as design principles. As such, it is a cycle of inductive and deductive steps: Design principles are abstracted from evaluating an intervention, and a new concept is deduced and implemented based on these principles. To apply DBR to our project, we would need to

design a loop where a prototype is developed and adjusted, which has both a development interface addressing EdTech developers, and an authoring interface addressing educators, connecting both in a prototypical explorable authoring system. According to Collins (one of the original theorists), the approach is especially strong for addressing theoretical questions in practice and to form theories inductively [CJB04]. Other researchers appreciate precisely that the method generates measurable theses that can be investigated in follow-up research [AS12]. Here also lies the main tension and critique of the method: Is generating a testable thesis a research project in itself warranting its own method, or is it rather just a step of "true" experimental research? We do not intend to pass judgment on DBR in general in that vein – there is not one scientific method but many different ones across science (for example, the field of history primarily uses its own method of source criticism [Mül20], as there is rarely a way to conduct experiments on the past). Still, for our project, DBR is only appealing for the iterative design process, but cannot answer our research questions.

Considering Experimental Research Most viable in our judgment is an experimental approach which means constructing situations in which the design pattern and authoring paradigm, represented by the development and authoring interfaces of the explorable authoring system, can be tested in terms of expressiveness, usability, and remixability. An experimental design is a common approach across educational research. The gold standard of experimental design would be a randomized controlled trial (RCT) [CMM17], meaning the selection of participants is truly randomized and confounding factors are controlled for. We could for example design a prototype representative of our new authoring paradigm, recruit educators for a study, and randomly assign them to the different explorable authoring systems identified in the last chapter (A SCORM tool such as Articulate, MDX, H5P, and Idyll Studio), as well as our own prototype. However, this presents issues: First, the sample size needed so that the randomization procedure sufficiently controls for the numerous differences between educators (educational levels, fields, age/career phase, etc.) is far larger than practically attainable with our means – it is very unlikely that we would be able to recruit hundreds or even thousands of educators for what must be a multi-hour process which involves trying out the developed prototype. Second, this setup would only give us a single point of feedback for the developed prototype, which in our opinion is a near guarantee of failure due to the project's complexity. That way, we may get a methodologically perfect, but worthless result. We may run several RCTs in sequence in a longitudinal study instead, but this is even more unattainable for a small research project that involves substantial implementation work. The same issues would apply to the setup for finding a design pattern (RQ2) – developers of EdTech are similarly hard to recruit and diverse, with even more time they would have to spend to participate in an experiment. So while an RCT would represent the highest standard of validity we could achieve, we must scale back our approach on pragmatic terms. Instead, we seek to create a prototype, introduce it to groups of educators and developers as an intervention, and measure our variables in post tests. This has some methodological limitations in terms of control which we will discuss later, but we consider that approach sufficient for testing our hypotheses nonetheless.

We also want to distinguish between the early stage of creating a viable prototype, and the experimental evaluation of the final prototype – since we implement a novel, untested solution, it is important to be able to iterate quickly with some feedback. Furthermore, we anticipate the prototype implementation to be unusually complex, as we cannot adapt existing tools which run on different authoring paradigms and employ different design patterns for developers, but need to provide a faithful representation of a new paradigm. Involving feedback of developers and educators in this process is in itself is not methodologically necessary to test our hypotheses, but is a faster approach to arrive at a viable prototype for testing. As such, both phases involve our target groups of educators and developers, but realize that involvement in different ways.

Overall, we arrive at a two-stage design (see Figure 4.1) where the first stage uses participatory design methods to quickly iterate on a minimum viable prototype with a development and an authoring interface with input from developers and educators, while the second stage follows an experimental setup. The iterative, participatory development approach early on affords a lot of flexibility that is necessary to implement the novel design pattern and paradigm, while the more rigorous final evaluation is intended to prove that the research questions are addressed.

RESEARCH SUMMARY

CT: In the context of a technical solution for web-based, interactive resources to support active learning, aimed at post-primary STEM learners:

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

ME: Create a prototype with participatory design. Evaluate it in experiments with authoring workshops and development projects, for each proving remixability theoretically and measuring usability (UEQ+) & expressiveness (QCA).

- **RQ_1**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
  - ◇ **HY_1**: Interactive elements are remixable if they are findable, accessible, interoperable, and reusable (FAIR), which is possible by applying the component and package patterns and adding a domain-specific API.
- **RQ_2**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?
  - ◇ **HY_2**: Authoring should support the remix workflow for educators by extending rich text editing with a new paradigm to cover morphological changes to interactive content.

CT=Context, RQ=Research Question, ME=Method, PR=Prototype, HY=Hypothesis

Chapter 5 WebWriter

In this chapter, we will present our prototype for an explorable authoring system, `WebWriter` [Sal25b]. This gives us a technical overview of the system, so the following chapters can focus on the specifics for our research questions.

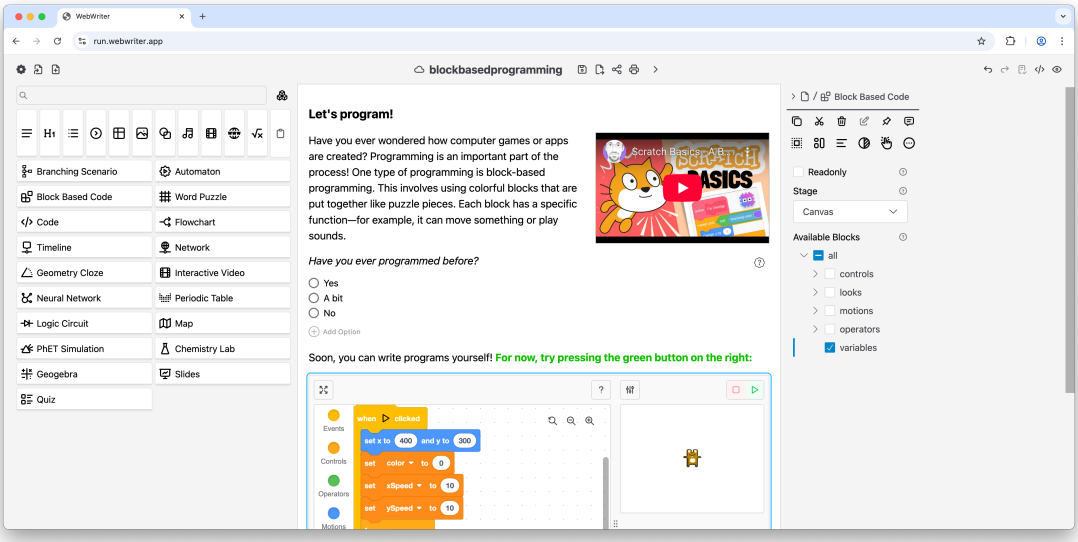


Figure 5.1: Interface of WebWriter, a web application available at `https://webwriter.app/`

The chapter is divided into four sections. First (Section 5.1), we discuss development methods and choose a process. Next (Section 5.2), the requirements study and requirement model is presented. Then (Section 5.3), we look at the finished concept of the system from the perspective of both target groups (educators and developers) and discuss design decisions. Finally (Section 5.4), we outline the finished implementation, also presenting several implementation challenges faced.

5.1 Development Process

As outlined, to test our hypotheses, we need a research prototype (see Chapter 4). Implementing it is a software development process, undertaken by the main researcher. Following a specific methodology helps systematize our work.

5.1.1 Development Methods

Software engineering research and practice offers many different development process models tied to different methodologies. One can broadly distinguish between sequential and iterative processes. Sequential processes (often referred to with the "waterfall" metaphor), have largely fallen out of favor as they are too inflexible for the dynamic and changing nature of modern software. Rather, iterative processes (usually called "agile") have become dominant [ASR02].

Within agile methods, there are many different processes which opt for different orderings, cycle lengths, and activities to account for the large range of team and project sizes involved. Some we will ignore since they are closer to philosophies as they are not well-defined and offer only a set of advice (e.g. agile modeling, pragmatic programming, and open source software). A few methods allow us to formalize the whole development process (see Table 5.1). Some are more appropriate for business contexts with very large teams (DSDM, FDD, and Crystal). Others such as extreme programming (XP) are too prescriptive about specifics not viable in our context (e.g. pair programming). Approaches such as Scrum are popular, but also ill-suited to solo developers. Out of the methods presented, adaptive software development (ASD) and the rational unified process (RUP) are the most viable, since they model the complete process in an agile fashion without prescribing team sizes.

Method	**Description**	**Year**
Dynamic Systems Development (DSD)	2 sequential (feasibility & business study) and 3 iterative phases (functional model, design/build, implementation)	1994
Scrum	Small teams, defined 30-day release cycles, sprints and standups	1995
Extreme Programming (XP)	Customer-driven, small teams, daily iterations, pair programming	1996
Feature-Driven Development (FDD)	Overall model, feature list and plan, then short iterations for each feature (hours to weeks)	1997
Rational Unified Process (RUP)	Four phases (inception, elaboration, construction, transition) with iterations of nine workflows	1999
Adaptive Software Development (ASD)	Incremental cycles (1-2 months) of 3 phases (speculate, collaborate, learn) with feedback (learning loop)	2000
Crystal	Family of models for different team sizes and project criticalities, medium-length increments (1-4 months) with iterations	2005

Table 5.1: Agile software development methods listed in [ASR02], sorted by year

We opted to base our work on the RUP model, since the RUP's phases (inception, elaboration, construction, and transition, explained below) give us a straightforward plan of action for the implementation, while the RUP's activities offer helpful terminology to think about and present our process (we will use this in the next chapter). During each phase, different activities occur as necessary, with the some activities (use case modeling, requirements) mostly in the first phase, and others (deployment) mostly in the last phase. In total, the RUP model defines a set of nine activities [Som11]:

1. **Business/use case modeling**: Modeling how a system may be used
2. **Requirements**: Actors and use cases are considered to model requirements
3. **Analysis and design**: The architecture and functionality is modeled and documented
4. **Implementation**: The design is realized with code
5. **Testing**: Fulfillment of requirements is verified
6. **Deployment**: The system is packaged and distributed to users
7. **Configuration & change management**: Changes to the system are tracked and managed
8. **Project management**: The overall development process is managed
9. **Environment**: Tooling for developers is distributed

Development Tools Sticking to modern software development best practices, the whole process can be tracked and modeled using `GitHub` [Pre+08a] as a development platform. GitHub offers several features which make the development process easier and also ensure FAIRness of the source code (e.g. version control, project management, and releases). As `GitHub` is the largest platform for software developers by far, the project's source code can be found and accessed easily. While `GitHub` is proprietary, Git itself is a distributed version control system, making it easy to mirror source code locally or to other remote locations. This means that we do not become captive to the platform, and migrating to another platform such as `GitLab` [ZS11] or `Gitea` [Xia16] in the future would be trivial.

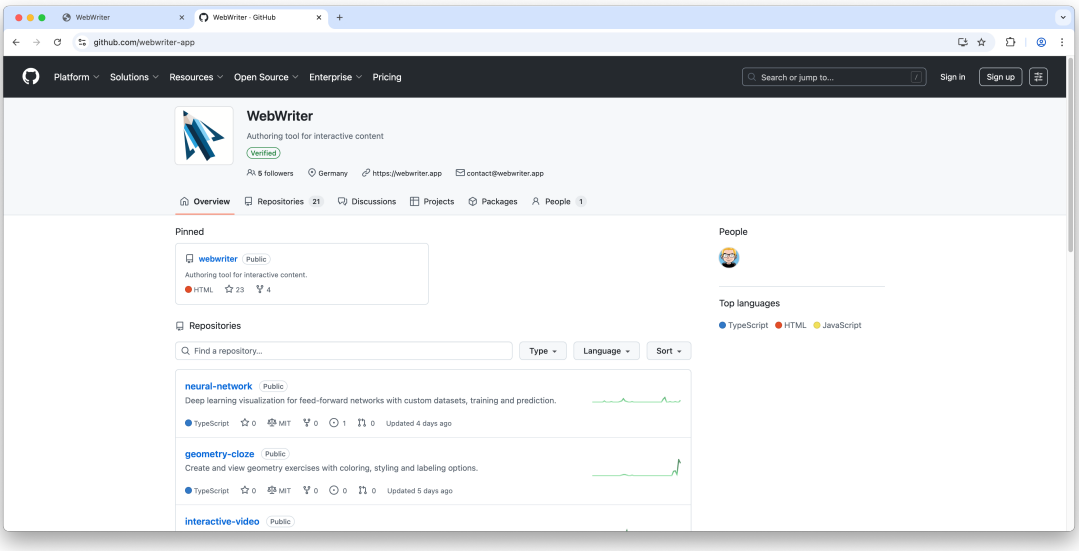


Figure 5.2: WebWriter's GitHub page with the main project and widget repositories

5.1.2 Development Phases

Development is undertaken by the researchers as they perform the above activities using the development tools with continuous input from both educators (authors) and developers. We distinguish three phases: Alpha, beta, and release.

Alpha Phase The goal of the alpha phase (covering the inception and elaboration phases in RUP terminology) is to formulate a concept for the explorable authoring system and develop a minimum viable prototype. First, this takes into account the related work, which is condensed into our two hypotheses. Second, the phase includes a concept workshop study with educators from which requirements are extracted. Before an implementation, a concept needs to be created. On one hand, this can be derived from the extensive related work we documented in the last chapter. But on the other hand, basing the system on related work alone fails to account for the unmet needs of educators. To supplement the related work, a requirements study is needed, which we will report on in the next chapter. Both the related work and extracted tool requirements come together in an initial prototype for the authoring interface, development interface, and the system connecting both parts. It is not the goal for that prototype to be feature complete or free of issues, but to implement all core functionality. During the rest of the alpha phase, the prototype is iterated on with participation from both target groups (developers and educators) to make sure that the implemented system fits the requirements and to identify missing features that become more apparent when working with an actual prototype. This phase is necessarily incremental, as both feedback from educators and developers, and also technical limitations may warrant changes in the implementation. This is based on the concept of participatory design [Spi05].

Beta Phase Once the system reaches feature completeness as envisioned by the concept, it enters the beta phase (the construction phase in RUP). In the beta phase, the system can be formally evaluated as both the authoring and the development interface can be considered complete implementations of the authoring paradigm and design pattern respectively. For the implementation, this also serves as a testing phase to eliminate smaller technical issues and identify features to extend the system with in the future.

Release Phase In the context of a research project, it should be assumed that the final software prototype will only receive limited support (or none at all) after the project ends. For this reason, the final phase (the transition phase in RUP) is concerned with the FAIRness of the prototype itself: Small issues brought up in the beta phase should be fixed and most importantly, the prototype should be published according to best software engineering and research practices. We plan to ensure the FAIRness of the research prototype by undertaking several measures according to FAIR4RS (Table 3.1): A versioned release of the prototype is made and attached to the repository (F1, F1.2), while the interactive elements which are components of the main prototype are separately versioned and released (F1.1). The available facilities of GitHub are used to provide searchable metadata (F2, F3, F4). The code will be released under an open source license (R1.1), provenance is documented through the Git history (R1.2), and the software will be implemented with the web standards (R1.3). Also, the platform provides easy access through web or Git protocols (A1, A2). Finally, we intend to use domain-relevant data formats (e.g. HTML) and make qualified references to other objects where needed (I1, I2).

5.2 Requirements

Section adapted from own work [SRS23a]

In the alpha phase, the first step towards realizing `WebWriter` was the specification of requirements, which are based on a requirements study with a small sample of educators. The insights from that study and our analysis of the related work (Chapter 3) yield a model called the explorable authoring requirements (EAR).

5.2.1 Requirements Study

Study Setup For the collection and analysis of teachers' requirements on explorables, their creation and usage, we conducted a workshop-based user study with 19 teachers (11 male, 8 female) at different proficiency levels (8 teacher training students, 2 teachers in training, 8 teachers of various experience levels, and 1 former teacher working at a university). The study was constructed as an introductory workshop on `H5P` [H5P13] (see Section 3.3) and `Lumi` [SR20], chosen as the closest approximation of system to author and remix explorables usable for all STEM educators. It was advertised through mailing lists, social media, flyers, and in different teacher networks. The workshop was structured through six phases (in 3,5 hours): (1) Exploration of prior knowledge and experience with tools for creating learning materials; (2) Introduction to explorables; (3) Introduction to `Lumi` and `H5P`; (4) Creation of

a specified explorable; (5) Creation of a personal explorable; and (6) a closing session reflecting on the experiences and lessons learned. While the workshop was moderated by one person, another person was there to observe and record the participants' contributions. In order not to prime the participants with the ideas and concepts behind explorables, H5P and Lumi, the initial phase of the workshop was used to get the participants talking and creative. As such, the participants were divided into four groups to discuss and collect answers to the following questions: Which digital tools have you been using to create learning materials? What do you consider important in a digital tool used for creating learning materials? What are the advantages of digital materials? What advantages do you see in non-digital materials (e.g., paper worksheets)? In the second phase, the concept of explorables was demonstrated by showing examples, which also served to motivate the participants for the next phases. For the third phase, participants were introduced to the tools used, first with a slideshow-based general overview, and then with a live demonstration by the moderator. Phases 4 and 5 constituted the main part of the workshop, where educators first created a simple explorable under guidance, and then created a second explorable more freely. During the phase, the moderator answered individual questions. At the end of the fifth phase, participants voluntarily presented their created explorables to the group. Furthermore, the participants' opinions and experiences were discussed at the end of each phase to collect more requirements as well as critical viewpoints and feedback on Lumi and H5P, as both may serve as opportunities to provide a more suitable alternative, resolving issues and disadvantages. To quantify the usability of Lumi, we used the System Usability Scale (SUS) [Bro96]. The SUS (see Table 5.2) is a 10 item questionnaire where subjects rate statements about a system's usability on a five point Likert scale (from "strongly disagree" to "strongly agree").

Item	**1**	**2**	**3**	**4**	**5**
I think that I would like to use this system frequently			x		
I found the system unnecessarily complex			x		
I thought the system was easy to use				x	
I think that I would need the support of a technical person to be able to use this system		x			
I found the various functions in this system were well integrated					
I thought there was too much inconsistency in this system					
I would imagine that most people would learn to use this system very quickly					
I found the system very cumbersome to use					
I felt very confident using the system					
I needed to learn a lot of things before I could get going with this system					

Table 5.2: The system usability scale (SUS) [Bro96] – note that 1 would also be labeled "strongly disagree" and 5 would be labeled "strongly agree".

Results Based on the different phases of the workshop, we collected the participants' opinions and feedback on Lumi and H5P as well as their viewpoint on explorables compared to analogue learning materials. As such, the first phase was used

to activate participants prior knowledge and let them share personal experiences and opinions. So far, teachers have been using a large variety of tools to create learning materials. Mentioned tools include text editing and presentation software as well as specialized tools like quizzes, audience response systems (e.g., Kahoot [Wan08]) or simulations (e.g., PhET [WAP08]). Also, school- and subject-specific applications were mentioned (e.g. GeoGebra [Hoh01]). When it comes to the question of what participants find important in a digital tool for creating learning materials, expressed properties include interactivity, ease-of-use, intuitive design, compatibility to different operating systems, export functionality, personalization features and secure, privacy-preserving data handling. Participants argued that ease-of-use and intuitive design are relevant for both for the teachers authoring and the students learning with the explorable. The participants expressed that the authoring process for digital learning materials needs to be intuitive and suitable to existing workflows. A tool adding additional complexity and not integrating well into the teachers' workflow is considered a hassle and may be not successful among the majority of teachers. In line with the advantages of explorables based on the literature, the participants argued that interactivity and adaptability are advantages of "going digital". Furthermore, motivational aspects and activation of learners were mentioned. For non-digital materials, the participants emphasized reliability and independence from the school's infrastructure, and the advantage of no required digital literacy skills. Some of the participants also argued that learning through handwritten notes is effective. The overall workshop concept was well accepted and engaging for the participants, observable through the lively small group discussions and mingling participants during the different phases. The participants also gave positive feedback on the format and workshop content in the closing session. Lessons learned included positive but also critical perspectives on Lumi, as it was a bit crowded and easy to get lost in. The UI was criticized as being lengthy and needed a lot of scrolling through forms for different elements. However, Lumi was also considered more suitable to author H5P content compared to direct H5P authoring in Moodle. Furthermore, the different H5P content types were received positively, as they are a more uniform alternative to a large set of different tools that teachers use so far. In a closing survey, participants assessed the usability of Lumi using the SUS (see above), resulting in a score of 70.26, to be interpreted as a "C" letter grade.

The system should...	**Specific features from the workshop**
. . . allow creating interactive multimedia usable for students	mix of content types, making interactive videos, embedding simulations
. . . be usable for teachers in remixing workflows	live preview, undo/redo, appealing visuals, unlimited composition of content, copy/paste external resources
. . . support personalized learning	specific content customization options, theming, conditional branching
...enable reuse & retention	use on any operating system, import/export, usage within LMS, usage without LMS

Table 5.3: Educator's requirements for explorable authoring systems (Specifics not fulfilled by Lumi underlined) (reproduced from [SRS23b])

Discussion The goal of the evaluation was to derive educators' requirements for an explorable authoring system – we formulated a set of requirements and specific features that were either commonly used in the workshop already or frequently requested (Table 5.3). A core strength of Lumi emerges from the results: Being usable on any operating system, teachers can distribute authored content flexibly, either within or without an LMS, achieved by a file-based approach with different export formats. The results also suggest that mixing different content types from a broad selection is desirable, and H5P already provides many content types – yet the lack of usability features such as copy & paste or drag & drop of external resources hinders the remixing workflow. The evaluation identified multiple general usability issues, foremost the lack of a live preview, causing teachers to lose track of their current activity. Furthermore, the results align well with those of Serth et al. [Ser+19], confirming the centrality of enabling the remix workflow for such systems. Note that the requirements found are intended to be general, but subject-specific requirements may also be worth exploring [Now24].

5.2.2 Explorable Authoring Requirements

In this section, we will introduce a systematic catalog of requirements for authoring explorables. This catalog (compare Fig. 5.3) is a synthesis of the empirical results of our requirements study, work by Serth et al. [Ser+19], the ISO 25010 characteristics [ISO23], and our analysis of the related work (Chapter 3). The catalog has two main perspectives: Authoring and development. Authoring refers to the editing of documents (corresponding to the authoring paradigm perspective, see Section 3.2, and development refers to the editing of extensions (corresponding to interactive elements, see Section 3.1). For each perspective, there is both the quality of functionality (which enables expressiveness) and the quality of usability to consider, resulting in the four requirements categories of *authoring functionality*, *authoring experience*, *development functionality*, and *development experience*.

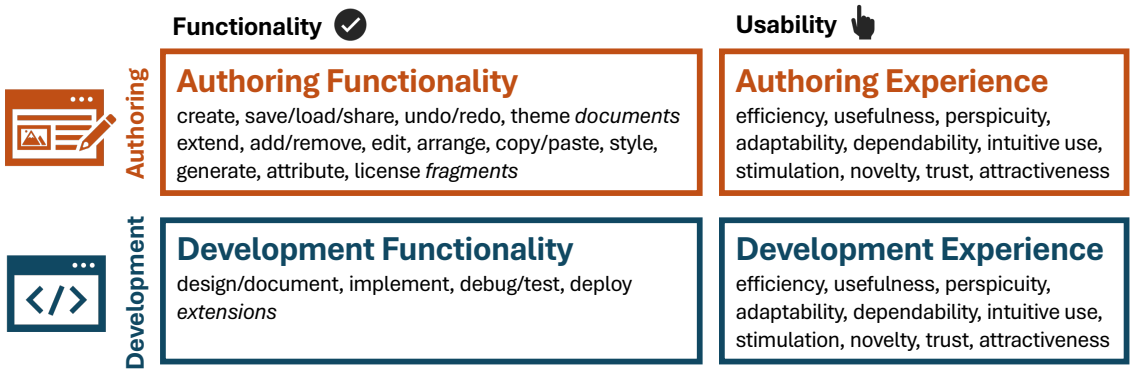


Figure 5.3: The explorable authoring requirements model summarized

Authoring Functionality The authoring functionality category reflects the functional suitability characteristic of ISO 25010 [ISO23]. It also makes use of the domain terms *document* (structured document, see Section 3.2) and *fragment* (part of a structured document) to define a set of functionality intended to cover the needs observed in the requirements study, as well as adding some "common sense" requirements for completeness:

EAR-AF1 *Create*: Make new documents[1]

EAR-AF2 *Save/load/share*: Store/restore document as/from external source

EAR-AF3 *Undo/redo*: Restore a previous document state

EAR-AF4 *Extend*: Register new kinds of elements in the document

EAR-AF5 *Add/remove*: Add fragments to the document

EAR-AF6 *Edit*: Modify fragments

EAR-AF7 *Arrange*: Reposition fragments in the document

EAR-AF8 *Copy/paste*: Duplicate/add fragments

EAR-AF9 *Style*: Change the appearance of a fragment

EAR-AF10 *Generate*: Automatically generate a fragment from a prompt

EAR-AF11 *Attribute*: Annotate a fragment with an attribution

EAR-AF12 *License*: Annotate a fragment with licensing terms

[1] Each requirement has an identifier: EAR stands for explorable authoring requirements. The character after the dash is either an A for authoring or a D for development. The final character is an F for functionality.

The first requirements of the model are typical and straightforward for document-based work (AF1-AF4): Documents must be *created* and *saved/loaded* (in a file-based approach) or *shared* (in a service-based approach). Another universal requirement is *undo/redo*, As we established widgets as a content category, widgets also need to be made available for a document before they can be authored and remixed further, which we call *extending* (AF4). The main part of the requirements concerns the manipulation of fragments (AF5-AF9). The complexity of an authoring tool lies in the multitude of element types (see Table 5.4): While *adding/removing*, *editing*, *arranging*, *copying*, *pasting* and *styling* text is straightforward, this is significantly more difficult for trees of hypermodal content and/or interactive content. Simply put: Each function should work with every element type. To account for the emergence of LLM models, we also introduce *generate* as a functionality (AF10). For OER, we add *attribute* and *license* (AF11, AF12), since remixing content may legally or ethically require to attribute the original author, and in turn, content may need be licensed to enable reuse itself.

Content Type	**Description**
Metadata	Information about the document such as title, author, etc.
Plain Text	Text without any formatting
Rich Text	Bold, italic, underlined, strikethrough, link, etc.
Headings	Several levels of headings for to create a content hierarchy
Sections	Semantic content blocks such as headers, footers, paragraphs, etc.
Lists	Ordered or unordered lists of items
Tables	Tabular data with rows and columns
Forms	Interactive inputs such as text fields, checkboxes, etc.
Disclosures	Toggleable elements such as dialogs
Formulas	Mathematical formulas (MathML)
Graphics	Vector graphics (SVG) and graphics canvas
Images	Embedded image media
Audio	Embedded audio media
Videos	Embedded video media
Documents	Embedded documents (web, PDF)
Scripts	Code to execute on the document
Themes	Styles to apply to the whole document
Widgets	Custom interactive content (with JS/Web Components)

Table 5.4: Content types of the web platform

Development Functionality Previously, in our authoring functionality requirements, we included extending (EAR-AF4), which is registering new kinds of elements in the document. Creating these elements is largely a process of software development, and similar requirements apply. Software engineering research and practice offers many different development processes tied to different methodologies, such as the waterfall model or agile approaches such as SCRUM [Som11]. Because of the multitude of approaches to software engineering used today, authoring tools and their

extension interfaces should be agnostic as to which one is used. Instead of prescribing specific approaches, authoring tools should focus on providing technical support for a common denominator of core functionality.

The Rational Unified Process (RUP) model introduced earlier (Section 5.1) defines a set of processes ([Som11]) which we can use as a starting point for finding this common denominator. For the purpose of creating extensions for explorable authoring systems, we can disregard the use case modelling, requirements, and project management processes here as they are independent of any interface provided by a tool. We also view configuration and change management as external to the authoring, as for projects such as extensions, this is commonly achieved with a version control systems such as Git. Furthermore, we do not consider tooling as its own requirement, but rather as a part of supporting the implementation. Finally, we extend the design process to include documentation.

EAR-DF1 *Design/Document*: Model the architecture and functionality of an extension

EAR-DF2 *Implement*: Realize the functionality of an extension with code

EAR-DF3 *Debug/Test*: Find and fix issues with an extension manually or automatically

EAR-DF4 *Deploy*: Publish, update, deprecate and remove extensions

Authoring & Development Experience We want to closely align the design of our authoring and development interface with modern user experience (UX) principles. An established model that is practically useful can be derived from the User Experience Questionnaire (UEQ). For more specific applications, a modular framework to construct questionnaires based on the UEQ is available, named UEQ+ [Sch21]. This both extends the scales applicable to every system (with usefulness, adaptability, intuitive use, and trust), and introduces system-specific scales, for example for GUIs or voice assistants.

A consensus in evaluating the developer experience (DX) has yet to emerge. Lee & Pan [LP21] outline that DX is in fact a special case of UX. This insight explains the large overlap between models of DX and models of UX. Consider their questionnaire based on the model of Fagerholm & Münch [FM12] and several studies from the area of acceptance research: In the questions listed, we can find almost every aspect of the UEQ+. For example, one aspect of the UEQ+ is efficiency, which appears in Lee & Pan's items 21 ("it improves the efficiency") and 15 ("the platform is fast").

Scale	Publication	Items	Subscales
After Scenario Questionn. (ASQ)	[Lew95]	3	0
System Usability Scale (SUS)	[Bro96]	10	0
Post-Study System Usability Questionn. (PSSUQ)	[Lew02]	19	3
Net Promoter Score (NPS)	[Rei03]	1	0
User Experience Questionn. (UEQ)	[LHS08]	26	6
Single Ease Question (SEQ)	[SD09]	1	0
Subjective Mental Effort Questionn. (SMEQ)	[SD09]	1	0
Visual Aesthetics of Websites Inventory (VisAWI)	[MT10]	18	4

Table 5.5: Usability Scales (adapted from [Hin+19])

Given this research into UX and DX, we can make use of the idea that developers can also be seen as users. Developers of an extension for an authoring tool are, in other words, users of a specific subset of functionality the tool provides. Namely, they use the documented interfaces and parts of the tool itself to complete their task of developing an extension. For example, while perspicuity (ease of learning) in the context of the authoring experience may be achieved by a clearly designed UI with helpful tooltips, in the context of the development experience, it may mean adding type hints and documentation for an API. Making use of this perspective and considering the UEQ+ model, we can divide our users into authors and developers and formulate parallel authoring and development experience requirements:

EAR-AX/DX1 **Efficiency**: Develop/author without unnecessary effort

EAR-AX/DX2 **Usefulness**: Fitness of purpose of the tool for authoring/development

EAR-AX/DX3 **Perspicuity**: Ease of learning to develop/author with the tool

EAR-AX/DX4 **Adaptability**: Customizing the tool for authoring/development

EAR-AX/DX5 **Dependability**: Having control of the tool for authoring/development

EAR-AX/DX6 **Intuitive use**: Immediate usefulness of the authoring tool for tasks

EAR-AX/DX7 **Stimulation**: Fun while using the tool for authoring/development

EAR-AX/DX8 **Novelty**: Creativeness of the design of the tool for authoring/development

EAR-AX/DX9 **Trust**: Safekeeping of documents/code when authoring/developing

EAR-AX/DX10 **Attractiveness**: Overall impression of authoring/development

5.3 Concept

From the perspective of our user groups (authors and developers), the explorable authoring system `WebWriter` [Sal25b] has several components: The website[2] introduces the project and offers access to the documentation (for developers) and examples of packages and explorables (for both educators and developers). Also, there is the web application[3] (used by educators for authoring and by developers for testing). Finally, the source code, examples and project management such as issue reporting are available on `GitHub`[4] (for developers).

The specific concepts we aim for in our research questions (the design pattern and the authoring paradigm) are not addressed in detail here. Refer to the next chapters (Chapter 6, Chapter 7), instead.

5.3.1 Authoring Interface

Generally, `WebWriter` is a file-based authoring tool: Authors create documents through a user interface which can be saved as self-contained HTML files and shared on other platforms. Documents consist of standard elements (e.g. paragraphs or images), and also custom elements distributed in packages (called widgets). Furthermore, packages can also offer predefined document fragments such as specific examples of a widget in context (called snippets) or document-wide styles (called themes).

Overall, the authoring interface is a web application, so the user's experience begins with them opening the application in their browser. They are presented with an interface reminiscent of other popular rich text solutions such as word processors. There is an app bar with buttons for general actions and information about the currently opened document at the top. Below, in the center, the current document is displayed in an editable state (starting out, a blank page). On the left (or at the bottom on small screens as the app is responsive) is the palette, which lets the author choose and manage elements to insert. On the right (or hidden behind a right click or button on smaller screens) is the toolbox, which shows the location of the current selection in the document and offers ways to manipulate that selection. A few minor views (preview, settings, source & test mode) are hidden behind buttons in the app bar.

We will walk through each part of the interface and describe it in detail from the author's perspective.

[2] `https://webwriter.app`
[3] `https://run.webwriter.app`
[4] `https://github.com/webwriter-app/webwriter`

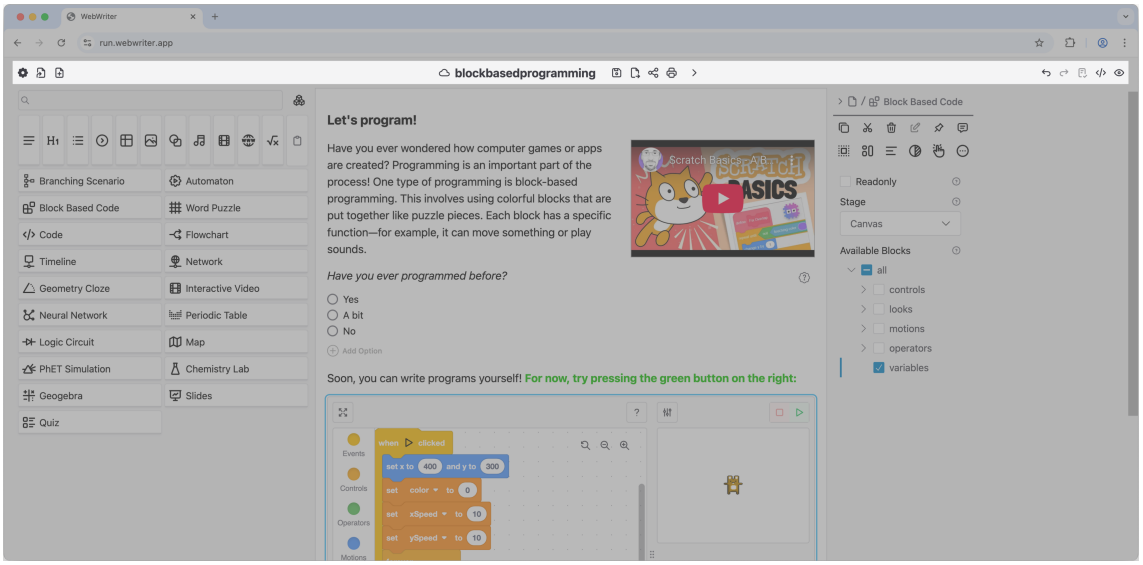


Figure 5.4: WebWriter' app bar with buttons for app-wide (left), document-wide (center) and editor-wide (right) functionality

App Bar The app bar (Figure 5.4) has a left, center, and right part. The left part contains buttons for actions at the application level: Accessing the settings and opening or creating a document (EAR-AF1, EAR-AF2). These actions usually lead to a new tab with another `WebWriter` instance to allow for a multi-document workflow. This is a step to use the available conventions and standards available in browsers, instead of creating a tab system unfamiliar to users from scratch. In the center, information and actions for the whole document are provided, such as the name, save status and location (saved locally or in the cloud), buttons for saving, sharing (for cloud documents) and printing (EAR-AF2), and finally access to the metadata editor (EAR-AF11, EAR-AF12). On the right are buttons for actions at the document editing level, which are undo/redo (EAR-AF3) and the preview. The preview button could also be sorted into the middle, but is so commonly used that we considered it helpful to place it in the corner to be easily clickable according to Fitts' Law [Mac92].

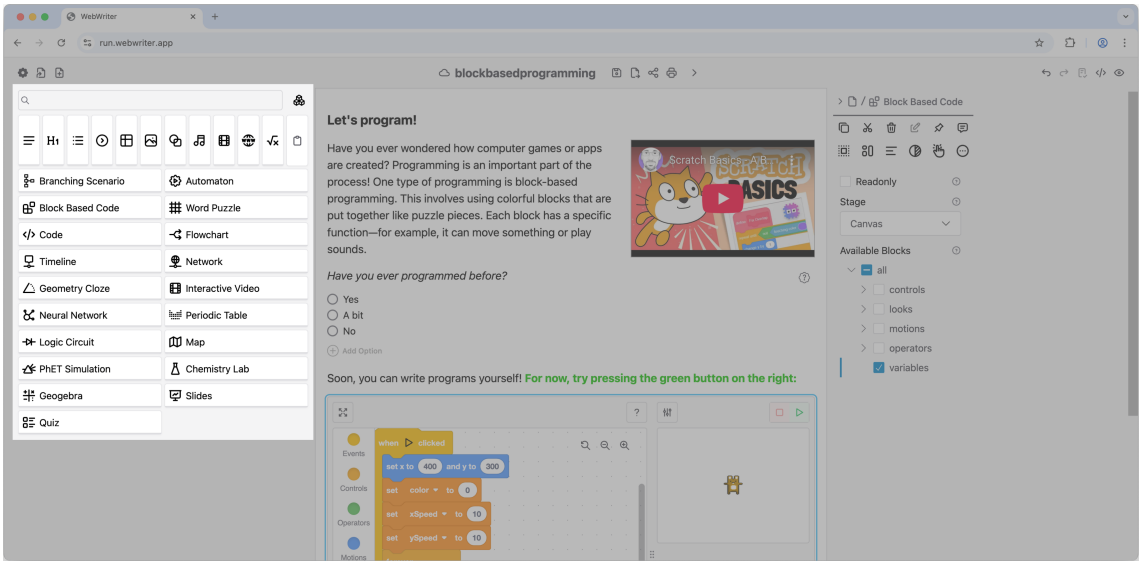


Figure 5.5: WebWriter's palette with the search and a button for management mode (top) built-in elements (middle) and insertable content from packages (bottom)

Palette The palette (Figure 5.5) has two parts: The top part (left part on small screens) contains a search bar and management mode button for packages (EAR-AF4). The rest of the palette shows buttons for each element that can be inserted (EAR-AF5). The standard elements (paragraphs, headings, lists, disclosures, tables, images, vector graphics, audio, video, websites, and formulas) are shown as icons only as authors are expected to become familiar with them quickly. Some elements are grouped with the button inserting the primary element, and a dropdown menu to select other elements in the group shown on hover, e.g. for different levels of headings. There is also an "paste" button here to paste from the system clipboard. Under the standard elements are the currently installed packages, whose widgets and snippets can be inserted with a click or again browsed through a dropdown on hover. If the search bar or management mode button is clicked, users can install new packages, update or remove existing ones, or hover over packages to get a description and metadata.

The palette uses NPM's search API to look for widget packages in real time. In the background, adding a package adds it to the list of installed packages, and this change rebuilds the document, automatically including the necessary scripts and stylesheets from each package. The palette underwent a few design iterations: Originally, managing packages was relegated to the settings view, but this was changed to make packages easier to discover. Over time, the list of built-in elements supported by `WebWriter` also grew, leading to a more condensed display with dropdowns for less used elements. Similarly, provisions were made for better searchability as the number of packages is constantly growing and unbounded (unlike the built-in elements, which are relatively fixed). The search bar and detailed metadata display are two measures added later for searchability.

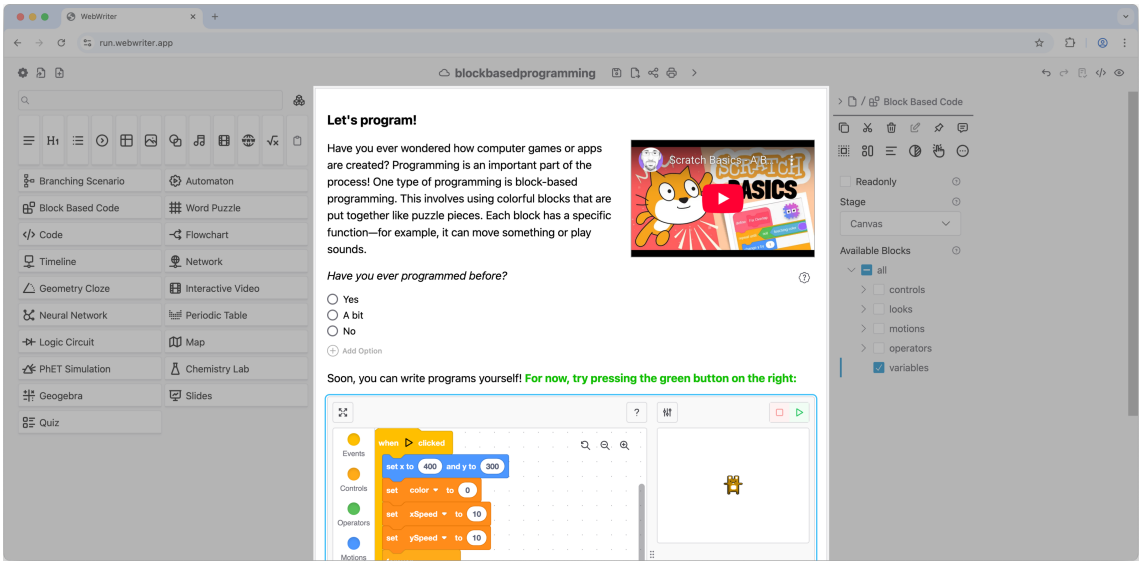


Figure 5.6: WebWriter's document view following the principle of What You See Is What You Get

Document The document itself (Figure 5.6) is displayed in the middle in an editable state similar to word processors (EAR-AF6, EAR-AF7, EAR-AF8). Following the "What You See Is What You Get" (WYSIWYG) principle, the document is displayed almost exactly as the end users (e.g. students) will see it – only a representation of the current selection is added to the document, which is a caret or text highlight in case of a text selection, or a blue highlight border in case an element is selected. Widgets may also break the WYSIWYG principle by showing custom affordances when they are being edited (e.g. a button to add a new slide in a slide show widget).

From a technical perspective, this part of the interface represents the HTML `<body>` of the document being authored. Over the development process, there have been very few changes to the fundamental concept of the document editor.

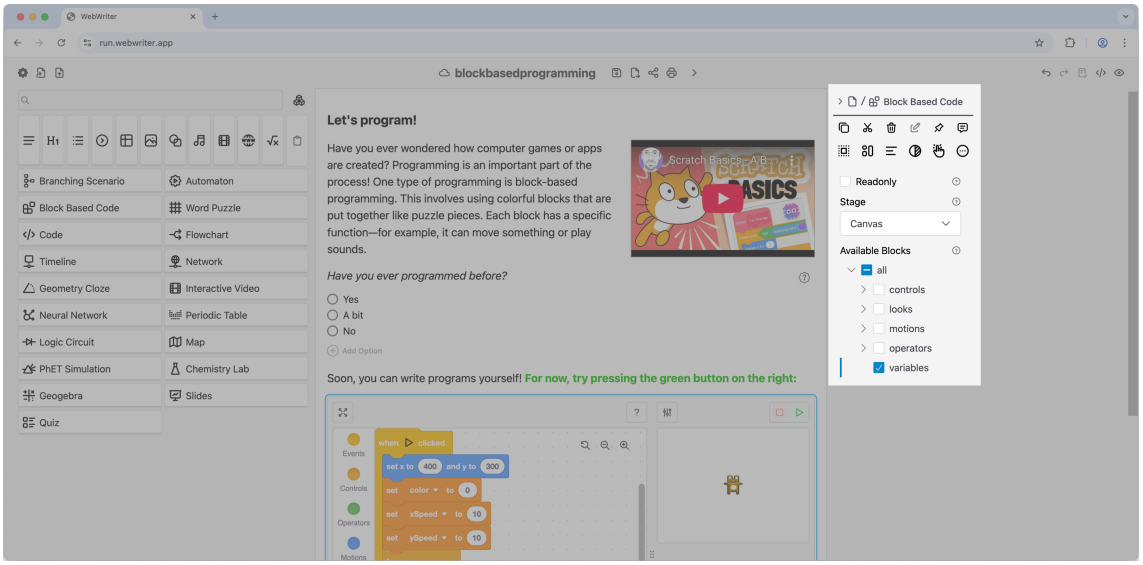


Figure 5.7: WebWriter's toolbox with a breadcrumb representation of the selection (top), general options for any selection (middle), and context-dependent options (bottom)

Toolbox The toolbox (Figure 5.7) is split into three: At the top is a breadcrumb (path) representation of the current selection. This can be expanded with a button to display a complete outline of the document. Below the breadcrumb, several buttons to manipulate the selection are positioned: Actions common for rich text editing such as copy, cut, and delete, come first (EAR-AF7). Next are special editing actions such as pinning the selection (saves the selection as a snippet in the palette to be reused later) and adding comments for other authors. Finally, there are several buttons leading to menus for advanced styling options to be applied to all selected elements, such as for borders, margins, colors, text alignment, and so on (EAR-AF9). At the bottom, the toolbox dynamically offers up actions depending on the selection. If text is selected, text formatting options such as making text bold or italic are offered (EAR-AF6, EAR-AF9). If a standard element is selected, options specific to that element are shown (e.g. for an image, the author can enter the source URL or embed a local file). Finally, if a widget is selected, the custom options defined by the widget are shown.

Technically, the toolbox offers a set of pre-defined commands to apply to the selection, and an alternative way to navigate through the document with the breadcrumb and outline. As such, it combines the form fill-in and menu selection interaction styles. The toolbox has evolved throughout the development process, especially as more features were added and the display had to be condensed. In the toolbox, screen space is very precious, as on narrow displays, it may also be displayed as a popup and should not cover too much of the document. This also serves the dual purpose of not overwhelming the user, as the toolbox is always displayed – for this reason, details are revealed only after more interaction, e.g. by accessing the styling dialogs or opening the outline view.

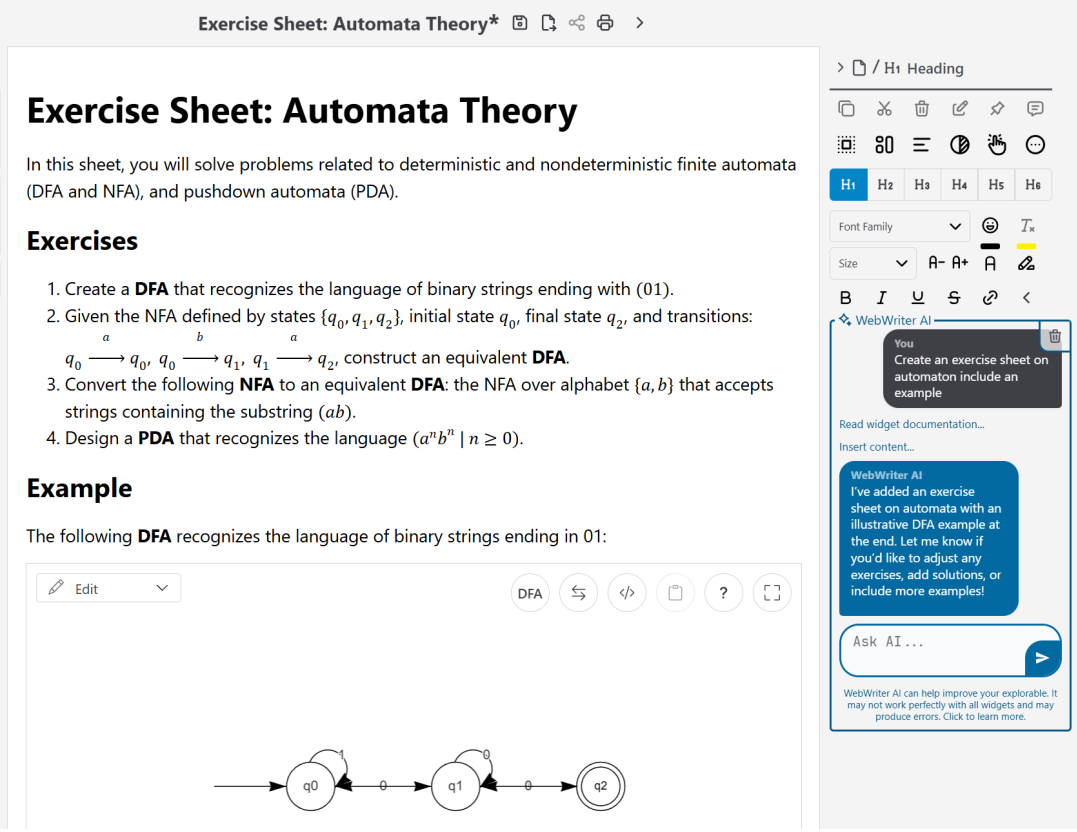


Figure 5.8: WebWriter's optional AI assistant in action

WebWriter AI The `WebWriter` AI assistant (Figure 5.8) offers an optional, agentic interface which can help educators generate content and modify documents (EARAF10). From a user experience point of view, the assistant is very simple and similar to state-of-the-art offerings such as OpenAI's `ChatGPT` [Ope22]: A chat interface is shown which allows the author to query the AI assistant, which knows the current document and has full capabilities to suggest changes. Crucially, the assistant can request widget documentation live and read it, enabling it to generate complex document fragments including widgets. As a general principle, no changes to the document are done without confirmation by the user.

On the technical side, the generation of content is enabled by HTML-based format of documents in WebWriter, with which interactive content can be expressed declaratively within static and hypermodal content: The HTML markup language becomes the underlying data format, which enables interoperability with other HTML-producing tools (e.g. other AI assistants such as `ChatGPT` generating rich hypermedia content to copy and paste).

The AI assistant is the only part of `WebWriter` developed by a student as part of a thesis [Fel26]. It builds on prior student work which looked at integrating LLMs for automatic spell checking and grammar correction [Kul24]. As of now, it is experimental and a work in progress.

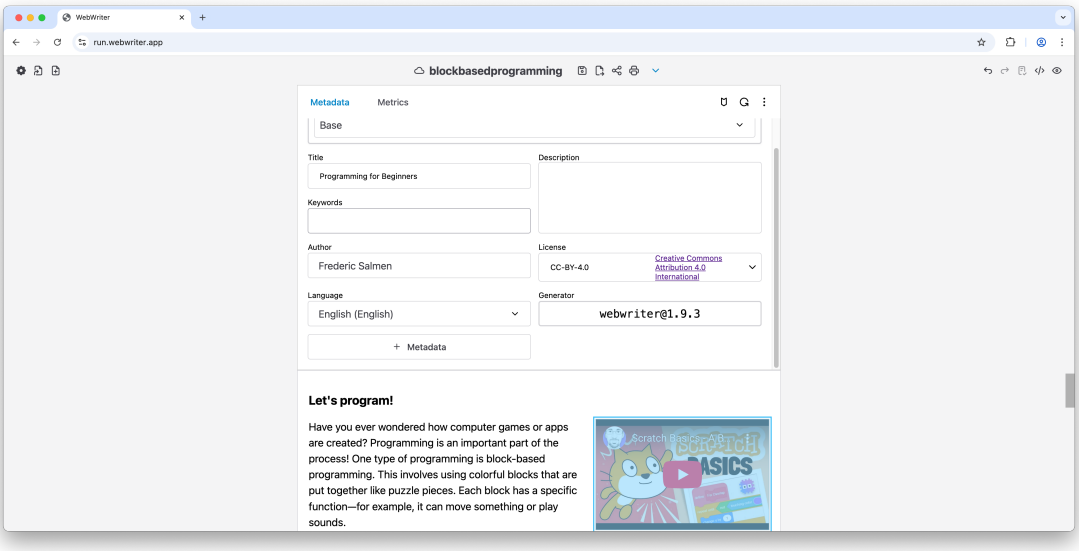


Figure 5.9: WebWriter's form-based metadata editor

Metadata Editor The metadata editor (Figure 5.9) is accessed through a button in the middle of the app bar. By default, it allows authors to add a few well-defined pieces of metadata (EAR-AF11, EAR-AF12): A title, keywords, a description, an author, a license, and a language. Arbitrary metadata can be added with a button, as well. Also, a theme can be selected which changes the default appearance of the document. There is an advanced mode which allows experts to add scripts and styles to the document, and an expert dialog to edit the attributes of top-level elements in the document (EAR-AF12).

On the technical side, the metadata editor allows editing the `<head>` of the document (while the main editor modifies the `<body>`). It uses form fill-in as the main interaction style, since it is well-defined in advance which types of metadata could be added – there are a few types defined directly in the HTML Living Standard [WHA25]. Also, the head mostly consists of key-value pairs, with only the very specific feature of the `<noscript>` element allowing for nesting of content (`<noscript>` marks sections which are only shown if the user has disabled JS completely). That element is realized as a container into which other the key-value input fields can be moved. The expert dialog to edit attributes modifies the `<html>`, `<head>`, and `<body>` elements. The metadata editor is a lesser used feature and has remained mostly unchanged throughout development.

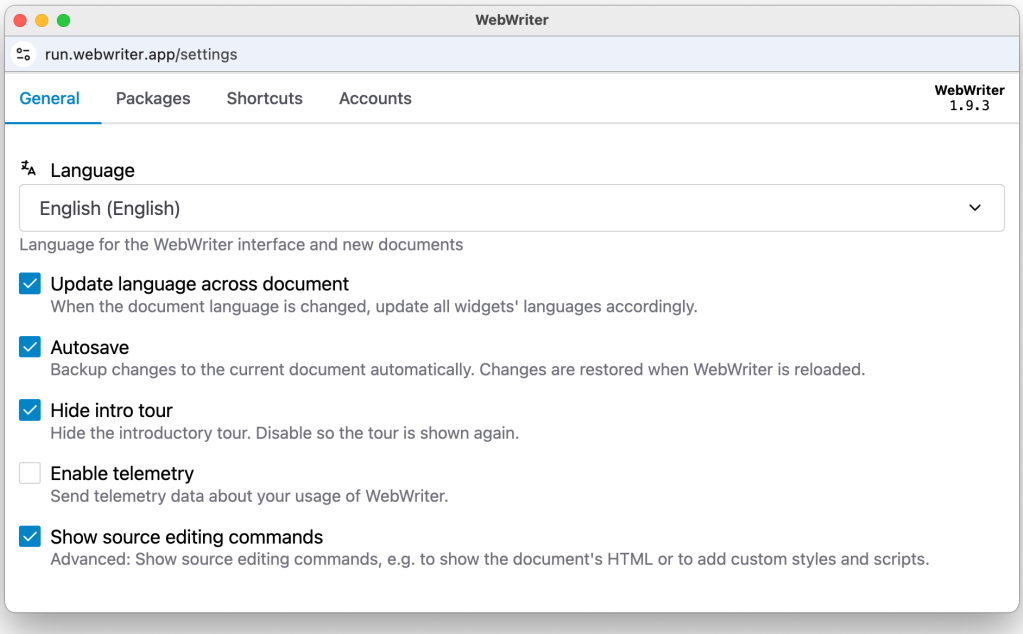


Figure 5.10: WebWriter's settings page with general settings such as the interface language shown

Settings The separate settings window (Figure 5.10), realized as a pop-up, offers several options to personalize the authoring experience. For example, an interface language can be set (English and German, and also 31 other languages in machine translation via `DeepL` [Kut17]), which is also applied to new documents. Furthermore, there is a complete overview of available commands with descriptions, and also keyboard shortcuts for each of the commands that can be reconfigured. Finally, there is a panel to manage remote accounts (currently only for the built-in cloud functionality).

A few small features were not mentioned, such as the preview, or the guided tour when first starting `WebWriter`. Also, some general notes apply across all features of the authoring interface: The design is fairly minimal, with the editor front and center as the focus of the author's attention, which shows the status of the document at all times. Real world metaphors and system conventions are used where possible, such as "palette" (from painting, and commonly used in GUI tools), "toolbox" (also common in GUIs), and "snippets" (handicraft with paper), also visually with the paper-like document. Dialogues can easily be cancelled and every editing action that changes the document can be undone. An icon set with a familiar visual language for common actions is used, e.g. a floppy disk icon for saving. Almost every user interaction occurs in the main view, so the most important information, which is the current document state, is visible and does not need to be recalled. Tooltips are available as additional help to understand what buttons do and what keyboard shortcuts are available for experts.

5.3.2 Development Interface

The development interface is not a single entity but several related building blocks which allow developers to implement interactive elements (widgets) which can be published and used within `WebWriter`.

In general, developers would start by reading the provided documentation, which guides them to all other building blocks of the development interface. They may view examples on the website's package gallery, set up their project using the developer tools, and import their package locally in the `WebWriter` app for testing. As such, the workflow is intentionally similar to other web development experiences.

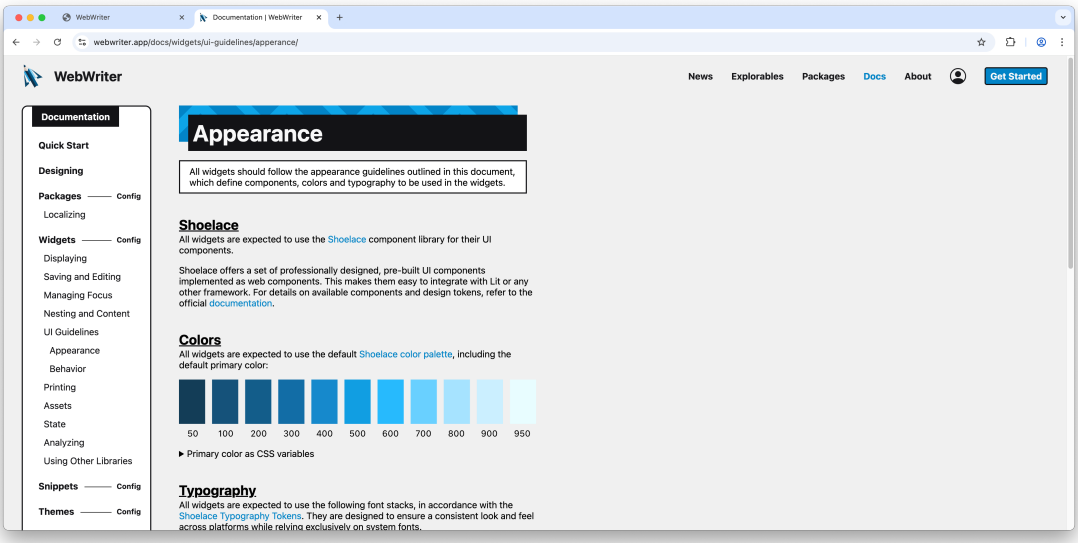


Figure 5.11: WebWriter's documentation on the website for developers, with an example subpage about widget design shown

Documentation & Examples The documentation (Figure 5.11) available on the website guides developers through the whole development process and is the starting point for all projects. The first page is a quick start guide which helps developers create a minimal package as fast as possible. The rest of the documentation is ordered by the typical steps developers take during their project, from designing over implementing specific features of widgets, snippets and themes (see Chapter 6), to documenting, testing, and publishing their package. Each page contains example code for how to implement certain features. There are also "config" pages to thoroughly document the formal interface between `WebWriter` and the package. All packages developed so far are shown as a gallery (Figure 5.12) on the website with a live sandbox, and links to the source code are provided for each. This can be used by developers to access real-world examples for packages to supplement the simplified examples in the documentation.

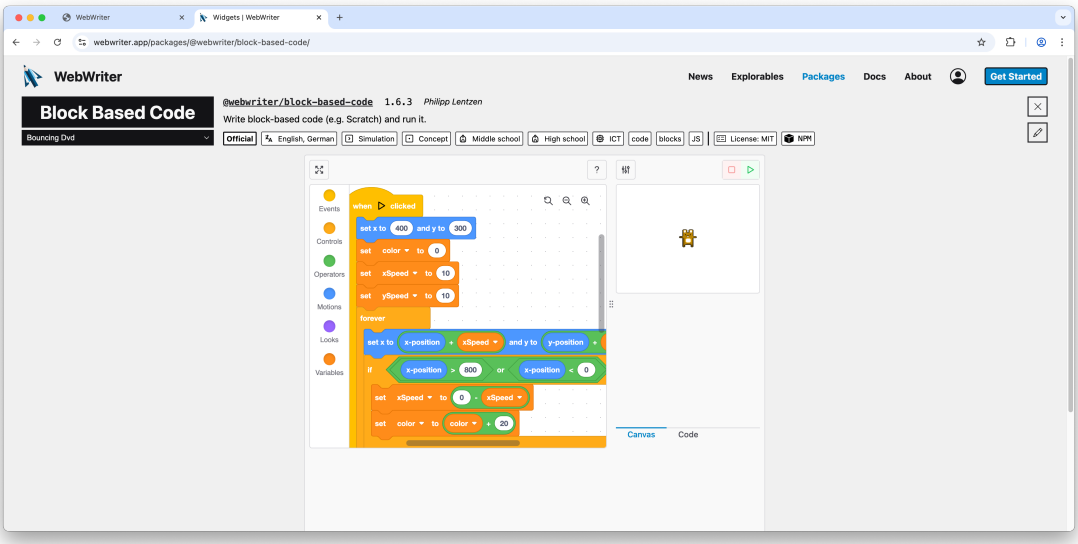


Figure 5.12: An example in the packages gallery on WebWriter's website with metadata and a live preview

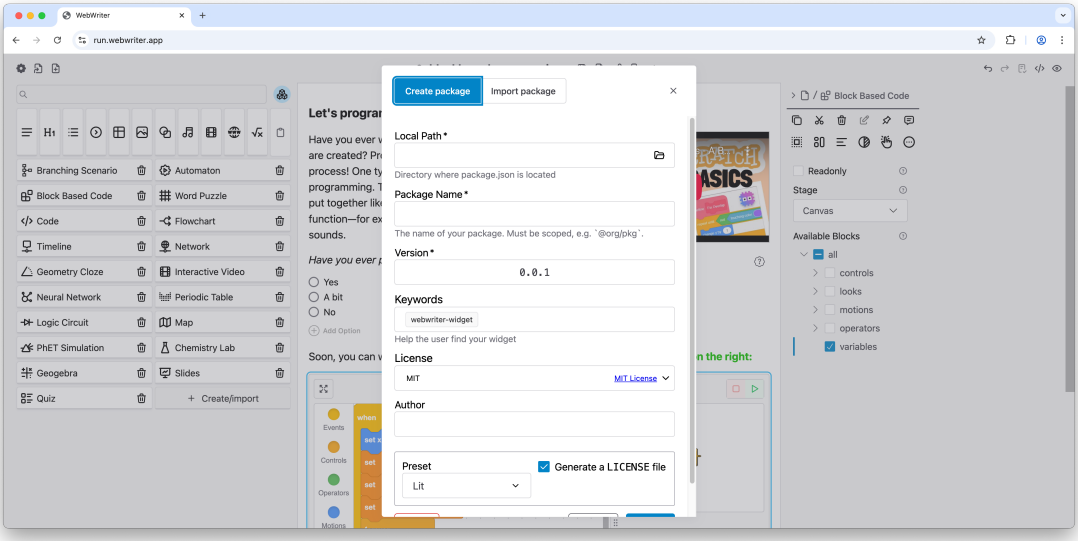


Figure 5.13: Form to create or import a local package into WebWriter

Local and Live Packages Developers also make use of part of the authoring interface, as they can add packages (Figure 5.13) from a local directory. A package added that way is shown alongside other installed packages and can be inserted normally. As a special feature, if the source of the local package changes, it is automatically reloaded and updated in the document (live reloading), allowing for a developer workflow where `WebWriter` serves as the live preview for a developed widget. Additionally, there are advanced options available for local packages, such as a testing mode where developers can run predefined unit tests on a widget in the `WebWriter` environment,

or the source mode, where developers can see the source code of the current document to help with debugging. Those advanced options need to be enabled in the settings first, as they are not useful for authors and might lead to confusion.

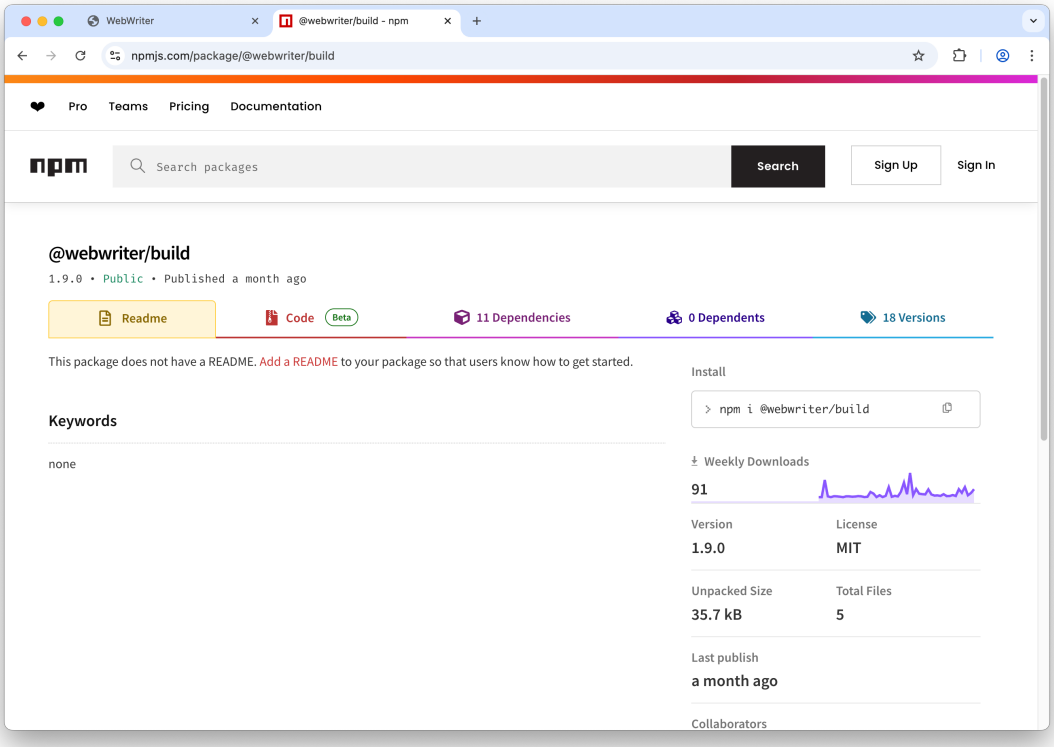


Figure 5.14: NPM page of `@webwriter/build`, the main developer tool offered

Optional Tooling The build tool [Sal25c] is an optional helper developers can use for common tasks. It is a command line tool that developers can install on their machine, running commands for their project. It offers three main features (Figure 5.14): First, through an integration of the tool `esbuild` [Wal20], it can bundle widgets in the package according to the exports specified in the `package.json`, a necessary step both for testing locally and for publishing. For local testing only, a watch mode is also available which re-bundles the code each time the source is changed – in combination with the live reloading of local packages, this means the widget in `WebWriter` is updated automatically each time a source file is changed. Another optional helper is the component helper library [Sal25d] which offer a starter class with some common functions to build on for developer convenience. Both the build tool and the component helper library are optional and exist only as shortcuts to shorten development times. In fact, as we will explore in the next chapter (Section 6.1), the implementation of our design pattern in `WebWriter` means that widgets can be implemented with any web technology, as they only rely on web standards.

5.4 Implementation

On the technical side, the concept is realized as a web-based implementation. We will discuss the overall architecture and the technology stack. We do not look at every technical detail in depth[5], but present several major challenges faced in the implementation which may be valuable to consider for similar projects.

5.4.1 Architecture

One dimension of architecture, specifically on the web, is how to distribute responsibilities between the client and the server – if there is a server at all. This is the distinction between single page applications (SPAs) and multi page applications (MPAs). Traditionally, technology dictated that only MPA were practical for complex applications, as the frontend technologies of HTML/CSS/JS in tandem with the web platform were not powerful enough to implement many features. Modern web technologies starting with HTML5 allowed developers to express business logic on the frontend, eliminating the need to involve the server at all for many business tasks. Overall, we follow the SPA approach (see Figure 5.15) as well, for three reasons: First, development complexity is reduced as there is no network or technology boundary between different parts of the system, speeding up prototype development. Second, as educators often work with more sensitive data of learners, a local-first SPA offers less of an attack surface for privacy violations. Third, the system becomes easier to deploy, making the prototype more sustainable according to the FAIR4RS principles.

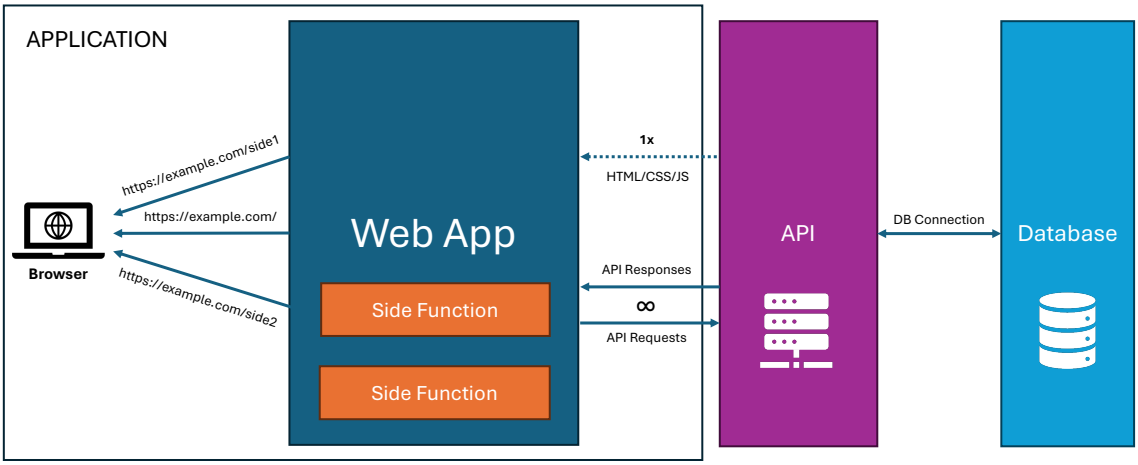


Figure 5.15: The single page application architecture on the web

The most high level view of the implementation may be to consider how data flows through the system – at this level, we can distinguish between several possible architectural design patterns Table 5.6 which divide responsibilities between different system components. But why use a specific architecture rather than improvising one? Simply put, all architectural patterns considered here help realize the principle of the separation of concerns [Dij82]: Each part of a code base should address a single concern. Following this principle achieves modularity and makes it easier to reason

[5] Refer to the repository for specifics: `https://github.com/webwriter-app/webwriter`

about parts of a software project in isolation (borrowing from LLMs, this decreases the context window necessary for working on that part of the software). Code that is easy to reason about is easier to change, maintain, and extend. And while our prototype needs to be fairly unique in some parts of the implementation (see Chapter 6 and Chapter 7), it is mostly a conventional graphical user interface application (GUI): Educators interact with a graphical user interface to manipulate a representation of a structured document, which can be persisted on the file system. So we can observe that innovation mostly happens on the lower levels of implementation in our case. On the architectural level, we can make use of established patterns to reduce complexity and separate concerns.

There are many architectures for GUI applications (compare Table 5.6), with new ones constantly emerging (for example Martin's "Clean Architecture" from 2017 [Mar17]). Choosing an architecture may mostly come down to preference, as different developers prefer different architectures and consider them more or less helpful in separating concerns. There are a few deciding factors, however: Some architectures are fairly complex, with many different layers (PAC, MVI, Clean) – this is probably too much incidental complexity to introduce into our project, as our prototype is small to medium scale. While they bring interesting benefits in debugging and testability [Sta15], the three major unidirectional patterns (Flux/Redux, MVU, MVI) we find too restricting and verbose as they enforce a functional style in every situation. As the object-oriented paradigm is far more common in the web development world, we are further concerned this would create a tension in using the web platform and external dependencies. That leaves us with the "model-view-something" family of architectures, which divide presentational logic and business logic between the view and the model, and connect both with "something" in between.

For WebWriter, the Model View ViewModel (MVVM) pattern was chosen: There is a data *model (M)* which contains all business logic and data. Then, there is a *view (V)* which is responsible for the presentation of the application and receives user inputs. It does not directly depend on the model, but is connected to it through the *view model (VM)*, which contains additional view logic and most importantly binds data from the model to the view. Overall, this means that the model and view are decoupled, and the view model provides an interface between both. The advantage is that both model and view can developed and tested mostly independently.

Pattern	Description	Discussion
MVC	Model (data, logic) ↔ Controller (bridge) ↔ View (presentation, input)	
[BTD16]	Manual/observer binding; Flow often bidirectional; Model notifies View; View may query Model	Well-known; clear separation of concerns; Controller / View bloat; ambiguous responsibilities in web UIs
MVP	Model (data, logic) ↔ Presenter (supervisor) ↔ View (presentation, input)	
[Pot96]	Manual binding, typically bidirectional flow via Presenter mediation	High testability (good for mocking); Presenter bloat; boilerplate
MVVM	Model (data, logic) ↔ ViewModel (mediator) ↔ View (presentation, input)	
[FSB25]	Data binding and observables, one-way flow from ViewModel to View or two-way data binding for inputs	Testable; clear state projection; suits declarative UIs; Hidden coupling and complexity in bindings
PAC/HMVC	(Presentation ↔ Control ↔ Abstraction) components in a hierarchy	
[CCN97]	No binding (composition, internal observers), hierarchical flow: parent–child agents/modules	Scales to large UIs; modular decomposition; Coordination overhead; increased complexity
Flux/Redux	Action → Reducers → Store → View	
[Gar18]	One-way subscription binding, strictly unidirectional flow	Predictable state transitions; time-travel/debuggability; Verbosity; normalized state complexity
MVU/TEA	Model → View (pure) → Update → Model	
[Fow20]	One-way binding, unidirectional loop	Strong guarantees; simple mental model; single source of truth; Forces functional style; learning curve
MVI	Intents Stream → Interpreter → ViewState → Render → New Intents	
[Sta15]	One way binding via reactive streams, unidirectional flow	Suits async/reactive UIs; reproducible state (testable); Event explosion if undisciplined; complexity
VIPER/Clean	Entity ↔ Interactor ↔ Presenter ↔ Router/View	
[Mar17]	One-way binding across interfaces, layered, often unidirectional flow across boundaries	Strong separation of concerns; high testability; Ceremony/boilerplate; risk of overengineering

Table 5.6: Overview of graphical user interface application architectures

To summarize the architecture: `WebWriter` is a web-based single page application (SPA) structured with the Model View ViewModel (MVVM) pattern. It is mostly self-contained with no required `WebWriter` server component, but with a dependency on external services (NPM, CDNs).

5.4.2 Technology Stack and Components

As a single page web application, the format of deployment is set: HTML, CSS, and JS. However, this does not force us to use only these technologies in development. As is nowadays common in web development, we make use of a build tool (`Vite` [You20]) which allows us to write code in other languages and compile it to our format of deployment. Furthermore, we do not limit ourselves to the standard library afforded by the web platform, but use external packages from NPM in development. Packages from NPM enable us to plug in popular libraries into each layer of our architecture, reducing the complexity of developing each layer further.

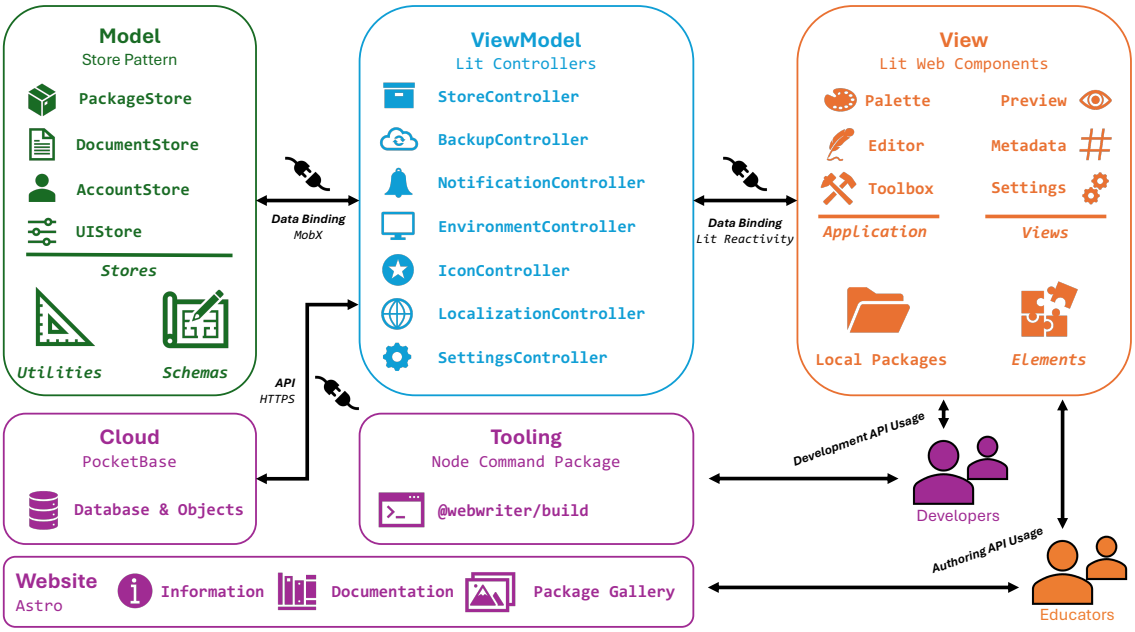


Figure 5.16: WebWriter's architecture in components across the Model, ViewModel, and View layers, with external components shown

Language (TypeScript) While modern JS is sufficient to implement any application, a fundamental property that is not bound to change is that JS is dynamically typed. We feel that our application is sufficiently complex to benefit from static typing however, as it would allow us to properly specify data types and interfaces between our layers. Furthermore, there is tacit evidence showing benefits of static over dynamic typing in avoiding errors and maintainability [PT98; Kle+12; Ray+14]. On the web, the default choice for static typing is Microsoft's TypeScript (TS). TS can be understood as a superset of JS: All valid JS is also valid TS. This allows us to add static types where they are useful, but also to occasionally break out of the static typing paradigm if necessary. Alternatives other than TS or JS (e.g. Elm, PureScript, Reason) are

fairly rare in practice (used by less than 1% of developers according to the 2025 StackOverflow developer survey [Sta25]) and might disconnect us from the rich JS/TS ecosystem (especially NPM), which we judge to offer far more value than specific language features.

View Layer (Lit Web Components) The view layer is abundant with solutions and sports the most well-known frameworks in web development. Nowadays, all significant frameworks follow the component pattern: The view is divided into a hierarchy of component instances, meaning instances can be used in multiple parts of the view or even reused by other applications. The biggest frameworks to consider are `React`, `Vue`, and `Angular` (Section 2.2). `Svelte` [Har+16] is also a notable new contender, which, similar to Vue, uses a custom `.svelte` format combining HTML, CSS, and JS to express components, that is compiled into JS. However, we consider each of these popular solutions problematic in terms of sustainability: Each framework introduces a strong dependency on that framework, making our application a `React/Angular/Vue/Svelte` application (this is especially true for `Angular`, which forces design decisions in the Model and ViewModel layers, as well). Any component we develop can only be reasonably used in that same framework, not in others. This is problematic as web frameworks are famously fast-lived (see our description of web components at Section 2.2), so our prototype may become outdated fairly fast, violating our goal of FAIRness.

However, in this case, we choose a lesser known solution: Web components (Section 3.1). Web components have the key advantage of standardization, meaning that a web component implemented today is bound to stay usable in the long term. Furthermore, web components can be used in all popular frameworks. To write web components, we use the small library `Lit` [Fan17] by Google, which offers class-based and declarative approach for creating web components. It is very performant and easy to reason about. For our prototype, this means that each component can be used and reused indefinitely: If `WebWriter` is transitioned into a different view library, specific components may keep being used without change. Also, other researchers can reuse the documented components in their own prototypes without having to adopt our technology stack.

As a convention, we also separate two types of components (often called "smart" and "dumb" components): There is a tree of complex (smart) components which depend on and access the ViewModel (`<ww-app>`, `<ww-explorable-editor>`, `<ww-metaeditor>`, `<ww-palette>`, and `<ww-toolbox>`). These also process events and contain advanced view logic. In general, they are not standalone and assume a certain DOM structure (i.e. `<ww-explorable-editor>` assumes its parent is `<ww-app>`. Then, there is a large set of simple (dumb) components which are used inside the main tree of smart components (i.e. a `<ww-button>` component with custom styling and a tooltip with the current keyboard shortcut of the command). These components are standalone and make no assumption about the DOMs structure, as all necessary data is passed to them with attributes and properties. For example, `<ww-button>` is used in the palette, toolbox, and app bar, among other places. The advantage of this separation is the dumb components are easy to test in isolation and can be used in

different places across the application, or even reused by other developers. The small amount of smart components on the other hand allows us to co-locate all view logic in a few central locations, making it easier to reason about.

Model Layer (Store Pattern) For the model layer, we work without a specific technology or framework. Since we use `MobX` [Wes15] on the ViewModel layer (see below) for communication between the Model and the View, this allows us to cleanly separate the model layer. Instead, we use a design pattern also recommended by `MobX` which is the Store pattern. In the store pattern, the state of the application is stored in singleton instances of store classes, which store all runtime data of the application and provide an interface to operate on that data. Each store is assigned a responsibility for a specific type of data. `WebWriter` currently implements four stores and a root store. The `DocumentStore` contains all business logic regarding the current document and editing state. The `PackageStore` is responsible for creating, reading, updating and deleting operations (CRUD) on packages that can be installed via the palette. The `UIStore` is mostly a data container for UI-related settings and is not very complex. Finally, the `AccountStore` can manage different kinds of credentials used in `WebWriter`. Currently, this is only used for the optional `WebWriter` cloud. All stores are part of the root store, which also contains some generalized methods applicable to all stores to set settings (e.g. changing the language from `en` to `de`) and persist them in LocalStorage.

Additionally, the model layer contains some standalone business logic modules used in the different stores, such as parsers and serializers for different file formats, clients for different storage backends (currently `file` and `pocketbase` - `WebWriter` Cloud), and schemas for different data types which allow us to parse user-provided data (i.e. a `package` schema for `WebWriter` packages).

ViewModel Layer (MobX) As our view layer written in Lit is reactive (updates when data changes), but our data lives in the model in multiple stores, we need a way of updating the view whenever part of the model changes. For this, we use `MobX`, which is mainly a library to implement the signal (or observable) pattern in JS. This layer is very simple in `WebWriter`, as `MobX` integrates perfectly with the store pattern. At the heart are a few lines of code with `MobX`'s `makeAutoObservable` and a listener which accomplish multiple things: First, all properties of the store, which contain the data (e.g. a string property for the current user interface language), are made observable, meaning that if a change in these properties is registered, the listener is called. Second, all getters become computed properties, where their result is cached until the state they depend on changes. Third, all methods and setters become actions, which are run inside transactions and are the only way to change state in the store. When a change triggers the listener, the view (with `Lit`) is prompted to rerender the whole app tree. That may sound performancy-heavy, but `Lit` takes care to highly optimize rerendering in that DOM nodes are retained and pinpoint updates are performed by default. For example, changing the language in the settings would trigger the listener and cause a full update, but components unaffected by the language would ignore the change completely.

To integrate with the view, the ViewModel is implemented as a mixin which is applied to the main app component. It adds the root store in its observed version to the app so smart components can trigger changes by calling action methods and setters. It also adds several controllers for advanced view functionalities: Notifications, icons, commands, environment, localization, settings, and a service worker-based API necessary for the package system. Overall, this forms a full application interface which the view's smart components can make use of.

Rich Text (ProseMirror) A library that operates across all layers mentioned (and all layers shown in Figure 5.16), but that is integral to `WebWriter`, is `ProseMirror` [Hav17]. ProseMirror, a JS library from well-known developer and author Marijn Haverbeke (Eloquent JavaScript [Hav19], `CodeMirror` [Hav11]), is a framework for building rich text editors. But why would we need a framework? We could simply use the DOM and make no separation between our data structure and the way the data structure is displayed. This has the advantage of native support in every browser, with highly optimized and well-tested native implementations of the whole API. Still, this leaves many problems unsolved: We need a way of updating the DOM in a rich text style, for which only the `contenteditable` attribute is available, which does not allow for much customization and behaves differently across browsers. Furthermore, a transaction processing system would be helpful, since we want to support typical features such as undo/redo for usability. Finally, we want to construct and enforce a kind of document schema so we can parse external web content (e.g. content pasted from a website) and to avoid invalid documents which may behave inconsistently when distributed to learners.

`ProseMirror` can address each of the limitations of the native DOM and `contenteditable` approach. It is separated into several modules, most importantly model (`prosemirror-model`), state (`prosemirror-state`), and view (`prosemirror-view`). The model module provides a data structure for representing structured documents based on a schema. The state module extends that data structure with a structure to represent the current document selection, and with a transaction processing system to manipulate the document. Finally, the `ProseMirror` view module offers logic to display an editor in the browser and keep it synchronized with the state. We can see that the `ProseMirror` model and state modules mostly fit onto our model in the MVVM sense (we keep all application logic in the model, so the state part fits there, as well), while the `ProseMirror` view module matches the MVVM view. Using the library makes many implementation tasks easier: The data structure offered is fairly sophisticated, as we can represent documents as trees composed of a fixed set of nodes. However, the nodes available and the permitted nesting of nodes in the tree is not hard-coded, but configured through a user-defined schema. Through the state module, we can also manipulate that schema-based document transactionally (e.g. on pressing "Enter", we may split a paragraph into two at the selection caret's position).

For `WebWriter`, we make use of each part of `ProseMirror`. On the Model layer, in the `DocumentStore`, the current editor state including the document's model is stored, using `prosemirror-model` and `prosemirror-state`. Also on that layer, the

base schema for all WebWriter documents is extensively defined using ProseMirror libraries. On the View layer, prosemirror-view is wrapped in a custom <pm-editor> web component which renders the view into an <iframe> and exposes ProseMirror's view API to the rest of the application.

There are a few more external libraries and technologies used in development, but they are used in limited ways for very specific implementation tasks (i.e. the library TEMML for conversion between LaTeX math and MathML) and not worth discussing in detail.

5.4.3 Miscellaneous Components

Cloud WebWriter is a file-based authoring tool, meaning that, comparable to word processing tools such as Microsoft Word, output is stored as local files (HTML). While this is positive for data ownership, privacy, and robustness in face of short-term disruptions in internet service and long-term disruptions in the availability of software, there are several weaknesses. Most importantly, sharing documents is left to the educators, which have to manage file access in another system such as Moodle [Dou02] and take care of synchronizing changes. This is both true in terms of sharing documents with students in class, and sharing documents as templates with each other.

To address this limitation, we introduced an optional cloud component that works similar to solutions in word processing tools: Users may connect a WebWriter Cloud account. If they do, they can now save and load documents from the WebWriter Cloud. Documents in the cloud may be private (only accessible to the author), community-accessible (all registered users may access them), or public (anyone may access them). Also, there is a sharing button for cloud-stored documents which produces a link and QR code to share with students to access a document as a website. The work-in-progress AI integration also makes use of the cloud component, where the cloud is responsible for forwarding user queries to the LLM API (for the prototype currently an Azure [Mic08] instance is provided via RWTH Aachen University) and also rate limiting users if necessary.

Technologically, the cloud service has to store basic account data, manage authentication and authorization, store documents including metadata, and finally offer simple CRUD (create, read, update, and delete) functionality for documents. Due to these limited requirements, an off-the-shelf solution called PocketBase [Geo23] was chosen. Advertised as an "open-source backend in one file", PocketBase already includes a database (SQLite [Hip00]), user management and the concept of collections, for which a RESTful API is automatically generated. If necessary, PocketBase can be extended via JS or Go. PocketBase may not be viable for larger projects, in which more complex architectures such as a full Node Express [Hol10] app with a seperate Postgres [Pos96] database may be chosen. For our use case however, PocketBase is sufficient.

Website The website[6] is another optional offering which is the location of several features of the system's interface (Section 5.3). It is used to inform about the project,

[6] https://webwriter.app

post announcements, present documentation to developers, and most importantly offer access to the WebWriter app. It is the point of entry for both developers and educators, and thus includes elements from both the development and the authoring interface concept.

As the website's requirements are fairly simple, the modern approach of a static web framework was chosen, where content and presentational logic are separated. Among the contenders in this space, the framework Astro [SM21] was chosen, as it is intended for mixing static and interactive content (the "Islands Architecture"). This is well-suited for WebWriter's website, as it allows us to include widgets from the app on the site, for example in the *Packages Gallery*, a regularly updated overview of all WebWriter packages available with an interactive preview. For other parts such as the *Documentation* and the *News* pages, the Markdown-based authoring of content makes it easy to update the website. The documentation can then be available in Markdown in the project's repository, meaning it is version-controlled and also accessible via GitHub [Pre+08a], introducing a redundancy for the website. This is a good practical measure to assure the availability of the documentation long term. Alternatives to Astro were considered, but some were too heavy and not developer-focused (WordPress [LM03]), not component-based and thus a poor match for WebWriter (Hugo [Fra13], Jekyll [Pre+08b]), or not content-driven, making documentation and news updates tedious (NextJS [Rau+16], Nuxt [CCP16]).

Developer Tools As introduced in the previous section (Section 5.3), we offer tools for developers to assist during development. This is a standard practice in frontend development, and there is a rich ecosystem of tools, from language servers providing syntax highlighting and parsing, across all kinds of command line scripts (i.e. to scaffold projects), to the newest AI integrations (i.e. GitHub Copilot [Mic21]). For WebWriter, we mainly want to help developers automate common development tasks.

During our incremental development process, three tasks were identified which could benefit from automation, all of which were bundled in a single command line tool called @webwriter/build [Sal25b]. Each task described can be executed on a machine with an NPM client installed (standard for frontend development) simply by running npx @webwriter/build <task>. The first task is building the package (build). Building in our context means bundling a set of different source files (e.g. TS modules with a chain of imports and external dependencies) into a single JS output file which contains all dependencies. Specifically, developers need to create bundles for each of their widgets so WebWriter can import them once published. Our build tool reads the package's package.json, extracting widget source files and applying the build tool esbuild to bundle. The second task is localization (localize): Packages can be available in multiple languages, which includes the widgets, example snippets, but also package metadata such as the description. The localization tool offers multiple localization steps: Through static analysis, interchange files in the XLIFF format are created which can hold translations for different languages. Next, developers can manually enter translations or apply automatic translation tools (if a DeepL [Kut17] API key is provided, that API is used for translation). Finally, changes in the interchange files are mirrored back into the widgets, snippets, and package metadata.

The third task is documentation (`document`): For each package, it is helpful to have an informative `README` file as an entry point for other developers. This is especially important for usage outside of `WebWriter`, where instructions need to be provided on how to import the widget's script and use the web component. For this, we again use static analysis via the tool `@custom-elements-manifest/analyzer` [Ope21]. The analyzer reads all widget code and converts `jsdoc` strings on methods and properties of the widget class into a human-readable documentation. In addition, the package's structure is read to provide general information, and generic installation instructions with different methods are included so other developers know how to reuse the component.

5.4.4 Implementation Challenges

While most parts of `WebWriter`'s implementation are similar to other web apps and come with the same issues, there are several aspects which are unique to the project. Likely, they are also generalizable to explorable authoring systems in general, and for that reason are worth discussing.

Desktop vs. web apps The early prototypes of `WebWriter` in the Alpha phase were not implemented as web apps, but as web-based, native desktop apps. For this, the framework `Tauri` [TN20] was used, which allows developers to implement a desktop app using a web technology core, which is then wrapped in a desktop app. This allowed us to leverage web development expertise and use the fairly advanced document presentation and multimedia capabilities of the web. The desktop-web mix approach with `Tauri` was chosen since on one hand, for a tool such as `WebWriter` which authors web-based documents, something web-based is a technical necessity. On the other hand, the user experience should closely resemble known software, and browsers at the time did not allow proper file system access, making a file-based authoring tool fairly awkward in practice (e.g. users couldn't save progress, only download documents again). For `WebWriter`, implementation in `Tauri` was at the time a good solution to balance both requirements: `Tauri` offers APIs to access the file system, but the application logic is fully implemented as a web app with no restrictions on technologies used (e.g. Angular, React, Vue, Lit). Compare to other solutions in the space such as `Electron` [Ope13], `Tauri` also has a specific performance advantage since it uses the platform's native web view: A `Tauri` app would use the `Chromium`-based `Edge` [Mic20] web view on Windows and the Safari web view on Mac, for example. This means that the app's size and also memory footprint is fairly small, in stark contrast to `Electron` apps which bundle a full `Chromium` runtime consuming disk and memory space for each app.

With the introduction of the new File System Access API in browsers, that major point of difference between desktop and browser became moot, as web apps could now request permission to access limited parts of the user's file system. Together with other desktop-like interfaces such as the Media Capture and Streams API, most limitations which would necessitate a desktop app are no longer present in modern browsers. As major obstacles were removed and pure web apps also came with major

advantages (users need not install anything, no need to maintain the native app across different platforms), `WebWriter` transitioned to a web app in the beta phase.

While the result has the aforementioned advantages, the transition proved challenging, mainly in the package system. Previously, packages were downloaded directly from NPM using a client and stored on the user's hard drive. A code bundle was then generated from all installed packages and inserted into the page. On the web, this workflow would be hard to implement, as NPM clients are not suited to running in browser environments. To transition to the web, the package system was changed to be more "web-native", as packages are now fetched from a content delivery network (CDN) mirroring NPM and simply included as external scripts. Bundling does not occur each time a package is installed, but only when saving a document (see "data format and dependencies" below). This has the additional benefit of being far more performant than the previous, file-system-heavy process, but also relies more on external, online services.

Relying on an External Service Transitioning `WebWriter` to a web app means that the package system now uses two external services, NPM and `JSDelivr` [Aku12]. NPM is the registry which handles the publishing, versioning and hosting of packages. However, NPM does not offer direct access to the resources of each package, but only management through NPM clients which download the whole package, resolving dependencies. For front end packages such as those used in `WebWriter`, this is inconvenient as we can easily include a distributable bundle script in the package, but have no way of importing that script into the document automatically. Here, `JSDelivr` helps by mirroring all public NPM packages and providing access to all resources through their CDN.

External online services are always points of failure, since the connection or the service itself may break down. While NPM and `JSDelivr` are large, well-known and established services, there is no guarantee of availability, especially since both services are free and public. We experienced that problem first-hand during the authoring study, when NPM unexpectedly broke down for hours (after years of reliable service). After that experience, we implemented several redundancies. A redundancy for NPM was fairly easy to achieve since the only interface used is the search endpoint for live lookup of `WebWriter` packages – a fallback was added which is simply a list of `WebWriter` packages hosted on the `WebWriter` website and automatically updated periodically. For `JSDelivr`, there are several services which perform the same task of mirroring NPM packages, and only differ in the URL scheme used to access these packages. In the new implementation, `JSDelivr` is tried first, and other services from a list are tried after in case of failure. The likelihood of all services failing at the same time seems too low to consider seriously.

Overall, the use of these external services means that the authoring tool is no longer fully functional offline (but authored documents still are). However, our web-native implementation using a CDN also means we can tap into browser mechanisms such as the browser's cache, meaning that intermittent loss of connection does not affect the app: A user on a train losing connection in a tunnel could still edit the

document and widgets would keep functioning, since the browser falls back on the resources cached earlier.

Dynamic Document Schemata In most rich text editing systems which feature a schema, it suffices that the app developer defines a schema once. A schema specifies constraints on a document's structure, such as the types of content which are valid. For example, a simple schema for Markdown-like documents may include definitions for headings, paragraphs, lists, and so on. In explorable authoring systems, the situation is more complicated. An explorable document may include any number of external components (widgets). We could represent them in the schema with a generic `widget` node, but this prohibits us from defining custom rules for the editing behavior of each widget. For example, a slide widget representing a single slide should only be allowed inside a slideshow widget.

In `WebWriter`, the solution chosen is to dynamically extend the schema depending on which packages are installed. First of all, there is a base schema including all HTML elements in `WebWriter`. On installing a package, for each widget defined in that package, a node definition for that widget is added to a new and extended schema. The current document is then parsed with that extended schema, thus completing the update. The parsing step is mostly invisible to the user, unless they uninstall a package with a widget which is included in the document, yielding a warning in place of the widget.

Data Format and Dependencies Thinking about saving and loading documents, it is clear that HTML is the natural serialization format - `WebWriter` allows authors create any web document. We could choose a custom, more limited JSON format, as some applications opt to, but this would not be interoperable at all. Choosing HTML has an interesting benefit: Learners can open, save and load the HTML file locally in their browser, or access the document as a website when it is hosted on a web server – choosing HTML is key for decoupling the authoring tool and the content. Using `ProseMirror`, the serialization implementation is fairly simple: `ProseMirror` can construct parsers and serializers based on the document schema. However, settling on HTML, there are some unsolved challenges. A `WebWriter` document is supposed to work offline as much as possible, but documents may include two types of externals: Assets such as images, audio/video, and widgets, which depend on package scripts that define the web component to function. For both types of externals, we need to perform a kind of embedding where the external resource is represented as a string in the HTML, instead of a hyperlink.

Assets are very common in websites and usually appear as hyperlinked resources in the corresponding multimedia elements (img/picture, audio, video). With browser APIs, this is fairly easy to convert: On saving, `WebWriter` finds all multimedia elements with external resources (URLs starting with `http://` or `https://`), fetches each asset and converts it to a data URL. Data URLs are a standardized string representation of data, where the media type and encoding is specified (`data:[<media-type>][;base64],<data>`), and the data, often in Base64 encoding,

is directly included. For example, an image `https://example.com/duck.png` may become `...`. The method works for any media type, with some caveats. First of all, compared to binary data formats, Base64 encoding is about 37% larger, however, this is unavoidable as HTML itself is not a binary format. Second, there are undisclosed security and performance limits for data URLs, depending on the browser. Outdated advice suggests that a maximum of 2MB works across all browsers, but our own tests show that larger files also work. For images and audio, the performance and file size impact is acceptable – videos on the other hand are more problematic. For this reason, larger videos are not embedded by default.

For widgets, there are two distinct parts to consider: The widget definition is a set of scripts which registers a custom element, and optionally applies custom CSS styles. Widget instances may then appear any number of times across the document. Widget instances are serialized in standard DOM/HTML fashion (attributes and content, more on that in the next chapter). We could embed widget definitions in a similar way to assets, but using the `script` element instead of data URLs. Definition scripts from the package would be fetched and the content inserted into script elements which are included in the document. However, this is suboptimal in terms of script size, as each package may share dependencies which are then included multiple times. As the packages are in the Node package format, dependencies are declared and could be resolved deterministically. So as an improved solution, all widget code is combined into a single bundle script using the WebAssembly version of the bundler `esbuild`. Due to the good performance of `esbuild`, this introduces only a small delay in saving, but has a large impact on script and thus file size. As the bundling obfuscates which packages and versions were used originally, an import map element is inserted as a type of manifest to show which package version is used for each widget. This helps authoring tools such as `WebWriter` to read outdated documents and update them where only minor or patch changes occurred, or to prompt the user for a decision when major changes occurred.

Isolation and Security From a security point of view, `WebWriter` has several vectors of attack which can be mitigated, but not avoided or disabled entirely. In general, attacks could either target authors (the non-technical end users) or, through the authored documents, learners. Apart from usual risks of web applications such as supply chain attacks on JS dependencies, hijacking the hosted app or website, and so on, there are at least two risks unique to explorable authoring systems. First, the package system allows users to execute arbitrary code in their documents, as installing a package causes each widget's definition script to be executed. This means that an attacker could include malicious JS in a fake, "trojan" package. Also, the package system's external services (NPM and `JSDelivr`) could be compromised (targeting the widget's supply chain). Second, users can paste arbitrary HTML snippets or import any website, which is another way of introducing malicious code.

As mitigations, there are several measures in place. Most importantly, the edited document is sandboxed and isolated from the the rest of the application through an `iframe`. This is a standard practice also used by mail clients to display rich emails, and means that any script in the interactive document cannot manipulate the editor's

interface. As `iframes` are standardized parts of HTML, security is kept up-to-date and maintained by browser vendors. This sandboxing helps both with malicious packages and documents, as all external scripts are executed inside the `iframe`, never outside. For malicious packages, as a further measure, a whitelist was implemented: Only packages from whitelisted NPM organizations are shown (by default, only the `@webwriter` organization). While protection against malicious scripts is already sufficient through the sandboxing measure, the whitelist also helps with possible spam and advertisements pushed through the package system. Balancing the need for security with long-term availability and user control however, there is a user setting to also show non-whitelisted (unofficial) packages.

These measures are helpful, but cannot preclude abuse completely. As JS is Turing complete, any program, including a harmful one, can be expressed. For example, the supply chain of a whitelisted package could still be attacked and data stolen or inappropriate content shown. To avoid this, we could limit package implementations to a subset of JS or a domain-specific language instead, but that would drastically impact development expressiveness and usability. We could also go further than the document sandboxing performed now and isolate each component in an `iframe`. That solution is used by the Python-to-web library Streamlit, for example. However, the performance impact would quickly become prohibitive (each iframe is its own full DOM document) and components accessing the surrounding document can be desirable (for example, a "table of contents" component would need to access the whole page to identify sections). For these reasons, we feel that for an explorable authoring system, an unlimited, but sandboxed document is the best approach. This allows `WebWriter` to benefit from the built-in and constantly updated security measures such as CORS, permissions, and so on, but does not limit interactivity in any meaningful way.

5.5 Summary: Prototype

In this chapter, we introduced `WebWriter`, an explorable authoring system. First (Section 5.1), we chose a development process, opting for a multi-phase, iterative approach modeled on the RUP with the Alpha, Beta, and Release phases. Next (Section 5.2), we presented a model called the explorable authoring requirements (EAR). Based on the current literature and a requirements study with educators, it considers functionality (covering expressiveness) and usability for the two groups of educators and EdTech developers. Then (Section 5.3), we derived a concept out of the EAR, divided into an authoring interface (app bar, palette, document, toolbox, metadata, preview, and settings) and a developer interface (documentation, local packages, and optional tooling). Finally (Section 5.4), we looked at the implementation. We outlined the architecture which follows the MVVM and SPA patterns, the technology stack (TS with `Lit` web components, `MobX`, and `ProseMirror`), and other components besides the main app (cloud, website, and developer tools). We also discussed project-specific challenges (desktop vs. web apps, external services, data format and dependencies, and security).

RESEARCH SUMMARY

CT: In the context of a technical solution for web-based, interactive resources to support active learning, aimed at post-primary STEM learners:

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

ME: Create a prototype with participatory design. Evaluate it in experiments with authoring workshops and development projects, for each proving remixability theoretically and measuring usability (UEQ+) & expressiveness (QCA).

PR: *WebWriter*, a single page web app implemented with web technologies in the MVVM pattern, offering both an authoring and a development interface.

- **RQ_1**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
  - ◇ **HY_1**: Interactive elements are remixable if they are findable, accessible, interoperable, and reusable (FAIR), which is possible by applying the component and package patterns and adding a domain-specific API.
- **RQ_2**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?
  - ◇ **HY_2**: Authoring should support the remix workflow for educators by extending rich text editing with a new paradigm to cover morphological changes to interactive content.

CT=Context, RQ=Research Question, ME=Method, PR=Prototype, HY=Hypothesis, EV=Evaluation

Chapter 6 Widgets

In this chapter, we address the first research question regarding a software design pattern for EdTech developers to create FAIR interactive elements.

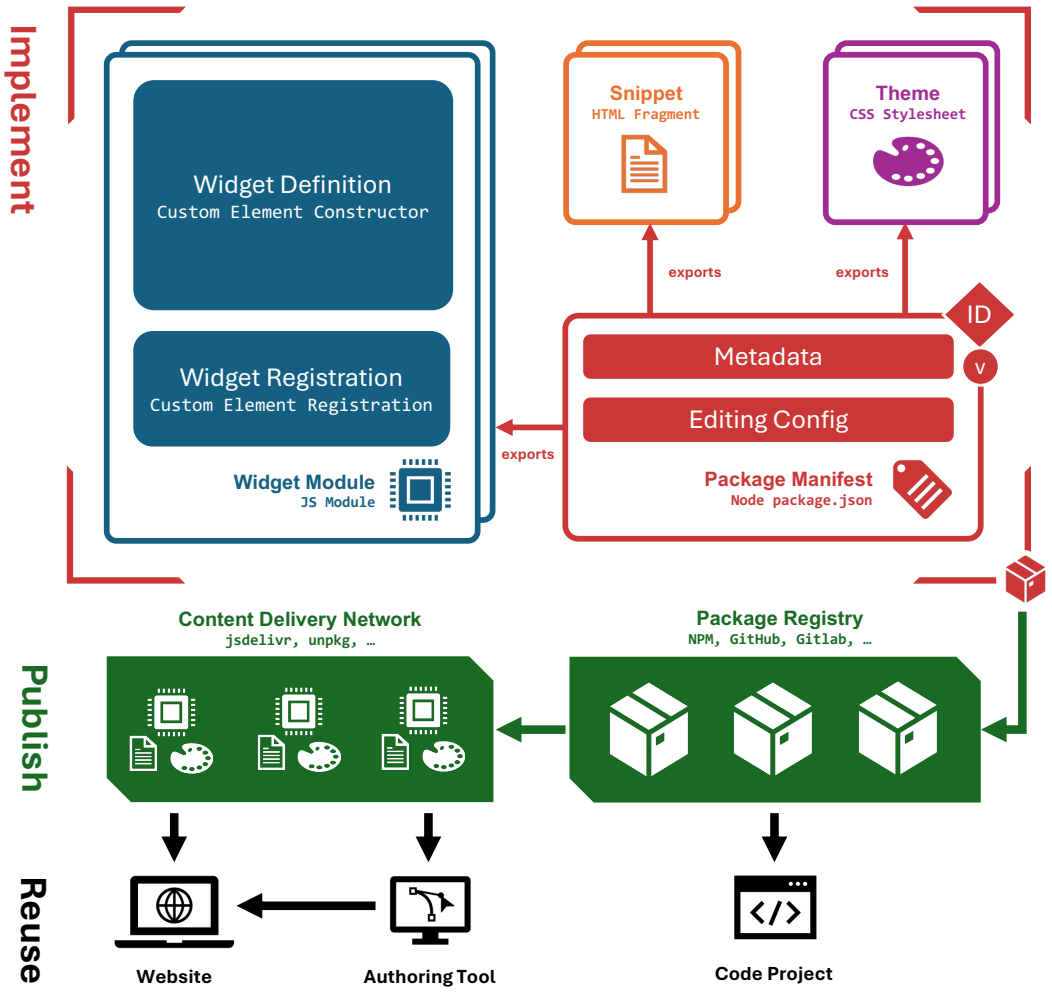


Figure 6.1: The widget pattern

The chapter is divided into three sections: First (Section 6.1), we formulate the design pattern. Next (Section 6.2), we show examples. Last (Section 6.3), we present the evaluation of the concept with developers including study setup, data analysis, results, and a discussion.

6.1 Widget Pattern

We want to find a new design pattern which makes developing FAIR interactive elements usable without sacrificing expressiveness. We first formulate our idea in theory, then show how it is implemented in practice on the web.

6.1.1 Widgets in Theory

To achieve FAIRness for interactive elements in a usable and expressive way (RQ1, see Section 1.4), we hypothesized that a combination of the package pattern and the component pattern could cover all FAIR principles (see Section 3.1).

As a first formulation of the concept, we can list the principles that an implementation of our pattern would have to fulfill. Combining packages and components is fairly straightforward: Recall that a package is a uniquely identified and versioned set of files, with a manifest included – that may achieve findability, accessibility, and reusability. If we now include a component inside the package, we may also achieve interoperability. That is the most basic formulation of the idea: Put components inside packages. We call this the widget pattern.

> **Widgets**: The design pattern of widgets wraps components in packages to achieve FAIRness, enabling flexible reuse for code projects, through authoring tools, or in (web) documents.

To understand the design patterns in more detail, there are a few principles any implementation must fulfill to fit the pattern (see Table 6.1). We formulate these in a similar way to the FAIR4RS principles introduced earlier (Table 3.1), which allows us to abstract away implementation details that may arise from specific technologies.

#	Widget Principle	FAIRness
1	There must be a *collection* of files with no restrictions on the type of file.	
2	The collection must include a *manifest* with a name and a version.	F1
3	The manifest may include any additional metadata.	F2, F3, F4
4	The manifest may declare dependencies on other collections.	I2, R2
5	The collection must be published on a *registry* which provides access and enforces that each name-version pair uniquely identifies that collection.	A1, A2
6	The manifest may declare zero or more *members* (widgets, snippets, or themes) to be exported, mapping a name to a declaration path.	F.1.1
7	A *widget module*, in a domain-standard programming language, defines the widget's behavior and interface as an extension of the abstractions provided by the programming environment.	I1
8	A *snippet* contains a document fragment in a domain-standard markup language.	I1
9	A *theme* contains an overall style to apply to a document in a domain-standard style sheet language.	I1
10	The manifest may contain an *editing config*, which describes how members should be treated within authoring tools.	R3

Table 6.1: Defining principles of the widget pattern, divided into package principles (red, 1-5) and domain-specific component principles (blue, 6-10)

The first five principles are derive directly from the package pattern (see Section 3.1). (1) and (3) allow us to include any file and any metadata we may need for our project, avoiding any limits in expressiveness. Yet data and metadata are separated according to FAIR, so machines can easily provide search capabilities, for example. (4) is helpful for interoperability and reusability, as dependencies are a type of qualified reference to other software. Finally, (5) is key to accessibility, as the registry is the mechanism which can offer a uniform access to widgets.

The other five principles weave in the component concept and focus on the domain of interactive documents. By Szyperski's definition (see Section 3.1), components are units of composition with explicit context dependencies and contractually specified interfaces, which are deployed independently. With interactive documents, it is worth considering what might constitute units of composition. The most central unit is certainly the interactive element, the eponymous widget. Widgets are only required to define their own behavior and interface, and register their name (7). The principle requires defining the behavior and interface with the abstractions available in the domain's programming environment, which increases interoperability between the environment and the component. Snippets and themes are two other types of components specific to interactive documents: While elements are the atomic and indivisible units of composition, larger units of composition may also be useful, which we call snippets (8). Furthermore, as many interactive document systems practice a separation of concerns, we may also introduce themes as another unit of composition (9). Finally, there is the consideration that there are at least two types of usage of an interactive document: End users employ a reader (also called player, browser, etc.), and authors employ an authoring tool. For authoring tools, there are likely specifics on how the interactive element should be treated (10).

6.1.2 Widgets on the Web

The previous subsection gave us a theoretical lens on the widget pattern. In practice, in the domain of web-based EdTech, we can use existing technology to adopt the pattern fairly easily, only taking three steps and a few minutes to get started:

1. Implement a widget module in JS (e.g. `my-widget.js`), with a custom element constructor for the widget behavior and interface, and a call to the custom element registry to register the widget name (see Listing 6.1).

2. Add a `package.json` with a name and version unique to a target registry (e.g. NPM), and an `exports` key pointing at the widget module (e.g. `"./widgets/my-widget.*":"my-widget.*"`) (see Listing 6.2).

3. Publish the package on the target registry (see Listing 6.3).

```
1 import {html, css} from "lit"
2 import {LitElementWw} from "@webwriter/lit"
3 import {customElement} from "lit/decorators.js"
4
5 @customElement("textarea-widget")
```

```
export default class TextareaWidget extends LitElementWw {
  render() {
    return html‘<textarea></textarea>‘
  }
}
```

Listing 6.1: Minimal definition of a widget using the Lit framework and the LitElementWw base class from WebWriter’s developer interface

```
{
  "name": "@webwriter/textarea",
  "version": "0.1.0",
  "keywords": ["webwriter-widget"],
  "exports": {
    "./widgets/textarea-widget.*": "./dist/widgets/textarea-widget.*"
  }
}
```

Listing 6.2: Package manifest for NPM in package.json format exporting a JS widget module

```
cd ./projects/@webwriter/map && npm publish
```

Listing 6.3: Publishing a widget through the NPM command line tool

In these steps, we can see two technologies, namely web components and Node packages (see Section 3.1) – the use of which enforces a basic degree of FAIRness.

Node packages can cover the first five principles (see Table 6.1), as they permit files of any type (1), require a manifest `package.json` with a name and version (2), which can include any metadata through key-value pairs (3), and may declare dependencies (4). Package registries such as NPM complete the system by providing access and enforcing uniqueness (5).

Let us look at the FAIRness of Node packages by exploring the `package.json` format: Findability is guaranteed as there is a globally unique and persistent identifier (F1), the package `name`, and there a `version` (F1.2) in the form of semantic versioning. The NPM registry enforces the uniqueness of the package name and version as it prevents publishing two packages with the same name and version. This guarantees that `mypkg@1.2.3` will always reference exactly the same artifact. Rich metadata can be attached to the package (F2), and the association is clear as metadata is always co-located with the `name` and `version` in the `package.json` (F3). As all metadata is contained in the `package.json` file, it is trivially searchable and indexable for package registries (F4). Accessibility is given through package registries, which offer a HTTP REST API to access packages which can be easily be used through a multitude of different clients. HTTP is clearly open, free, and universally implementable (A1.1) and enables authentication and authorization (A1.2). The same goes for the specific REST API for package managers and registries, which is well-documented and implemented by many different tools (besides the `npm` CLI tool, there are also popular alternatives such as `yarn`, `pnpm`, and `bun`, and even platforms such as `GitHub` or `GitLab` doubling as alternative package registries). The metadata is easily separated from the package itself and usually crawled by search engines and web archives,

meaning it will likely remain available long-term (A2). Finally, reusability is encouraged (though not enforced like findability) through extra metadata directly supported by package registries. This includes metadata for publishing such as `keywords` and `description` (R1), and a `license` (R2). Detailed provenance can be establish through several measures, such as specifying an `author`. Externally, the registry also associates the package with a registered user (R3). Qualified references to other software are also possible through the `dependencies`/`devDependencies` fields (FAIR4RS R2).

We can conclude that Node packages are in fact highly promising if viewed through the lens of FAIRness. Findability is mostly enforced automatically by the registry, guaranteeing certain properties such as a unique ID and versioning. Similarly, accessibility is provided by publishing on NPM. A relevant criticism here is that this ties developers to NPM, and if the platform were to become unavailable, this would be problematic. However, due to the modular nature of NPM, there is no single point of failure: Packages could be published in other registries, such as on `GitHub` or `GitLab`, or self-hosted on solutions such as `Verdaccio` [Pic17]. Similarly, besides the official and open source `npm` package manager, there are many alternatives. Finally, processing the metadata in `package.json` to interoperate with custom tools is also attainable, since the format is simple and well-documented.

The last five principles (see Table 6.1) can be covered by several web technologies, mainly web components in JS modules as a way of defining widgets (7), HTML files for snippets (8), CSS files for themes (9), and finally by extending the standard export and metadata capabilities of Node packages (5, 10).

Web components (see Section 3.1) may be a viable and modern way to realize interactive elements as components on the web platform. Compared to previous solutions such as React, they work independently of frameworks and offer a stronger integration with existing web technologies. They provide interoperability and there are strong incentives for major actors in the software space (FAANG companies developing browsers such as Google, Mozilla, Apple, and Microsoft) to maintain that interoperability in the long term. From the perspective of EdTech, which is only a very small subset of the software world, making use of the web platform also means making use of the far more abundant resources of the larger software world, instead of implementing parallel, but inferior and not interoperable solutions.

Viewed in terms of FAIRness, the interoperability web components provide is impressive: As there are interfaces with all major web technologies (HTML, CSS, JS, DOM), web components benefit from innovations in each technology. This interoperability is also far more well-defined and sustainable than solutions on the level of libraries, as web browsers backed by major organizations are committed to implementing the web standards and remaining compatible in the long term. As such, reading/writing/exchanging data with domain-relevant community standards (I1) is true with web components, and bound to remain true. Qualified references are also interesting due to the hypermedia principle the web implements (FAIR4RS I2). There are many ways to reference other objects, such as IDs in the local document, or URLs for cross-document and cross-media references. For example, a custom elements for interactive videos can easily reference an external video, e.g. on YouTube, with a URL. In theoretical terms, web components provide a component architecture [SGM02] for

the web which bridges the islands of frontend frameworks: Each framework mentioned both offers a way to wrap their type of component as a web component and to include web components as a unit of composition.

For snippets and themes, the web provides HTML itself and CSS, which is highly interoperable: CSS stylesheets can be included on any page or rules therein partially reused, and HTML fragments can be inserted into pages or broken up into parts.

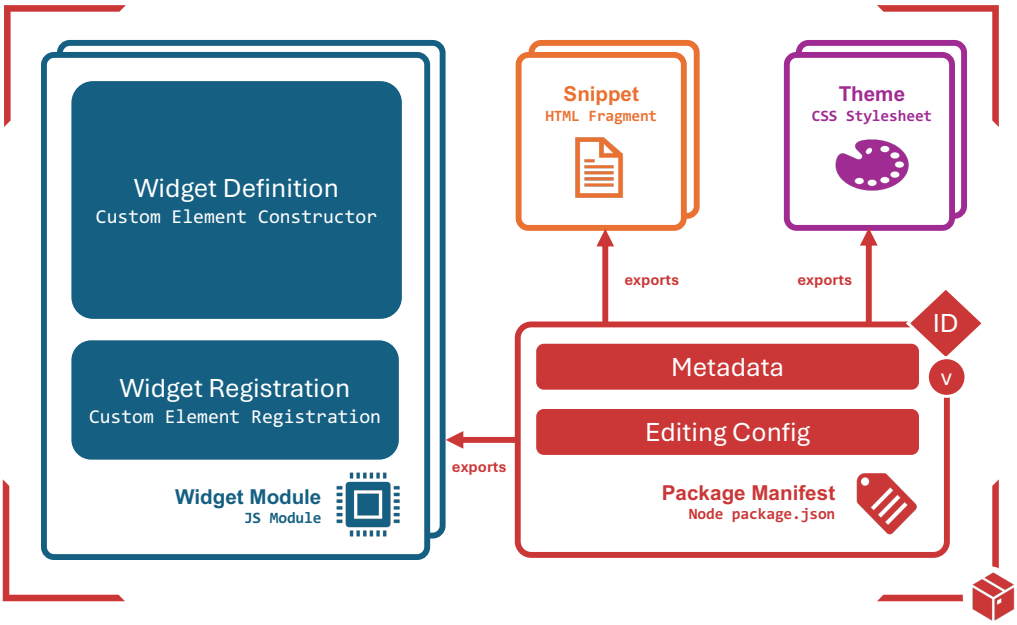


Figure 6.2: Implementing widgets, with our example application of the pattern to the web ecosystem with Web Components in JS and Node Packages

6.1.3 Reusing Widgets

After a widget is implemented, it needs to be published. Publishing happens through a single workflow, while reuse can take different forms (compare Figure 6.3).

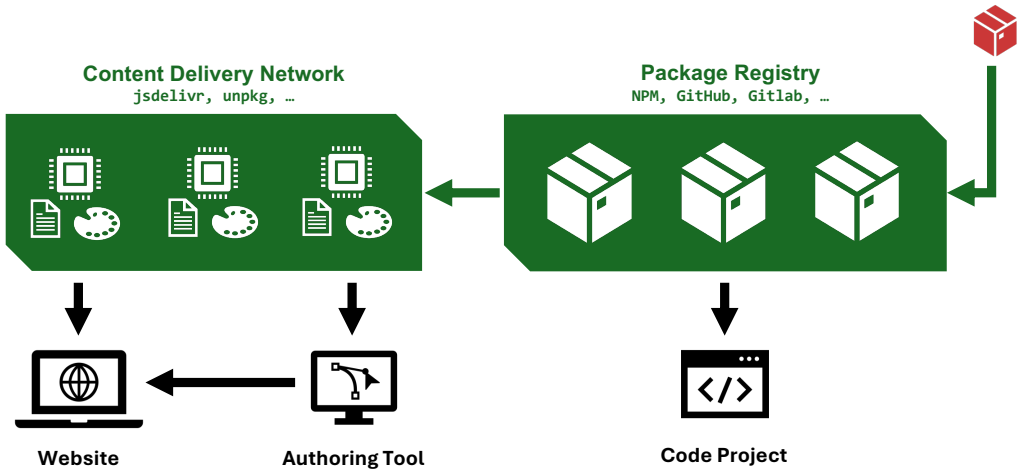


Figure 6.3: Publishing and reusing widgets

Publishing Widgets Widgets may be published on a registry. As outlined in our widget principles, a registry is responsible for enforcing the uniqueness of each name-version pair. In practice, the most significant state-of-the-art implementation is NPM. On NPM, anyone may publish and access packages free of charge, as of 2025. For publishing, a developer would need to create a free account. They would then use a package manager (e.g. NPM, `yarn` [Nis16], or `bun` [Sum21]), typically through a command line interface, to trigger a publishing operation (see Listing 6.3). For a typical developer, the workflow is very simple and can be completed in a few minutes.

When publishing is complete, a landing page with information for developers instantly becomes available (see Figure 6.4). That page also show's the package's readme, providing an easy reference for other developers when reusing the package.

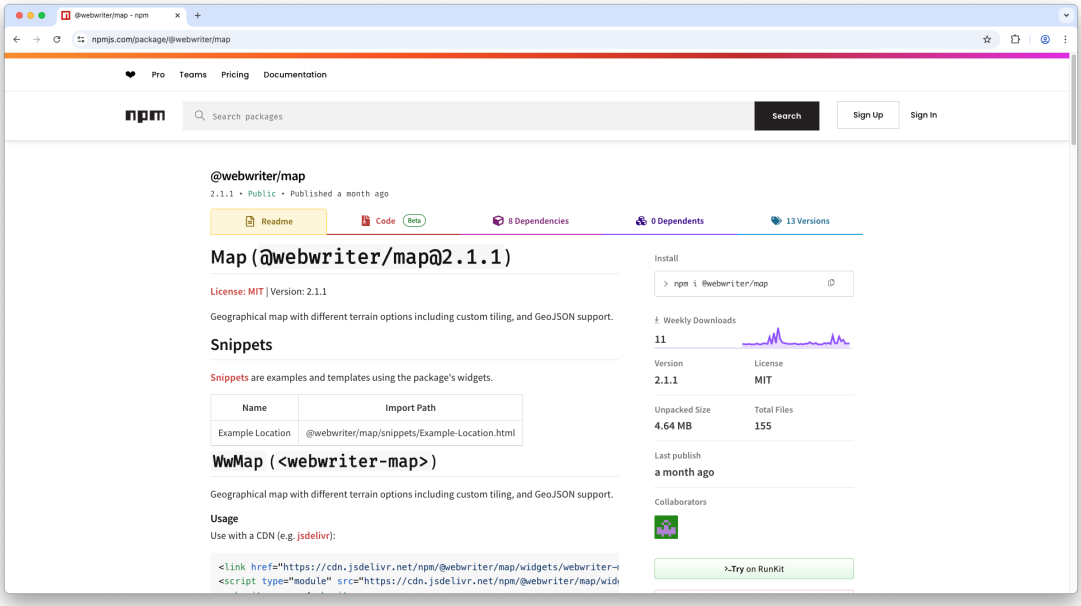


Figure 6.4: A published widget (@webwriter/map) on NPM, including a readme (middle) and statistics (right)

Reuse in Code Projects For other code projects, reuse is straightforward, as this is the standard use case for registries such as NPM. Another developer may want to reuse a widget in their project – for example, imagine someone developing a geography learning app and wanting to include an interactive map. As the widgets is available in package form on NPM, that other developer can install a version of their choice through a package manager (see Listing 6.4). Note that the package manager also has capabilities for specifying version ranges, so non-breaking changes such as security patches are applied automatically. Overall, the package manager can resolve all dependencies of all packages, so the widget can be installed and used in seconds. After the package is installed, it is available locally and can be imported in that developer's code (see Listing 6.5). Additionally, snippets and themes can be reused in similar ways. Note that typically, the widget's source code, which may differ from

the published artifact, is also publicly available, e.g. as an open source repository on GitHub, opening the way for source-level reuse such as forking the whole project.

```
npm install @webwriter/map
```

Listing 6.4: Installing a widget through the NPM command line tool

```
import "@webwriter/map/widgets/webwriter-map"
// The <webwriter-map> element is now globally registered
...
```

Listing 6.5: Importing a widget in another JS module

Reuse on Websites A more accessible way of reuse than installing the package can be to include the widget on a website. If a CDN service such as `jsdelivr` [Aku12] is used, this can be accomplished entirely declaratively by adding the widget module as an external script and then using the thus registered web component on the website (see Listing 6.6). Note that this type of reuse requires no additional steps by the original developer, as assets of public packages on NPM are automatically mirrored and made accessible on several CDNs.

```
<link href="https://cdn.jsdelivr.net/npm/@webwriter/map/widgets/webwriter-map.
    css" rel="stylesheet">
<script type="module" src="https://cdn.jsdelivr.net/npm/@webwriter/map/widgets/
    webwriter-map.js"></script>
<webwriter-map></webwriter-map>
```

Listing 6.6: Including widgets on websites through a CDN service (e.g. jsdelivr)

As users including widgets this way only need to add a few lines of HTML, reuse via CDNs is approachable even for technically adept non-developers such as web designers or hobbyists creating personal websites and blogs. A necessary disadvantage of this type of reuse however is that a dependency on the CDN is introduced, meaning the resulting web document does not work offline, and the CDN becomes a point of failure with risk of deprecation and possible security and privacy issues.

Reuse through Authoring Tools The most important way of reusing widgets for our project is in a tool-supported fashion through authoring tools such as our prototype `WebWriter` [Sal25b] (Chapter 5). Similar to the CDN reuse on websites explained in the previous paragraphs, an authoring tool helping a user create a web document may allow a user to install a widget, fetching the module in the background and embedding it into the document. Then, the tool may offer affordances to insert the widget into the document. This minimal kind of tooling is not complex, but for many use cases, a more elaborate interface between the widget and the authoring tool is needed.[1]

[1] A full description of the widget authoring interface as implemented in `WebWriter` can be found in the documentation: `https://webwriter.app/docs/`

State and Options: Widgets can make use of the strong integration of web components into the web platform to store state. Since widgets are web components, which are HTML elements, they can use DOM/HTML attributes. For developers, this means making sure that the widget's state is fully represented in attributes, which frameworks such as Lit support by mirroring the state of properties to attributes (see Listing 6.7, where we use the running example of a text area widget). This provides a standard, granular and flexible way of expressing a widget's state. On the authoring tool's side, this interfacing assumption enables many features: The authoring tool may save the DOM as HTML and expect that widgets restore their state when that HTML is loaded (in our example, the textarea would be filled with the saved text again). Furthermore, the authoring tool can observe and track changes in attributes, creating a history which enables undo/redo functionality. Finally, widgets may mark certain attributes as options, for which standard UI elements are generated automatically (e.g. our textarea may have a boolean `readonly` option for which a checkbox UI element is displayed in the authoring tool). We will explore how widgets and authoring tools can interoperate further in the next chapter (Chapter 7).

```
@customElement("textarea-widget")
export default class TextareaWidget extends LitElementWw {

  @property({attribute: true})
  accessor value: string

  render() {
    return html`<textarea @change=${e => this.value = e.target.value}>
      ${this.value}
    </textarea>`
  }
}
```

Listing 6.7: Storing the widget's state in attributes

Edit mode: A fundamental idea to interoperate with authoring tools is to dynamically adapt the widget's behavior and appearance when it is used within an authoring tool. As we will detail in the next chapter, this is a powerful mechanism which can form the basis for a new authoring paradigm (see Chapter 7). As a minimal example, widgets on the web may use the standard `contenteditable` attribute (automatically set by the authoring tool to announce the switch to edit mode) and show or hide certain internal elements depending on the mode (e.g. showing an input field to specify a placeholder value for the textarea when in edit mode).

```
export default class TextareaWidget extends LitElementWw {

  @property({attribute: true})
  accessor value: string

  @property({attribute: true})
  accessor placeholder: string

  static get styles() {
```

```
    return css‘
      :host(:not([contenteditable=true]):not([contenteditable=""])) .author-only {
        display: none;
      }
    ‘
  }

  render() {
    return html‘
    <input class="author-only" @change=${e => this.placeholder = e.target.value}></input>
    <textarea @change=${e => this.value = e.target.value} placeholder=${this.placeholder}>
      ${this.value}
    </textarea>‘
  }
}
```

Listing 6.8: Editing mode on widgets using contenteditable

Editing Behavior: Authoring tools, especially those adopting an approach such as `WebWriter`, may need a few additional hints on how to treat a widget within a rich text editing context. For example, we may want to assign the widget to a certain content category (the HTML Living Standard [WHA25] specifies a few standard ones) so the authoring tool may prevent the widget from being inserted in certain places. For example, a widget may be designed as a block taking up the full width of a document (e.g. a simulation widget needing screen space for displaying a UI), so inserting the widget as an inline element would be nonsensical.

```
{
  // ...
  "editingConfig": {
    // ...
    "./widgets/awesome-widget": {
      // ...
      "group": "flow simulation" // we add a custom group name which other widgets may also use in their group or content
    }
  }
}
```

Listing 6.9: Editing config to assign the content category group "flow" to the widget

Nesting: In some cases, widgets may want to allow content. For example, there might be a slideshow widget which allows one or more slides as content elements, which in turn can have arbitrary content (text, images, other widgets, etc.). Web components offer a standardized way of handling content, which we can use for widgets: The widget's template may include the `<slot>` element, which can be understood as a gap where content will be displayed (see Listing 6.10). For example, if the widget's HTML is `<webwriter-slide><p>hello world</p></webwriter-slide>`, the

paragraph element with the text node `"hello world"` is inserted into the slot when the template is rendered. To realize constraints such as allowing only slides as content inside a top-level slideshow widget, we need to pass a hint to the authoring tool through the editing config (see Listing 6.11).

```
export class MyWidget extends LitElementWw {
  render() {
    return html‘
      Here’s my content:
      <slot></slot>
    ‘
  }
}
```

Listing 6.10: Allowing content inside widgets for nested structures

```
{
  // your widget’s package.json
  "exports": {
    "./widgets/my-widget": "./widgets/my-widget.ts"
  },
  "editingConfig": {
    "./widgets/my-widget": {
      "content": "flow*"
    }
  }
}
```

Listing 6.11: Editing config to specify the types of allowed content

Localization: Similar to the standard `contenteditable` attribute, there is also the `lang` attribute which is available for all HTML elements including web components. The authoring tool may provide affordances to set the language of a document, and that language choice is propagated to all widgets through the `lang` attribute. The widget may then change its appearance (see Listing 6.12) based on the specified locale (e.g. showing a different language label).

```
export default class TextareaWidget extends LitElementWw {

  static LOCALIZED = {
    "What’s your name?": {"de": "Wie heisst du?"}
  }

  // Helper function: Get localized form of ‘str‘ if available, otherwise fall
      back to ‘str‘
  msg = (str: string) => CoolWidget.LOCALIZED[str]?.[this.lang] ?? str

  @property({attribute: true})
  accessor value: string

  render() {
    return html‘
```

```
15      <span>${this.msg("What's your name?")}</span>
16      <textarea @change=${e => this.value = e.target.value}>
17        ${this.value}
18      </textarea>‘
19    }
20  }
```

Listing 6.12: Localizing widgets through the lang attribute (simplified example)

Other Interfaces: There are other, minor interfaces using web platform standards, including interfaces for printing appearance, fullscreen behavior, focus handling, and so on. The ones supported by `WebWriter` are documented for the development interface. This illustrates that the design choice of tying into the web's existing component system of HTML elements yields many interoperability advantages – as widgets are HTML elements, their interface already has a pre-defined shape through the DOM which can be easily specified through frontend frameworks and used by authoring tools through web APIs. We can also see that interoperability makes the concept more sustainable, as new web APIs become available for use in widgets constantly, and existing ones are maintained by browser vendors and standards committees.

6.2 Examples of Widgets

In this section, we present two examples of widgets in-depth to better understand the pattern and how it can be applied in EdTech practice.

Section adapted from own work with collaborating students [KSS24; DSS25]

6.2.1 Neural Network Widget

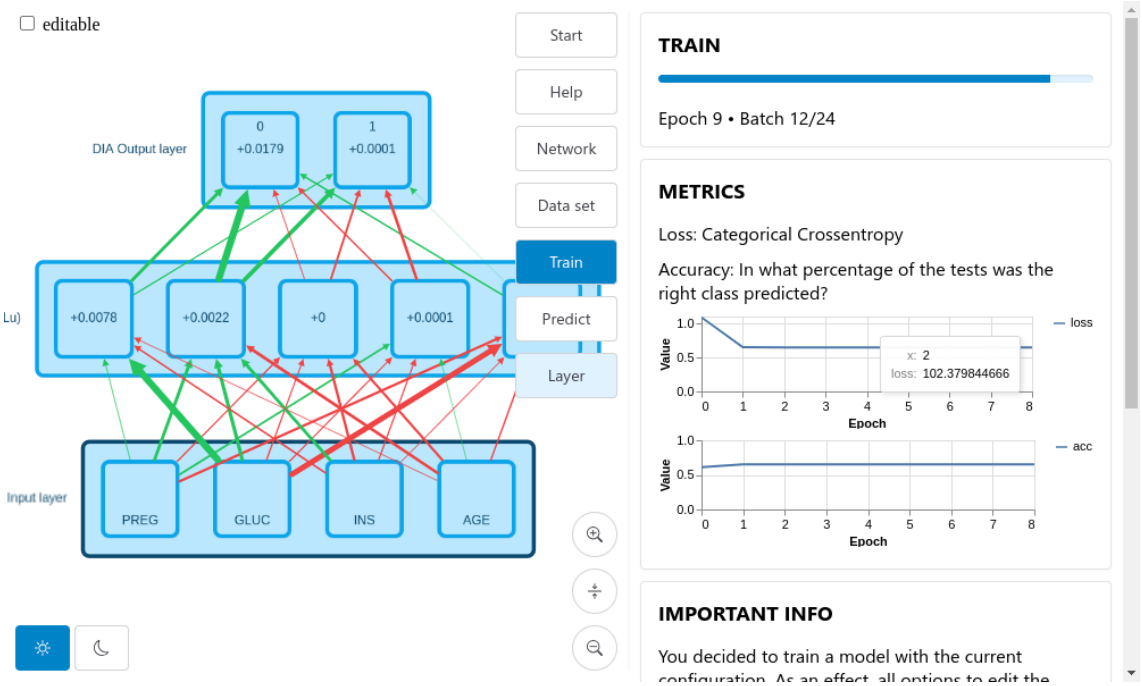


Figure 6.5: The Neural Network Widget while training is in progress

How can today's deep learning and large language models (LLMs) be understood? That it is necessary to understand them seems out of question to us: Through transformer-based models [Vas+23] such as `GPT` [Rad+23], `Gemini` [VP23], and `LLaMA` [Tou+23], LLMs have permeated the digital world, and the topic is discussed across the research community [VP23; Phi23; Rab+23]. Yet, there are fundamental challenges in understanding those models completely [Ted+20]: They are brittle (can be derailed by small changes in input data), threaten privacy (since they require massive amounts of data), biased (biases in training data produce similarly biased models) and they are opaque (it is difficult to reason why they behave the way they do). Understanding these attributes of LLMs which are caused by their data-driven, rather than algorithmic, nature requires new mental models and competencies, which can be summarized as "Computational Thinking 2.0" [TDT21]. EdTech may be able to support educational approaches: Simulations are already well-investigated and successful for teaching complex topics [WAP08]. They could be helpful to teach LLMs, as well. For these reasons, a visual and interactive deep learning simulation is worth prototyping.

Apart from traditional learning materials like textbooks and collections of worksheets about artificial intelligence, more modern approaches exist in the form of interactive simulations. Different key strategies to effectively visualize deep learning in such a simulation were proposed in the community, among them visualizing a neural network through an (animated) node-link-diagram, plotting input data and training metrics, and providing interactivity up to a degree that lets users customize parameters and the network architecture [Hoh+20].
A popular simulation making use of these techniques is *TensorFlow Playground* [Smi+17] which is a web application that visualizes a neural network and offers rich options to customize the network and corresponding training parameters. However, because of the complexity of the tool, it provides novice users not with much more than a basic intuition of what deep learning and neural networks are. Furthermore, tools like `Machine Learning for Kids` and `Teachable Machine` [Car+20] by Google explicitly target students. While the tools introduced do a great job at introducing the topic and giving first insights, they do not go in-depth or allow customization by teachers or students.

Simulation Based on the introduced different strategies to visualize neural networks effectively and the features existing simulations offer or do not offer, a new visual and interactive deep learning simulation was designed and developed (see Fig. 6.5). A major focus was put on offering teachers the greatest possible flexibility in customizing the experience for students while making the simulation accessible and easy-to-understand for students in each possible configuration. This flexibility is expressed in the opportunities for teachers to disable and hide certain features such as advanced training parameters all together or to restrict certain editing options for students in a separate settings pane, only accessible in a *teacher mode*. This is valuable to not confuse students with too many options that might even not be part of the curriculum. At the same time, care was taken to ensure that advanced options are

still explained in simple terms when chosen to enable them. The simulation can also run fully offline (see subsection ‘Technology’ for details).

Canvas The centerpiece of the simulation is a canvas filling the main part of the viewport. On that canvas, users can gradually assemble a neural network by dragging *layers* from a sidebar on the right onto the canvas. There are a few basic types of layers available with the option to add new advanced types of layers with future updates without introducing any breaking changes for existing simulations. When placed on the canvas, a selection of a layer (or a *neuron* inside it) spawns a new pane in the right sidebar that explains the layer (resp. neuron) and offers additional configuration options (if not restricted or hidden by the teacher). These options include customizing the number of neurons in a layer or changing a layer’s activation function. Also, layers can be freely connected with each other, offering the possibility to not only create simple feed-forward networks but also more complex setups.

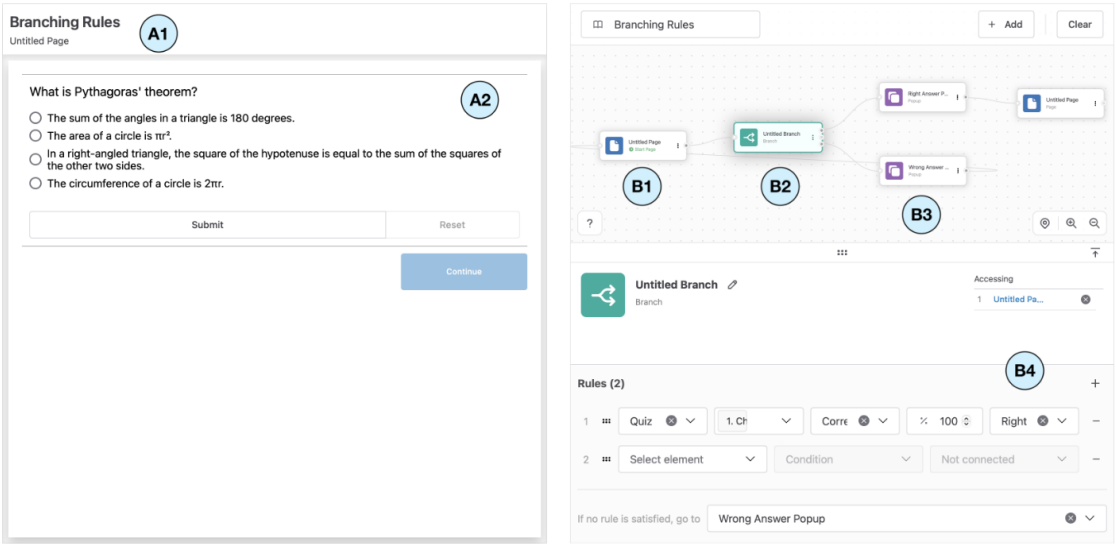
Data Input A second focus is feeding the neural network with well-formatted data. The simulation offers a few data sets by default. Users can inspect the raw data itself and view descriptions for its *features* and *labels*. In addition to that, the simulation enables users to create and upload their own data sets. This is facilitated by a multi-step wizard that guides the user through the process of first selecting the right parameters for the data set and then writing or pasting the data (in a standard, comma-separated format) into a text field. This way, a teacher can, for example, create a humorous example that involves predicting a students’ month of birth based on his/her height. It also enables the teacher (or the students) to do web searches for other deep learning data sets and use them right within the simulation with minimal additional configuration effort.

Training With a correctly designed neural network and a suitable selected data set, the user can customize training parameters and start the training in another pane. During the training, statistics are shown and the network visualization is constantly updated.

Technology The simulation is implemented as a web component with all functionality running in the browser, distributed as a Node package `@webwriter/neural-network` on NPM. It is written in TS, with major technology choices including the library `Cytoscape` [Fra+16] to control the canvas, and `Tensorflow.js` [Goo18] to handle training. This structure attains high standards of FAIRness in line with the expectations of the research community [Chu+22; KS22], especially in terms of providing a long-lived, properly versioned and uniquely identified artifact (as a package on NPM) that is interoperable with any and all web frameworks (as a web component). Additional benefits include the ability to use the simulation offline. The simulation also satisfies the so-called widget interface of the explorable authoring system `WebWriter`, enabling the seamless use and editing of the simulation directly within the authoring tool.

Use Case The simulation integrates into `WebWriter`. It may be used as part of an explorable, a kind of digital worksheet that mixes traditional content like text with hypermodal content like images, video and audio, and even interactive elements called widgets [Vic11a; Hoh+20]. For example: An educator creates an explorable for a first lesson on neural networks. They introduce the topic of neural networks with a video, then define some terms and give some examples with text. Each example could already be illustrated with an interactive simulation where they choose to disable most options to avoid distractions at this stage. For the main part of the lesson, the explorable includes exercises to be solved with the simulation, this time enabling more options for the students, allowing them to explore the simulation and to answer questions using quiz inputs like multiple choice. Finally, students can be encouraged to reflect at the end of the lesson with a simple prompt and a free text input asking them what they have learned about neural networks. While this example is very simplified in terms of didactics, it showcases the potential of the simulation. Due to the interoperability of the simulation, other uses are possible, such as embedding it in a digital textbook or integrating it into a larger learning application.

6.2.2 Branching Scenario Widget



(a) (b)

Figure 6.6: The branching scenario shows a page with a single-choice quiz (A2), represented in the branching structure by the page node on the left (B1). Using a branch node (B2), the author defines branching rules based on the student's performance on the quiz (B4). If solved correctly, the student goes to the "Right Answer Popup"; otherwise, they go to the "Wrong Answer Popup" (B3)

Advancements in educational technology (see Section 2.1) have significantly redefined the nature of learning materials and reshaped student learning. Traditional material such as worksheets can become digital, featuring interactive and multimodal learning experiences. Such digital worksheets (explorables), promote student learning and engagement [WAP08; Con21] and support personalized learning experiences

through self-paced learning and real-time feedback [Ser+19]. One key aspect of personalized learning is the tailoring of instructions to meet the needs of individual students [SS20]. Theoretically, explorables open up opportunities to create adaptive learning experiences, as they could consider the performance of a student on a quiz to branch off onto personalized learning paths or include decision trees where the student can make self-directed choices. Such functionality is similar to a digital gamebook, which Möslein-Tröppner and Bernhard [MB18] presented as a suitable method for learning. Moreover, Noichl and Schroeder [NS23] classified digital gamebooks based on their potential to facilitate learning goals, highlighting their relevance for teaching. We introduce an authoring tool that enables educators to create branching scenarios: Adaptive learning resources tailored to students' individual needs. Different authoring tools for creating branching scenarios exist; however, these tools are mostly not tailored for educational purposes, such as gamebook or interactive digital narrative (IDN) editors. Research on such editors is limited and lacks academic focus, as noted by Koenitz [Koe18]. Yet, existing editors offer valuable insights into general user requirements. Green et al. [GHC18] created `Novella`, an IDN editor featuring a node graph showing the story's branching structure. Their comparison of various IDN editor interfaces found that graph-based visualizations of story structure are the most intuitive for authors [GHC21]. Moreover, Murray and Salter [MS23] found that most branching scenario authoring tools feature a graph-based UI. A popular commercial tool that uses a node graph structure is `H5P Branching Scenario` [H5P19], which enables the creation of self-paced and adaptive learning experiences similar to non-linear slide presentations. However, the tool only supports decision trees, not branching based on interactive elements like drag-and-drop or fill-in-the-blanks. Furthermore, the popular open source tool *Twine* allows for the authoring of interactive, non-linear stories. However, the tool requires users to have coding proficiency in scripting languages to create complex branches, which severely limits its usability for many users. Our branching scenario editor allows educators to create branches based on interactive elements using a visual, graph-based interface, eliminating the need for coding and ensuring accessibility for all educators.

Branching Scenario Editor Based on the introduced research and tools, we designed and developed a new branching scenario editor tailored for educators, offering the greatest possible flexibility in creating adaptive learning experiences for students. The design and development followed Norman's iterative cycle of human-centered design [Nor02]. We developed an early prototype and gathered feedback through interviews with five secondary school educators, who interacted with the system to help shape user requirements. Building on these requirements, we refined the prototype, the results of which are presented in this work. The editor is divided into two sections (compare Figure 6.6): A node graph view at the top, which displays and manages the branching structure, and a node details view below, used for editing the content of the selected node. Each branching scenario is composed of individual pages, which serve as the fundamental units and are represented by *page nodes*. Pages are fully text-editable and authors can insert a variety of multimedia content, such as images, videos, and quiz elements. In addition to pages, authors can prompt the reader using

popups, represented by *popup nodes*. Popups are also fully text-editable, but appear layered over the currently displayed page when opened. To build a branching structure, each node can be connected to another one, forming a reachable path for the reader in the resulting branching scenario. When two nodes are connected, a navigation button is added to the content of the source node. Clicking it takes the reader to the target node. Authors can customize the button's label, size, width, alignment, and color to convey different messages and increase flexibility in authoring. Navigation buttons can be used to create decision trees for the reader; however, branchings can also be created based on the readers' performance on quiz elements, including fill-in-the-blanks questions, single- and multiple-choice questions, ordering questions, or select-the-text items. To create different learning paths based on a quiz, authors have to connect a *branch node* to a page or popup containing a quiz element. A branch node does not contain text or multimedia content, but allows authors to define branching rules, setting the learning path based on the readers' answers to the quiz element. Once the scenario has been created, students view and interact with it using the editor's viewing mode. The viewing mode systematically displays the branching scenario using an embedded canvas, featuring a header with the current page's and material's title. The canvas is scrollable, ensuring that all text, images, and quizzes can be displayed entirely. While primarily used to display branching materials to students, the viewing mode also helps authors verify their structures by displaying error messages, such as incomplete branch node rules.

Technology We implemented the editor as a web component with all functionality running in the browser, distributed as a Node package `@webwriter/branching-scenario` on NPM. It is written in TS using the `Drawflow` [Sol20] framework for displaying and managing a node graph. Users can access and use our editor seamlessly within the `WebWriter` environment as a widget. Moreover, this integration enables authors to use `WebWriter`'s entire multimedia suite and interactive widgets in our editor, such as `WebWriter`'s *Quiz widget* on which branchings can be based. Created branching scenarios are exported as HTML documents and can be used entirely offline. The development, architecture, and distribution of our editor attains the high standards of findability, accessibility, interoperability, and reusability (FAIRness) in line with the expectations of the research community [Sal+24; Chu+22; KS22]. We provide a properly versioned, and uniquely identified artifact (as a package on NPM) that is interoperable with any and all web frameworks (as a web component).

Use Case Our editor could be used for a lesson on civics. Imagine an educator creates a branching scenario, starting with a text and media about the German federal parliament. In the scenario, the students have to read the text and then can navigate to the next page, featuring a single-choice quiz. If a student answers the quiz incorrectly, they receive feedback to try again and are sent to repeat the task. Otherwise, they receive positive feedback and are directed to a popup offering a choice: continue with the topic on passing laws or learn about elections. Either option sends the student to a page containing texts and interactive multimedia about the chosen

topic, such as fill-in-the-blanks and multiple choice questions. Based on the students' performance on these questions, the student is guided down adapted learning paths offering more tailored explanations of the topics. While this use case is very simplified in terms of didactics, it underlines the potential of branching scenarios. Due to the interoperability of the material in WebWriter, different use cases are possible, such as embedding simulations or games in the branching scenarios or integrating exported materials as HTML into a larger learning application.

6.2.3 Other Widgets

Apart from the two examples introduced, there is a large set widgets available (19 packages, some with multiple widgets, at the time of writing) which were developed for `WebWriter`.

Overall, most widgets are focused on STEM subjects, with a base set of widgets comparable to state-of-the-art solutions such as `H5P` (e.g. quiz/assessment features). There are several widgets which are subject-specific simulations and tools: For computer science (`neural-network`, `automaton`, `network`, `flowchart`, `block-based-code`, `logic-circuit`), natural sciences (`map`, `phet-simulation`, `chemlab`, `chemdraw`), and mathematics (`geometry-cloze`, `geogebra`). Others are general-purpose (`interactive-video`, `slides`, `code`, `quiz`, `branching-scenario`, `timeline`, `word-puzzle`). Some widgets are still in development as of the time of writing [Sch25; Lan26; San26]. A few are discontinued or superseded by newer projects [She23; Kus23; Kug23; Phu24].

The table below (see Table 6.2) provides a complete (at the time of writing) listing with more information, and the package gallery on the website provides an automatically updated, interactive presentation of all widgets.

Package	Roles	Levels	Fields
`geometry-cloze`	Presentation, Information	2-4	Natural Science
[Gro24] Create and view geometry exercises with styling and labeling options.			
`neural-network`	Simulation, Concept	3-8	ICT
[Koc23] Deep learning in feed-forward networks with datasets, training and prediction.			
`interactive-video`	Presentation, Practice	*	*
[Vur24] Enhance learning by adding interactive content in popups to videos.			
`phet-simulation`	Simulation, Concept, Context	*	Natural Science, ICT
[TEAM] Interactive animations from PhET for physics, chemistry and mathematics.			
`slides`	Presentation	*	*
[Got25] Present content as a sequence of slides or tabs.			
`automaton`	Simulation, Concept	3-6	ICT
[Gen24] Build, visualize, and interact with different kinds of automata (DFA, NFA, PDA).			
`code`	Simulation, Practice	2-6	ICT
[Vie22] Write and run code in HTML, JS/TypeScript, Python, Java, or WebAssembly.			
`network`	Concept, Simulation	3-6	ICT
[Dao23] Visualization of network topologies. Can represent different kinds of networks.			
`chemical-formula`	Information	3-6	Natural Science
[The23] Periodic table of the elements with properties that are shown when clicked.			
`geogebra`	Concept, Information	*	Natural Science
[TEAM] Use the mathematics learning apps of GeoGebra for algebra and geometry.			
`quiz`	Practice	*	*
[TEAM] Create interactive tasks (multiple choice, order, free text, highlighting, etc.).			
`flowchart`	Presentation, Information	*	*
[Lu23] Create programming flowcharts with loops, branches, and interactive tasks.			
`branching-scenario`	Presentation, Practice	*	*
[Dym24] Create adaptable learning experiences with choices on how to continue.			
`chemlab`	Concept, Simulation	2-4	Natural Science
[Kre25] Prepare virtual laboratory environments for various topics in chemistry.			
`logic-circuit`	Simulation, Concept	2-6	ICT, Engineering
[Sch24] Create and simulate circuits with logic gates (AND, OR, XOR, etc.).			
`word-puzzle`	Practice	1-4	*
[Bog25] Generate crossword and word search puzzles with clues and interactive solving.			
`map`	Context, Information	*	*
[TEAM] Geographical map with terrain options, custom tiling, and GeoJSON support.			
`block-based-code`	Simulation, Concept	2-4	ICT
[Len24] Write block-based code (e.g. Scratch) and run it.			
`timeline`	Context, Practice, Information	*	*
[Zim25] Create/learn with a digital timeline and test your knowledge.			
`chart`	Presentation, Information	*	*
[Ngu24] Visualize data with various chart types. Explore and manipulate data interactively.			
`graph-structure`	Simulation, Concept	2-6	ICT
[Bau23] Visualize graphs, apply algorithms (Kruskal, Dijkstra, etc.), or record your own.			
`ai-blocks`	Simulation, Concept	2-6	ICT
[Sch25] Design AI models with feature engineering, training, evaluation and inference.			

Table 6.2: All packages in the WebWriter ecosystem as of 12/2025. Most are products of student theses (cited), those marked "[TEAM]" are made by the research team.

6.3 Developer Study

Section adapted from own work [Sal+25]

So far, we introduced the technical concept of widgets, providing FAIRness for web-based EdTech components. To comprehensively answer our research question (RQ1, see Section 1.4), however, we want to measure the two variables of expressiveness and usability (see Section 1.3).

In our prototype `WebWriter` [Sal25b] (see Chapter 5), we implement domain-specific design pattern for interactive elements in a development interface. We consider the design pattern useful if it is both usable and expressive for developers of these interactive elements. As such, we both elicit informal participation of developers in the early stages, and run a full evaluation measuring both constructs in the later stages.

Overall, we introduce the developer study's research design in the abstract first, which allows us to discuss design choices and fundamental limitations. Separately, we will later present the execution and results of the study.

6.3.1 Experiments: Development Projects

As our goal on the developer side is to evaluate the usability and expressiveness of the design pattern through the development interface, we need to set up a study where EdTech developers spend significant time with the interface, ideally implementing interactive elements fully. This excludes short-term experimental setups, such as workshops with observational measures or interviews and questionnaires afterwards – it is not realistic for developers to realize anything significant in that time frame, as typical development projects last months, not hours. This limited scale would threaten the validity of results, as long-term issues with expressiveness and usability would likely not be identified, since the experience would have to be heavily guided and thus biased by the researchers.

In turn, a more open-ended setup such as working with EdTech researchers and convincing them to use the pattern for their own system would be beneficial for validity (as we match our target group exactly), but has other issues: Pragmatically, gathering enough cooperative researchers to be able to control for confounding factors would prove challenging as a large time commitment and the risk of using an untested system would have to be asked of the researchers. Similarly, it seems unlikely that other EdTech researchers would allow many observational or controlling measures as this would interfere with their own work – in summary, our research setup would be hard to align with the goals of the researchers.

To strike a balance between control and openness, we opt to work with students instead of EdTech researchers. Specifically, the university environment offers the mechanism of final theses (Bachelor's and Master's theses), which in computer science degrees can involve completing a sizable software project independently (over 5 to 7 months). Students may choose the chair at which they complete their thesis themselves, so while they are a distinct group from accomplished EdTech researchers, we can at least assume an interest in EdTech. Additionally, we can assume some knowledge since research chairs screen student's applications for courses participated in or equivalent experiences. In this way, we can use the thesis projects completed at

the researchers' chair as long-term development projects over which some control can be exercised.

Concretely, students may implement a thesis project of their topical preference as an interactive element that uses the prototype's development interface. For example, a student interested in building an EdTech prototype that simulates automata for theoretical computer science lectures may implement that simulation with the prototype's development interface. The topical openness is important, as it will allow us to later make claims about the expressiveness of the design pattern.

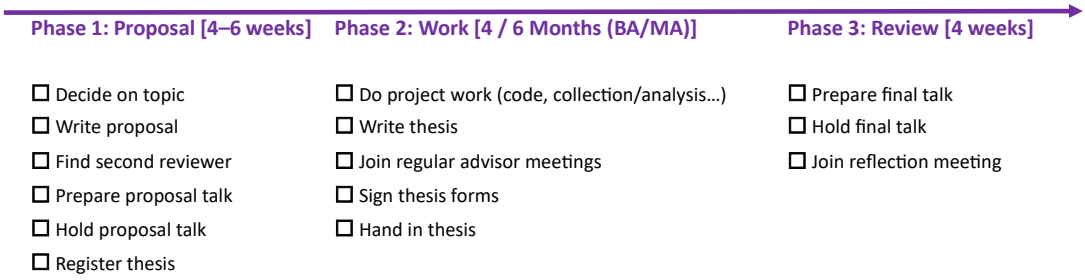


Figure 6.7: Phases of a thesis development project as shown to participating students

6.3.2 Data Collection & Analysis: UEQ+, QCA

We use three fairly common approaches for data collection. First of all, our experimental setup of development project produces a central piece of qualitative data: The interactive element (source code and artifact/runtime). As the source code is handed in with the thesis, an artifact can be built and executed by the researcher, meaning all facets of the interactive element are available for analysis. Second, we use interviews [CMM17] with the student developers to gather supplemental qualitative data about the development process. Third, a questionnaire [CMM17] is administered to control for confounding factors, and to gather self-reported data on the usability of the development interface from the student developer's perspective.

Remixability for developers The goal of our design pattern being remixability, a straightforward approach of evaluation would be to find a way to empirically measure remixability itself. However, this presents us with several difficulties: First of all, we fully acknowledge that successful remixing in a community, such as among EdTech developers, requires much more than a technical solution. Rather, a cultural shift towards sharing is needed, along with institutional incentives and the removal of legal barriers. So our solution's scope is necessarily limited to the technical world – we implement a system with technical measures that encourage remixing, but do not offer the complete ecosystem that would be needed to solve the whole problem.

For that reason, we want to evaluate the technical property of remixability only. We consider it sufficient to have shown logically how the principles of FAIRness follow from the application of our pattern (see Section 6.1), as we argue that FAIRness implies remixability.

Expressiveness & usability for developers (QCA, UEQ+) We want to measure the expressiveness of our approach for developers, the widget pattern. In the definition, we phrase expressiveness as "expressing any product desired" – for measurement, this raises the question of what the landscape of desired products looks like. Overall, we can view the produced interactive elements (source code and artifact) as qualitative content, which is supplemented by the transcribed content of the interviews.

There are several methods we could use to analyze the interactive elements produced by our student developers. For example, there is the grounded theory family of methods [WC22], which would derive classification and theories from the data without any previous schema. This would be well-suited for cutting-edge domains where no theory and taxonomy to use exists, but in our case, there is extensive research work in education and EdTech we may use to form theories, instead. Rather, if we were able to map out the space of interactive elements that could conceivably be expressed, according to current theories, we could then classify the elements implemented by our developers and see how they are distributed. If we find significant and unexpected gaps in the distribution, that could invalidate our design pattern. Another family of methods, qualitative content analysis (QCA) [May14], can give use the tools to apply classifications systematically. As we are interested in what developers desire to express, we want to classify different intents for interactive elements. We can use some of the taxonomies introduced earlier (Section 2.1), namely levels of education (ISCED), fields of education (ISCED-F), and Churchill's learning object types. We argue that if only a narrow distribution of interactive elements across each category is achieved, then the pattern is limiting in some way (for example being biased towards simulational content with specific APIs). If the distribution is broad, we consider that our hypothesis holds.

For usability, matching our requirements model, (Section 5.2), we can apply the UEQ+ to evaluate the development interface and measure the DX. The UEQ+ may be administered as part of the interviews.

To summarize briefly: In our developer study, student developers implement interactive elements with our hypothesized design pattern, which is enforced through the developer interface of our prototypical explorable authoring system. We collect the produced elements as content data, supplemented by transcribed interviews, and administer a questionnaire which controls for confounding factors and includes the UEQ+. We analyze the collected data using QCA to classify the content for expressiveness, and evaluate the questionnaire for usability.

6.3.3 Execution

We worked with a sample of students completing their Bachelor's and Master's theses in Computer Science. In total, n=11 thesis students students (9 Bachelor's, 2 Master's theses) participated over a period of one year (April 2024 to March 2025)—each thesis project lasting about five to seven months from developing the idea to handing in the thesis. It is worth noting that the first 6 of the theses were completed in `WebWriter`'s old, desktop-based system, while the latter 5 were conducted

in a reworked, fully web-based system with extended documentation. Recruitment was defined by students submitting an application for a thesis topic, which was then developed further in cooperation with an advisor. The students were in their twenties (range=22-28, median=24) with eight identifying as male and three as female. On prior experience (Likert scales of 1 to 5), all reported a high proficiency in programming (range=3-4, mean=3.36), while proficiency with web technologies was far more varied (range=1-5, mean=2.81). Proficiency in didactical topics varied strongly, including the topics of general teaching skills (range=1-4, mean=2.36) and EdTech (range=1-4, mean=2.09). Reported experience in the technical specialty topic of web components was generally low (range=1-3, mean=1.63). As a common denominator of these thesis topics, all students had to develop a widget, but were free to choose any educational subject based on interest. As the sample consisted of students completing their theses, we determined that the optimal point of measuring for each student would be shortly after they completed their thesis and grading was done. This was intended to remove the ethical dilemma of students feeling obligated to report only positive feedback. Based on the guidelines of the responsible ethics board, no issues with the study design were identified beforehand.

Students were surveyed in individual sessions, each starting with the same digital questionnaire and followed by an semi-structured interview based on a previously designed guideline. Both instruments were tested in a pre-study and improved with feedback. The sessions were conducted by a neutral person involved neither in the `WebWriter` project nor in any of the thesis projects. All participants were invited by email and were provided no prior information about the session. Sessions were all conducted digitally via Zoom and recorded with permission of the participants. At the start of the session, the interviewed student is reminded that the grading process is complete and they may speak freely about their implementation. Sessions started with a questionnaire, including a short introductory text and privacy agreement, followed by two main sections: The first section asked participants to report gender and age, as well as their experience, prior to development of the widget, with following topics: programming, web development, web components in specific, education, and EdTech. This limited personal data was intended so confounding variables could be considered as correlations in the results. The second section displayed the UEQ+ (in our case the subscales of trust, intuitive use, adaptability, and usefulness).

After the questionnaire, the session transitioned to an expert interview following a guideline. To begin, students were encouraged to show their widget and code, and were asked to shortly introduce their widget. This served to establish a rapport between the interviewer and interviewee and to prime the student to think about their implementation. Next, the interviewer stepped through four topics that matches phases of development (design/document, implement, debug/test, deploy, see Section 5.2). For each topic, the interviewer asked open-ended questions on the experience of and challenges during development. The interview was concluded by asking for the most difficult steps of development and for general feedback. Overall, the interviewer was instructed to ask follow-up questions when necessary. The topics were chosen to provide a complete picture of all experiences during development.

Package Name	Lvl.	Field	Type	Package Name	Lvl.	Field	Type
slides	1-8	*	P---	block-based-code	1-5	ICT	-SC-
interactive-video	1-8	*	P---	code	2-8	ICT	-SC-
branching-scenario	1-8	*	P---	periodic-table	3-8	Phys. Sci.	---IX
chart	1-8	*	---I-	automaton	3-6	ICT	-SC-
flowchart	1-8	*	PE-I-	logic-circuit	3-6	ICT, Eng.	-SC-
timeline	1-8	*	PE---	computer-network	3-6	ICT	-SC-
quiz	1-8	*	-E---	graph-structure	3-6	ICT	-SC-
word-puzzles	1-8	*	-E---	security-system	3-6	ICT	-SC-
map	1-8	*	---IX	chemistry-lab	3-4	Phys. Sci.	-SC-
geogebra	1-8	Math	-SC-	geometry-cloze	3-3	Math.	--CI-
phet-simulation	1-6	Phys. Sci.	-SC-X	neural-network	4-8	ICT, Math.	-S-I-

Table 6.3: All packages available in WebWriter with metadata, packages from study underlined

6.3.4 Analysis and Results

To get an overall picture of both expressiveness and usability, the interview recordings were transcribed using OpenAI's Whisper v3 model [Rad+23] locally (with some manual corrections). Next, a deductive content analysis [May14] was conducted by two researchers previously unassociated with the project, using the tool `QCAmap` [MF17]. The process was supported through a coding manual asking to identify statements based on the three dimensions: *phase* (design/document, implement, test/debug, deploy), *sentiment* (positive, negative, suggestion), and *subject* (`WebWriter`, the widget's subject domain, web technologies, package & tools, and miscellaneous). Purely descriptive statements and informal parts of the conversation were intentionally not coded. Following the recommended procedure for deductive content analysis [May14], the guideline was revised with more detailed coding instructions (and the added category of "web technologies") after completing four interviews (36%) for more accurate results.

Overall, a large majority statements extracted were concerned with implementation (47%) and testing/debugging (29%). Designing/documenting (17%) and deployment (7%) were only minor topics. For design/document, developers judged the documentation to be helpful, but desired more guidance and information for "best practices" to solve issues unique to `WebWriter`. The main phases of implement and test/debug mostly contained statements about specific bugs in the implementation which were fixed during the survey period (e.g., errors in the preview and reloading of packages), and many others were not about `WebWriter` but about experiences with other WebTech (42% and 33%, respectively). Specific implementation features were mentioned by multiple participants and rated positively (e.g., build scripts, local package with live reloading). Deployment was perceived as successful, with most associated statements not concerning `WebWriter` (61%) or being positive (25%).

The widgets created by the evaluated developers, together with the larger set of widgets made for `WebWriter` (see Tab. 6.3), show a wide range of expression in terms

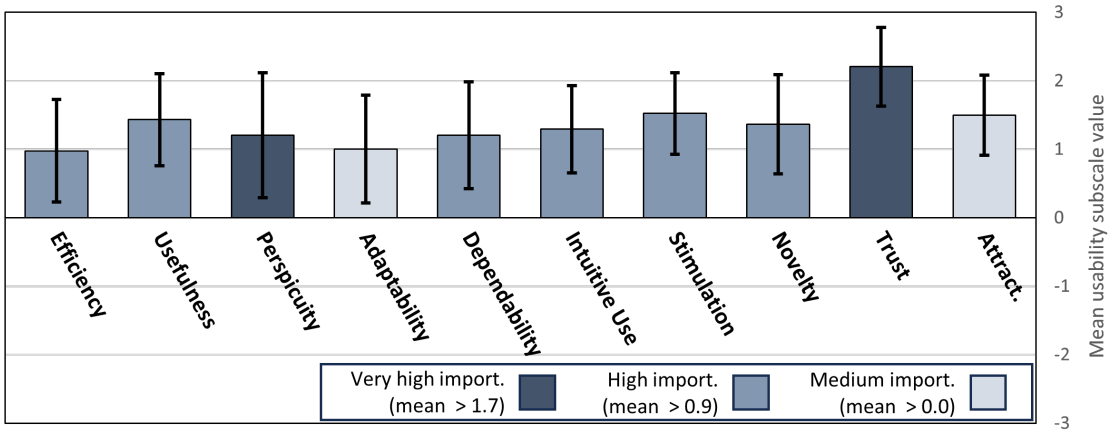


Figure 6.8: Mean usability ratings for widget development in WebWriter (UEQ+, n=11 developers). The height of the bars indicates the mean score (between -3 and 3) on each usability subscale, and the color indicates the mean importance score of the subscale.

of levels (ISCED) and fields (ISCED-F) of education, and types of EdTech [Chu07] implemented. These categorizations were made by the researchers based on the final product and the extensive descriptions provided in associated theses by the student developers. Looking at levels, 10 of the packages are usable for any field of education and any age group from elementary to doctorate level, since they are presentational (P) types (e.g., a `slides` package for arranging content as a slideshow) or exercise (E) types (e.g., a `quiz` package with multiple choice tasks), where the level and fields the widget can be applied to depend on the specific topic authored. Only ISCED level 0 (preschool) is excluded for all packages since literacy and numeracy must be assumed for the learner. More specific content was also created, with a large category of 9 packages being simulational (S) & conceptional (C) widgets (e.g., an `automaton` simulation for DFAs and NFAs). Informational (I) and contextual (X) widgets were the most rare (e.g., an interactive geographical `map` or the `periodic-table`). Finally, there are a few widgets intended only for specific topics and age ranges (e.g., a `chemistry-lab` for virtual chemistry experiments in secondary education).

As seen in Figure 6.8, developers were surveyed about usability on ten subscales, with additional weighting by importance. Results were analyzed using the UEQ+ handbook and Excel tool. Developers place very high importance on *efficiency*, *perspicuity*, *dependability*, and *trust*; high importance on *usefulness*, *intuitive use*, *stimulation*, and *novelty*; and medium importance on *adaptability* and *attractiveness*, indicated by color saturation in Figure 6.8. In general, all results are solidly positive (the standard interpretation considers > 0.8 as good), falling between 1 (*efficiency*, *adaptability*) and 2.2 (*trust*). Some confidence intervals are relatively large (e.g., *perspicuity*, *adaptability*, and *dependability*), indicating a wider spread of answers. Negative correlation coefficients for self-reported experience values and some sub scales were found, namely between programming experience and *trust* (-0,70), and between web component experience and *dependability* (-0,51). Due to the small sample size, the results above can only indicate a preliminary trend that may not replicate within a larger sample.

6.3.5 Discussion

For usability, the positive results on each sub scale show that the concept of widgets was easy to grasp for developers, with strengths being *trust* that the tool kept data safe (such as the build script working directly with the local code of the widget), and *stimulation* and *novelty* (showing that developers considered the concept exciting and innovative). The large spread of answers for *perspicuity* indicates that introductory resources could be improved. However, some limitations in interpreting the results apply: The sample size is low, so the results can only be understood in conjunction with the interview data. At first glance, there is a conflict between the many negative and suggesting statements (many being bug reports) in the interviews, compared with the high user experience scores. This can be explained by the fact that participants were aware that `WebWriter` is a research prototype, lowering expectations of *dependability* accordingly. Following this interpretation, the result for *dependability* should be judged less positively for real contexts, showing room for improvement.

The range of levels of education and types of EdTech content in the 22 widgets shows expressiveness, as broad resources usable for any topic or age group were implemented. However, results are currently limited to STEM subjects (likely due to the personal interests of the developers). While it is a strong assumption that the concept may also be usable and expressive for non-STEM EdTech, further investigation is needed. The interviews mostly show that the widget concept was more of an afterthought fading into the background. Interviewees made no statements that would indicate fundamental barriers of expressivity (e.g., “I don’t know how to do X as a widget”), with almost all negative statements concerning bugs in the implementation. Especially the deployment phase was judged by participants as easy, with most issues lying outside of the widget concept and implementation (e.g., small annoyances such as “two-factor authentication in NPM”).

As with any research design, there are several issues arising from tradeoffs made in the design, which limit the conclusions drawn above.

No Empirical Measure of Remixability We argued earlier that a valid measure of remixability would be very hard to attain, leaving us with the rather limited solution of a logical analysis of the the design pattern by ourselves. A full study of remixability would need to occur in a whole reuse ecosystem, where we observe the reuse, not the initial implementation. We consider this a pragmatic limitation in scope needed to get the research project done, but it means that results in this area must be taken with a grain of salt.

Validity of Experiment We intend to measure the properties of a theory, our design pattern – but that theory must be implemented to be used. The development interface of our explorable authoring system is supposed to be a faithful representation, supporting developers in implementing interactive elements in our design pattern. However, it necessarily has implementation details and practicalities which are not inherent in the design pattern – these practicalities become confounding factors for

our developer study. In theory, we may control for implementation details by comparing different explorable authoring systems which implement the same pattern, but this is not practically viable as the pattern is novel, meaning there are no systems to compare to.

Sampling Issues If we compare our target group (EdTech developers) with the sample we attain through our student developer projects from final theses, there are several points of mismatch: First of all, students are not EdTech developers by choice or profession, but are extrinsically motivated to write a thesis to attain their degree. While the students' choice of writing an EdTech final thesis strongly implies an interest in the topic, some students may lack other options or choose based on availability and convenience of topics. We can introduce limited controls by including items about experience and interest in the questionnaire, but the fundamental issue of motivation remains. Second, there are several sampling biases inherent in the design, including a geographical bias (as only students of RWTH Aachen University can participate), and also age and gender biases (considering enrollment patterns at the university).

Biases of Participants There are also biases present which may influence the answers given by students in the interviews and questionnaire, threatening validity. Most importantly, the researcher is not neutral and unknown to the participating students, but is their thesis supervisor. Several measure were taken to mitigate this issue and to enable students to freely express themselves, such as the point of measurement (after the project completed and a grade was received), not disclosing the goal of the study, and having another, previously unknown researcher conduct the data collection. However, these are only mitigations, as the familiarity of researcher and subject remains and their experience during thesis supervision can still influence their answer. For example, a bad grade may prompt only limited or unfavorable answers, while a good grade may lead to more positive answers.

6.4 Summary: Widgets

In this chapter, we presented the concept of widgets, which is a software design pattern that helps EdTech developers to create FAIR interactive elements.

We first (Section 6.1) described the widget pattern in theory by formulating a set of principles which an implementation would need to fulfill. These principles combine ideas of packages and components into a coherent concept. We then showed how the design pattern may be implemented on the web using the domain standard technologies of Node packages and web components. We went into detail on how widgets may be reused, including through package managers in other code projects, through a CDN service on other websites, or through authoring tools. For reuse through authoring tools, we describe a domain-specific interface implemented in the `WebWriter` prototype which allows widgets to specify their appearance and behavior inside authoring tools (edit mode, editing behavior, nesting, localization, etc.).

Next (Section 6.2), we looked at two examples of finished widgets in depth to better understand the pattern. First, we looked at a complex simulational widget called `neural-network`, which is a deep learning simulation with a visual editor to create, train, and use a neural network. This widget showcases how even advanced web technologies such as web workers (for training) and a combined canvas and DOM-based UI work within the widget concept. Second, we presented a presentational widget called `branching-scenario`, which allows educators to create adaptive learning scenarios where learners may choose from different paths. The capabilities of this widget show that complex nested structures are also possible (each branching scenario page containing more content, possibly other widgets), even offering a custom interface for manipulating that content (the graph editor to arrange pages and transitions).

Finally (Section 6.3), we outlined the evaluation of our widget design pattern. This includes the experimental setup (development projects with aspiring EdTech developers across 4-6 months), the data collection & analysis methods (UEQ+ and QCA), and some inherent methodological limitations (no empirical measure of remixability, fit of experiment and theory, sampling issues, and biases of participants). We presented the results including demographic data (n=11 developers), and measurements (usability and expressiveness), also possible correlations. We discussed these results, stating that while broad expressiveness and high usability are indicated, the sample's quality and size limit any conclusions.

RESEARCH SUMMARY

CT: In the context of a technical solution for web-based, interactive resources to support active learning, aimed at post-primary STEM learners:

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

ME: Create a prototype with participatory design. Evaluate it in experiments with authoring workshops and development projects, for each proving remixability theoretically and measuring usability (UEQ+) & expressiveness (QCA).

PR: *WebWriter*, a single page web app implemented with web technologies in the MVVM pattern, offering both an authoring and a development interface.

- **RQ_1**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
  - **HY_1**: Interactive elements are remixable if they are findable, accessible, interoperable, and reusable (FAIR), which is possible by applying the component and package patterns and adding a domain-specific API.
  - **PR_1**: *Widgets*, which extend Web Components with a minimal interface for blackbox reuse through authoring tools. They are published and accessed as Node Packages with learning-related metadata.
  - **EV_1**: Developer study over 4-6 months with n=11 participants creating widgets, with a post test consisting of interviews and survey. Results showed broad expressiveness and a positive developer experience, but the sample selected is limiting.

- **RQ_2**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?
  - **HY_2**: Authoring should support the remix workflow for educators by extending rich text editing with a new paradigm to cover morphological changes to interactive content.

CT=Context, RQ=Research Question, ME=Method, PR=Prototype, HY=Hypothesis, EV=Evaluation

Chapter 7 Explorable Authoring

In this chapter, we address the second research question regarding an authoring paradigm for educators to create remixable interactive learning resources.

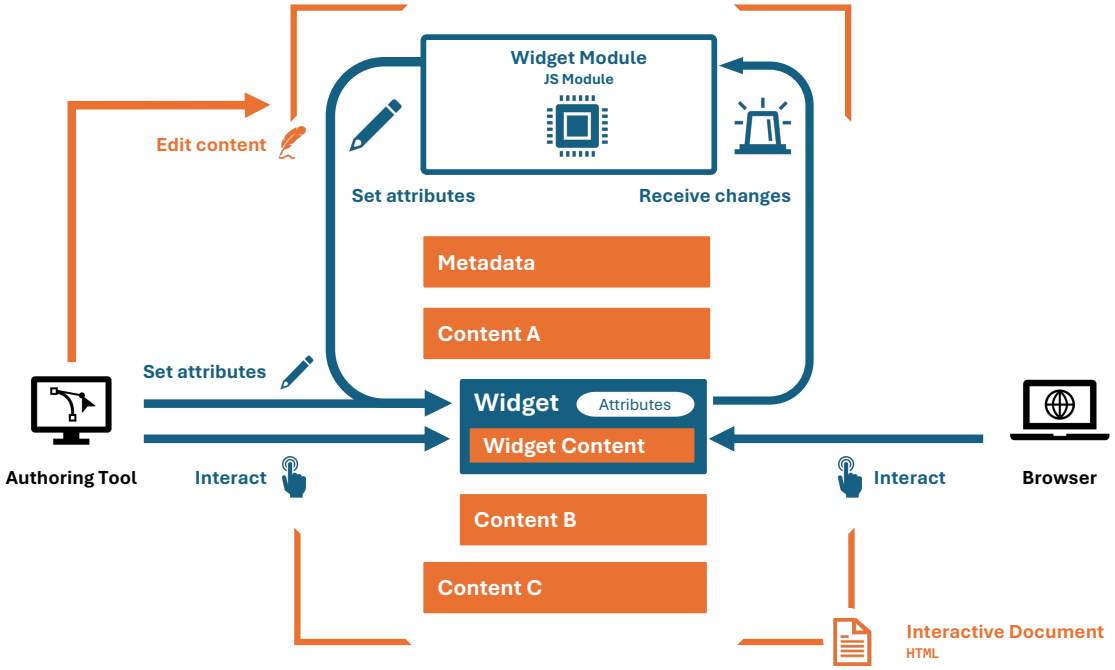


Figure 7.1: The explorable authoring paradigm

Parallel to the previous chapter, there are three sections: First (Section 7.1) we formulate the authoring paradigm in the abstract. Next (Section 7.2), we show examples. Last (Section 7.3), we present the evaluation of the concept with educators including study setup, data analysis, results, and a discussion.

7.1 Explorable Authoring Paradigm

To support the remix workflow for interactive content, we seek a new authoring paradigm which makes creating interactive content more usable without sacrificing expressiveness. We first define interactive documents, and then introduce explorable authoring as an extension of rich text editing.

7.1.1 Explorable Authoring in Theory

To discuss how interactive content could be authored in a usable and expressive way, we should define that content. On a conceptual level, we can think of interactive documents as an ordered tree of content nodes, which may have different types.

Interactive Documents Documents which are interactive may also contain static elements such as sectioning, media such as images, and so on. Furthermore, metadata may also be included as a type of element. These types of element come from a limited set. Crucially, they may contain widgets (Chapter 6), instances of interactive elements which derive their behavior and appearance from a widget module (also included in the document). As widgets are defined dynamically, they do not come from the same limited set of elements. All elements in an interactive document, including widgets, may have two types of child nodes: First, attributes, which are serializable key-value pairs used for storing limited amounts of state. Second, content, which may be any nested structure permitted for the node type. This uniform interface for elements allows offering a standard API which even applies to the dynamic and unknown widgets – that widgets are elements is what will enable our authoring paradigm.

In practice, this definition of an interactive document mirrors HTML documents and the DOM – however, we left out many implementation details which would be unnecessary for the authoring paradigm. In the abstract, the paradigm which we will present would work with any kind of document with the given properties.

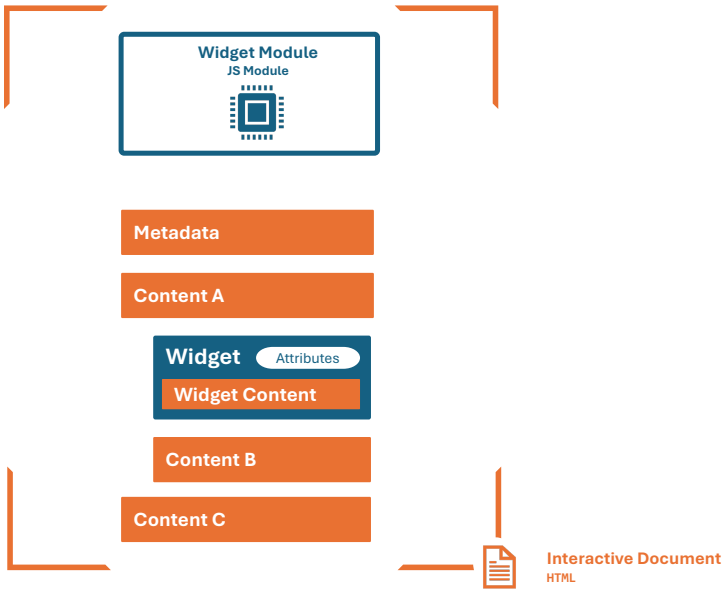


Figure 7.2: Interactive documents as the basis of explorable authoring

Self Editing & Explorable Authoring Previously (Section 3.2), we introduced direct manipulation as the overarching idea behind many popular modern tools (e.g. Word, PowerPoint): Authors directly interact with an accurate representation of the content to change it (WYSIWYG). For static content, the rich text editing paradigm implements the idea successfully. Authors see the formatted text as readers would see it, and may type out, select, copy, cut, and paste content at will. For specific types of content such as images or tables, content-specific affordances are provided.

We could imagine that interactive content may be supported the same way, e.g. offering a checklist content type for which the authoring tool renders clickable check boxes. This approach is taken by many rich text tools (e.g. OneNote). That would be a strong integration of interactive content into the document, however, it has an obvious limitation: Only a fixed set of interactive content types well-known to the authoring tool is possible. On the other end of the spectrum, tools may opt for a weak, encapsulated integration: Content may render and behave in any way, shape or form inside an isolated sandbox, but not influence the rest of the document. This is how interactive content is commonly reused on the web (iframes). However, this approach limits expressiveness as nested content becomes impossible, and cross-element interaction is not allowed.

So how may we extend rich text editing for interactive content, striking a balance between expressiveness on the developer side (expressive interactive elements) and the authoring side (expressive interactive documents)? We contend there is no usable way to accomplish this on the side of the authoring tool alone – the solution lies on the side of the interactive element itself:

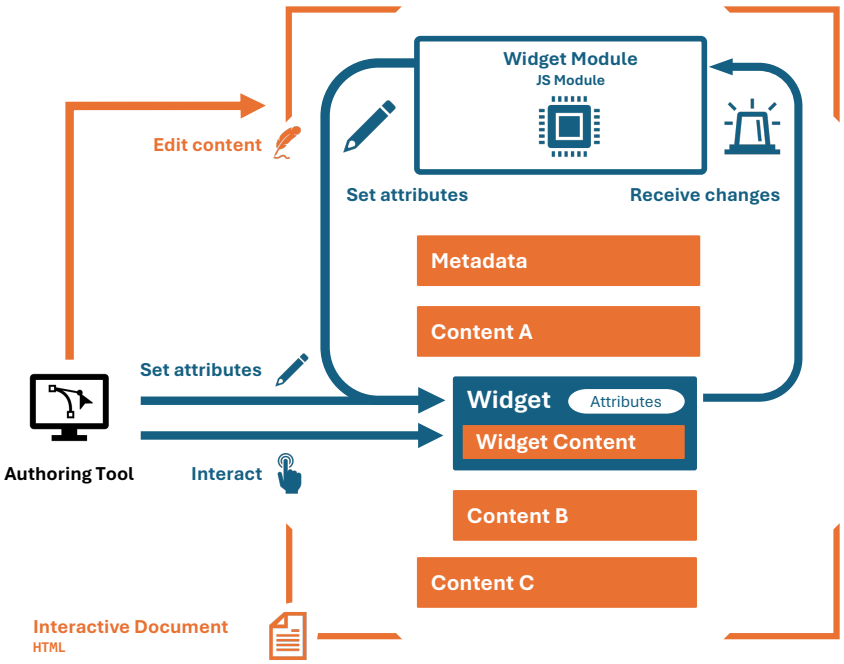


Figure 7.3: Authoring tool perspective of explorable authoring

Self Editing Paradigm: Interactive elements may provide affordances for direct manipulation of themselves and their content, and reactively update when their state in the document is changed.

We can better understand this idea with an example: Imagine a simulation for automata in computer science (DFAs, NFAs, and so on). The simulation component shows a single automaton which is represented in the standard graph notation. If we were to use the form fill-in interaction style, there might be a way to enter the node names and transitions in separate form fields and recompute the simulation. Instead, the component takes a direct manipulation approach: Nodes can be dragged and their names clicked to be renamed. They can also be connected with new transitions via drag and drop. An authoring tool would simply communicate to the component that it is being edited, causing the component to offer up the described affordances: The component is responsible for manipulating itself. Component and authoring tool must only interface at the level of state management, as changes in the component must trigger matching changes in the document. As such, the authoring tool observes the component. That covers the component changing itself, but the component must also react to changes in the document, for two reasons: First, each time the document containing the component is opened again, the component needs to be "rehydrated" with its current state that is stored in the document. Second, other components may want to programatically modify the component's state, such as a text input component that could be connected with the automaton component to input an automaton with mathematical notation.

To our knowledge, this paradigm is novel in the current research, and there are only a few similar concepts and attempts, but no proper implementations. One framing is provided by Aßmann, who calls this invasive composition [Aßm05]: The included component has the power to "invasively" change the document itself, it is not sandboxed as in classic component systems such as CORBA. Reactivity (also mentioned by Aßmann but called "transconsistency" [Aßm05]) is a more well-known concept, used both in programming contexts such as web development (React being so closely tied to the concept as to be named after it), but also spreadsheet software such as Excel.

Integrating self editing and rich text editing is simple, but powerful: Recall that we distinguish between static elements and widgets as content types of our interactive document. For static elements, authoring tools can work exactly as other rich text tools, rendering the elements as they would appear in the final document and providing pre-defined affordances depending on the type of content (a caret when editing text, drag and resize handles when editing images, and so on). For interactive elements, authoring tools place the element in the document and let the widget module render it in place. As widgets are elements, some standard manipulations applicable to all types of content can still be provided: Widgets can be inserted, selected, deleted, copied, cut, pasted, and so on, including nesting static elements inside widgets and vice versa. As the widget is rendered in place in the document, authors may interact with it (e.g. clicking on a check box in a checklist). This interaction is handled by the widget module, which triggers a change in attributes (or content). This

way, the document is changed through the widget module, not through the authoring tool's logic. As an additional way of editing apart from interaction, the authoring tool may also set attributes directly.

In summary, our paradigm, which we call explorable authoring, adds widget-handled interaction and setting of attributes as two more ways for authors to manipulate the document.

> **Explorable Authoring**: Treat interactive documents as rich text, with rich text editing behavior for static elements. For interactive elements, extend this behavior through self editing, where interactive elements may provide affordances to modify themselves depending on the author's interaction.

Interaction as Limited Authoring Coming from the perspective of the remix workflow, change of the interactive document does not stop with the author (educator), but also extends to the reader (learner). In the metaphor of paper worksheets, our authoring paradigm covers how the educator designs the worksheet, but not how the learner scribbles on their copy. For example, the educator may add a text box to the paper worksheet, and the learners then fill out some answer with a pen. This interaction by the learners can be understood as a limited form of authoring – learners are in fact co-authoring the worksheet, turning it into their own, unique copy (or "fork" in programming terms). Still, they are not intended to change most of the worksheet, rather, there are fixed points at which participation is encouraged.

Explorable authoring is thus incomplete, unless the reader (learner) is also considered. As a solution, we can open up the self editing loop to the readers: Readers may also interact with widgets, which triggers a change in the document. This also clarifies the distinction between an authoring tool (such as `WebWriter` [Sal25b]) and a reading tool (such as a browser): The browser renders the document, including the widgets. Through the reader's interaction, the document changes (and changes may then be saved). An authoring tool can tap into that same self editing loop, but also provides traditional rich text editing capabilities on top. Shortly: Readers can self edit widgets through their browser, and authors can self edit as well as rich text edit through an authoring tool.

Of course, this distinction represents a design decision: We may also erase the distinction between author and reader completely, and let all document access happen through the authoring tool. This is common in some web-based authoring tools such as Miro (collaborative boards). However, that is also the point at which our focus on education influences the paradigm – educators and learners are not the same. That learners have less degrees of freedom to change the document is beneficial to learning settings, as educators can exert control and tailor a learning experience, carefully encouraging a certain learning path. If learners had no limitations, there would be no control on the side of educators, meaning that learning resources do not provide much value at all. Still, readers may become authors and vice versa, depending on the context. For example, a teacher may create their own learning resource while using resources created by other teachers as sources – or a learner may create their

own learning resource as an exercise. Overall, for education, authoring and reading should be separate activities.

Explorable Reading: Render a document and allow self editing for interactive elements, where those elements may provide affordances to modify themselves depending on the reader's interaction.

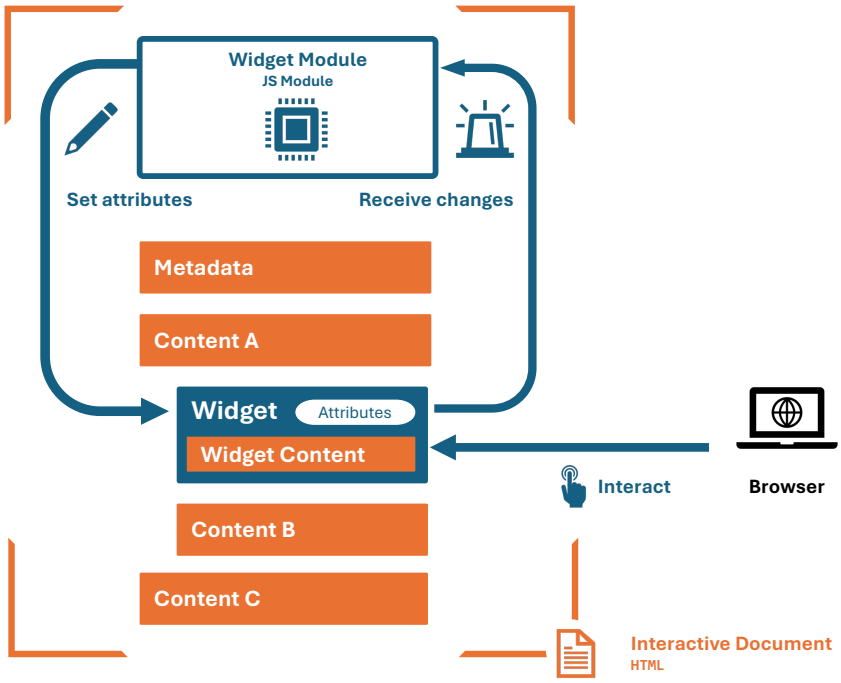


Figure 7.4: Browser perspective of explorable authoring

7.1.2 Explorable Authoring on the Web

Previously, we introduced the explorable authoring requirements (Section 5.2), specifically the authoring functionality requirements of the model: *Create, save/load/share, undo/redo, extend, add/remove, edit, arrange, copy/paste, style, generate, attribute, and license* (EAR-AF1 to 12). In `WebWriter`'s concept (Section 5.3), we showed how each requirement maps onto the UI: Document management is done in the app bar (EAR-AF1 to 3), elements are installed and added with the palette (EAR-AF4 and 5), the main editing operations are done in the document view (EAR-AF6 to 9), further and overlapping editing options are provided in the toolbox (EAR-AF6 to 9), and the metadata editor allows adding attributions and licensing (EAR-AF11 and 12).

We can say that an implementation of explorable authoring is most closely concerned with the main editing operations of *editing, arranging, copying/pasting, and styling*. The paradigm mainly differs from others in these operations. For a web-based authoring tool implementation such as `WebWriter`, we need to answer the question of how each operation can be realized, both for the web's built-in set of HTML elements (Table 5.4), and for widgets (Section 6.1). Additionally, including widgets means that

the operation of *extending* is also a concern, as developer-defined widgets are not known at the time of tool development and need to be included dynamically.

Handling built-in elements Built-in elements such as paragraphs or headings are not editable by default, but all support the web standard `contenteditable` attribute. If set on an element, that element's sub-tree is presented in an editable fashion (Figure 7.5), where a selection caret is shown and text can be typed in, copied, and pasted, for example. The behavior is hard to customize and can vary across browsers, which `WebWriter` solves by using the `ProseMirror` [Hav17] library (see Section 5.4). With `ProseMirror`, the editing behavior of each element can be customized. This allows `WebWriter` to implement custom operations for content such as lists (start a new list item when pressing enter, nested lists when pressing tab) or tables (handling rows and columns). Overall, this is a realization of the rich text editing core of explorable authoring, as rich text editing typically also includes affordances for specific types of elements (consider word processors handling lists and tables, for example). The operations of arranging and copying/pasting are subsumed in the normal keyboard shortcut or context menu approach, as even complex structures can be selected (Figure 7.6) with mouse of keyboard interaction and then copied, cut, and pasted.

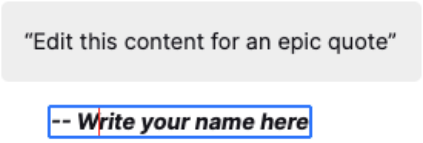


Figure 7.5: Native behavior of the `contenteditable` attribute (in the Firefox browser)

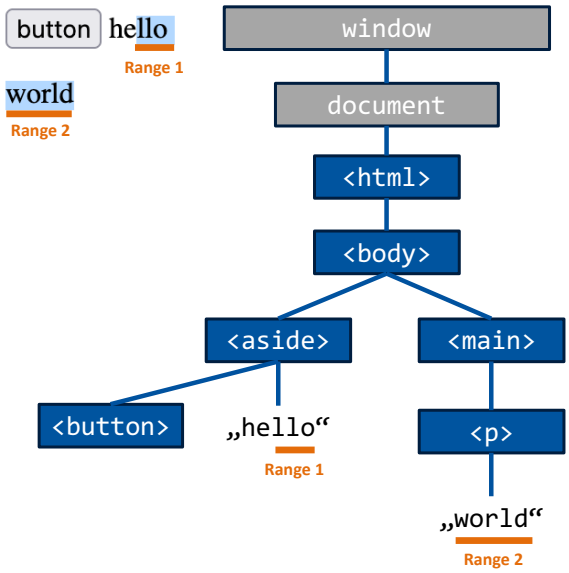


Figure 7.6: Complex selections in document trees on the web

Handling widgets The unique aspect of an explorable authoring implementation of course lies in how widgets are handled, as this is the most complex issue. Recall that the heart of explorable authoring is the loop of self editing, where interactions with a widget cause the widget module to update the document (usually updating the widget itself).

To realize the loop, the first step for a tool is to allow extending the document with widget modules. In `WebWriter`, this is accomplished through the palette. When a widget package is installed in the palette, all widget modules are added to the document, causing each widget to be defined. When documents are saved later however, only the modules of widgets actually used in the document are included. This setup in the background means that when any widget is added, the widget module is called immediately to initialize the custom element through the custom element lifecycle. When the widget is added to the document, an authoring tool such as `WebWriter` should set the `contenteditable` attribute on that widget to communicate to the widget module that the widget is reused in an authoring context. After initialization, the widget is ready to handle interactions.

From the perspective of implementing an authoring tool, the next step is to deal with the widget modules as an update source. In normal rich text editing tools, the document is always changed by the authoring tool itself, normally prompted by the user. For example, the user may delete a section, triggering a matching change in the data model. With the self editing paradigm however, we introduce a second source of changes: The module of the self editing interactive element. Consider an interactive element which allows us to draw vector shapes on a canvas. The element's module processes the input (e.g. drawing a line with the mouse) and updates the element's DOM instance with a new attribute value (e.g. changing a `shape` attribute which stores the drawn shapes with coordinates). That example is simple, but updates of any complexity are possible. Elements may manipulate a whole part of the document tree for example. So overall, updates may originate from element scripts, but should mostly be treated in the same, transactional way that user-prompted document updates are treated. This is necessary to create consistent behavior between changes triggered by elements and other changes, for example to undo/redo the lines drawn on our vector canvas normally. ProseMirror provides limited capabilities to register DOM changes and parse them, but is not intended for the self editing paradigm. By default, an infinite loop of parsing and rerendering occurs which crashes the application. To work around this, a so-called `NodeView` for widgets was implemented in `WebWriter` which customizes the change handling and rendering behavior of the element. Shortly: If a change of the widget's content or attributes is registered by ProseMirror, a special kind of update is triggered where the whole sub-tree of the widget is rerendered, but that rerender is marked to not trigger another update.

With the logic of handling updates in place, widget interactions can now cause permanent changes in the state of the document, for example by changing an attribute or rearranging content. As such, widget interactions can realize any editing or arranging operation. There is also a range of specific interfaces between widget and authoring tool that can be supported, such further defining the selection and

nesting behavior, handling interactions such as focusing and printing, and so on (Section 6.1). These are not integral to explorable authoring, but can help provide a closer integration with the web platform.

7.2 Examples of Explorable Authoring

In this section, we will briefly present a few examples for the kinds of interactive learning resources that can be created with explorable authoring.

7.2.1 ExplorableJS Digital Textbook (2023, Bachelor Level)

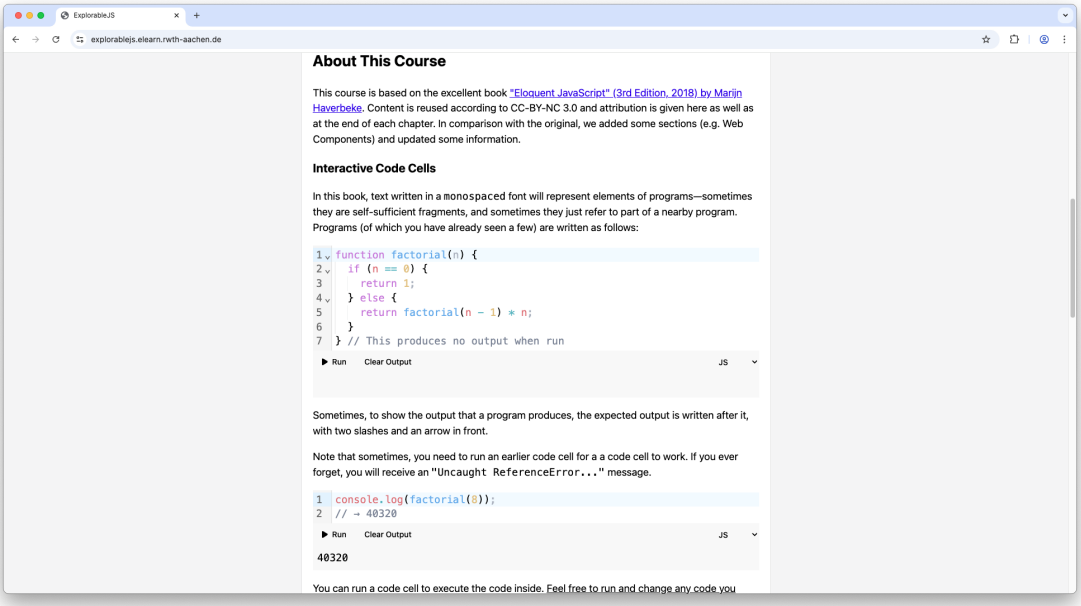


Figure 7.7: ExplorableJS JavaScript course based on Marijn Haverbeke's "Eloquent JavaScript"

As of 2023, JS is assumed to be the most widely used programming language in the world (see Section 2.2). It can power any type of web project, from a simple "Contact Me" form on your personal website to complex web applications such as `Facebook` [Zuc+04] or `Figma` [Fig16]. Due to this significance, JS is commonly taught alongside HTML and CSS in many contexts, including schools and universities. In 2023, for the "Introduction to Web Technologies" lecture at our research chair, the team wanted to update the previous video-based JS course offering (a supplement to the lecture). The idea was hatched to adapt the OER licensed, web-based textbook "Eloquent JavaScript" [Hav19], which could as a first test run for an early version of `WebWriter`. Eloquent JavaScript is an optimal base for the project since it mostly features static content, interactive code cells being the only non-static feature.

ExplorableJS [Sal24] is a set of 22 lessons intended for flipped classroom usage. For the course, the students complete lessons parallel to the lecture at home, and apply their knowledge in in-person exercise sessions. Each lesson is mainly text-based, with a running explanation supported by images and frequent code snippets, which can be changed and executed by readers. Lessons end with sets of exercises, which can again be completed using the interactive code cells. The course assumes no prior knowledge in web technologies and little experience in programming, meaning that students can use it at their own pace depending on their (wildly varying) previous experiences.

In terms of explorable authoring, this first example showcases that simple content in the form of computational notebooks can be authored, covering an important type of learning resource in practice, especially in tertiary STEM education. Specifically, the example hints that explorable authoring as implemented in WebWriter may be as expressiveness as the popular tool Jupyter.

7.2.2 Cryptography School Lab Unit (2024, Grade 5-6)

The unit is based on a 2023 master's thesis completed in the WebWriter project [Fer23]. It also integrates a few ideas from a collaboration with IT security researchers [Löb+23].

Today, an integral part of children's lives is online [Nat23; Gra18]. Like in the offline world, the online environment presents risks and potential harms to children [QCJ21; Tsi+16], particularly related to security and privacy. While some research addresses children's understandings of security and privacy topics such as viruses [Han+19; Tsa+20], cryptography [LM20], dark patterns [Ren+24] or the Internet in general [BW21; EK19; Bro+21; Bab+23; Bro+23; Mer19], learning materials especially for younger children (e.g. in grades 5 and 6) are sparse. Curricula challenge educators to address the topic [UK13; Ges08] – locally, in the German state of North-Rhine Westphalia, teachers now face a greater need to cover security topics as Computer Science has become a mandatory subject in secondary schools. Due to this need, we developed a unit for the local school lab InfoSphere, which is a place groups of students such as classes from local schools can come to to learn about Computer Science topics.

The unit is planned for a total duration of four hours (one afternoon at the school lab) and is intended for students in grades 5 and 6, which is reflected in the difficulty, language, and visuals. It has two facets: First, it teaches children the basics of cryptography and password storage. Second, based on that knowledge and on expert recommendations, it gives advice to children on how to pick good passwords.

From the perspective of explorable authoring, the unit is similar in scope to ExplorableJS, but leans more into subject-specific simulational content with the cryptography widget. It also marks a first creation from scratch, as ExplorableJS was only an adaptation of "Eloquent JavaScript". Furthermore, it is intended for blended learning instead of a flipped classroom setting, showcasing that the concept also fits into in-classroom use on laptops and tablets.

7.2.3 Block Based Programming School Lab Unit (2025, Grade 5-6)

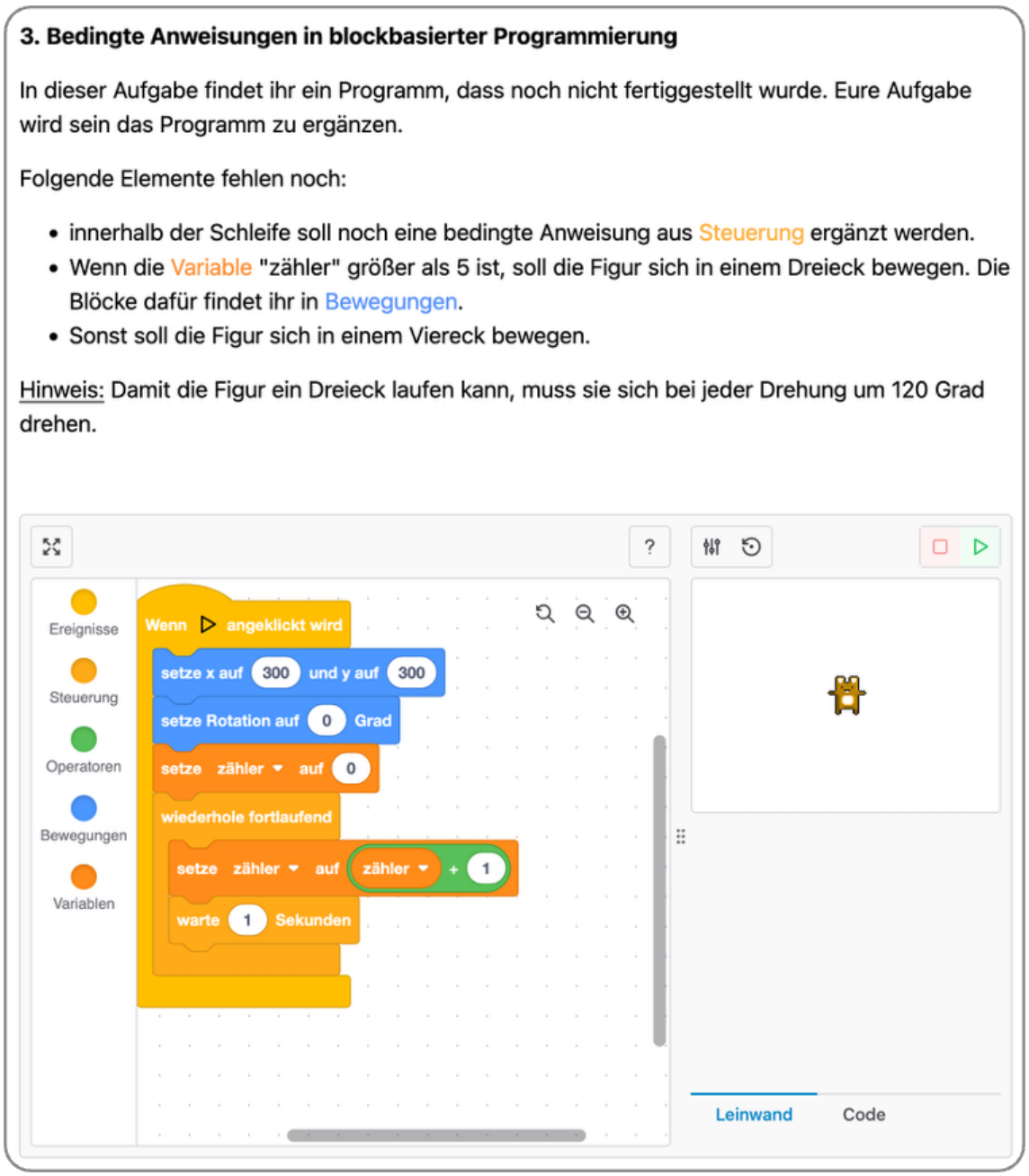


Figure 7.8: Explorable used in the block-based programming unit by Laura Marten

This example is based on a Master's thesis currently being completed in the `WebWriter` project [Mar26].

In schools, digital devices are being used increasingly in lessons; however, when it comes to worksheets, work is still done either with paper-based worksheets or with a digital equivalent. These sheets are generally not designed to be interactive, so students only receive feedback on their work when the teacher goes over the worksheet after the working phase. In the German state of North-Rhine Westphalia, since the school year 2021/22, computer science has been a compulsory subject at the lower

secondary level (Erprobungsstufe). Within the content area “Algorithms”, students are often taught with block-based programming. Most computer science teachers teach these contents and competences using Scratch, which, however, offers a great deal of potential for distraction. This is precisely the point at which a case study of the Master’s thesis is intended to tie in.

In the ongoing thesis, a four hour unit for the school lab InfoSphere (similar to the cryptography unit) is being developed. The goal is to foster an understanding of the basic attributes of algorithms (e.g. determinism), and apply them in a block-based coding environment. The unit is a blended learning scenario which uses traditional digital tools such as a `PowerPoint` [Mic90] slide show, but also an explorable created with WebWriter which students use on individual devices. Additionally, a few exercises and games with physical puzzle pieces are included to break up the long digital sections and sustain the young students’ motivation. The explorable, which is used continuously throughout the unit, contains several exercises: Multiple choice quizzes at the beginning provide an easy way for the kids to get familiar with the explorable, and to test their knowledge on the attributes of algorithms. Later, the block-based code widget is used in several exercises of increasing difficulty, ending with a longer section where students complete a little game using the programming constructs they were introduced to previously.

For explorable authoring, this is the most complete and recent example of the type of interactive learning resource that could be created: It mixes static content and a block-based programming widget with different quiz tasks and multimedia, showcasing the full range of content that could be created. In technological terms, we can see that much of the learning content available could be expressed with some subset of these elements: Subject-specific widgets, general-purpose assessment/quiz widgets, and rich text with multimedia.

7.2.4 Further Examples

On the website[1], there are a few other, smaller examples available. We include a small collage of examples here (Figure 7.9) to gain a brief impression, but recommend visiting the website to experience the interactive documents as intended.

[1] `https://webwriter.app/explorables/`

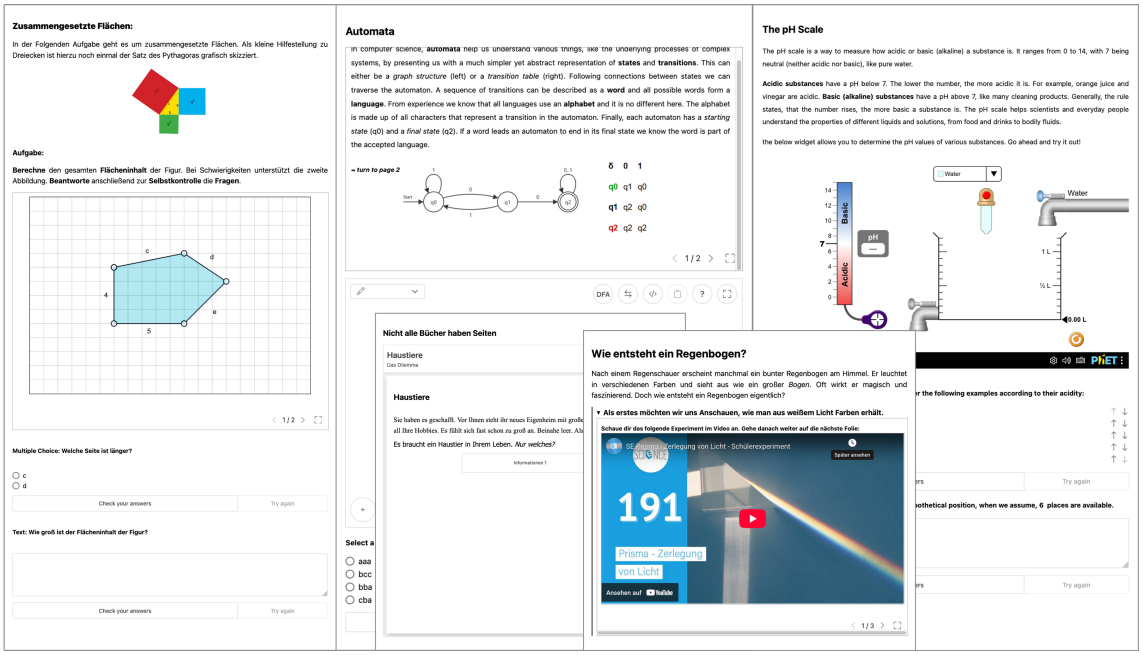


Figure 7.9: Collage of five example explorables for STEM topics

7.3 Author Study

Previously (Section 7.1), we introduced the technical concept of explorable authoring, which is a way to achieve FAIRness for interactive learning resources. The authoring paradigm is implemented in our prototype WebWriter (Chapter 5). We also looked at a few examples of explorables in different learning contexts (Section 7.2). To comprehensively answer the research question (see Section 1.4) however, we need to measure the two variables of expressiveness and usability to show the viability of our authoring paradigm.

Section adapted from own work [SRS25]

7.3.1 Sample

For both user experience and our working definition of expressiveness, real educators should author explorables in a controlled setting. Therefore, we recruited educators using contacts to local schools and universities to advertise our workshop. No exclusions or inclusions were made by the researchers, allowing all interested educators to register and participate. As participation was voluntary, a bias towards more motivated educators was expected, as well as participation only by educators local to the area.

Of n=23 participants who took part in our study, 13 identified as female and 10 as male, and ages between 20 and 62 (median=28) were reported. 11 of the participants were students of education, 9 were school teachers (one being both a student and a teacher), and 5 were lecturers (one being both a school teacher and a lecturer). On prior experience (Likert scales of 1 to 5), participants reported a high proficiency in ICT skills (mean=3.35), education (mean=3.13), and learning resource

design (mean=2.96), and medium proficiency in digital learning (mean=2.61) and digital learning resource design (mean=2.39).

7.3.2 Method

To give educators ample time to author a realistic explorable with the authoring interface, a long session is needed. The session was conducted with up to 20 participants by a moderator and an observing researcher. The session started with an introductory part where the moderator asked for experiences with digital authoring tools, which were gathered in a word cloud. Next, there was a short section where the moderator demonstrated the major features of the authoring tool while participants follow the moderator's steps on their devices. The main part of the session is an open working phase where participants could try the tool on their own. For guidance, the participants were requested to author a single explorable for a topic of their choice. They were also provided with a checklist of elements to include in their explorable to encourage using the full scope of the tool. Concluding the session, participants were asked to present their explorables, which were saved (also stored for later analysis) and shared among the group.

Afterwards, participants completed a digital post questionnaire: The first section covered some socio-demographics and personal details (gender, age, occupation, levels of education taught on the ISCED scale [UNE12], fields of education taught in the ISCED-F taxonomy [UNE14]), followed by questions on previous experiences (whether WebWriter had been used before & self-assessed competency in digital technology generally, education, teaching by digital means, design of learning resources, and design of digital learning resources). That data was collected to consider confounding factors and influences on expressiveness and usability. The second section included the UEQ+ [ST19] (parallel to the developer study including the subscales of trust, intuitive use, adaptability, and usefulness).

The workshop design was created collaboratively between a moderator with an education background and a researcher familiar with the authoring tool. To avoid introducing biases, no information about the relation between the researcher and the tool was given beforehand. Also, while the whole workshop was conducted by the moderator, the survey was administered by the researcher to lessen the effect of possible sympathies or antipathies developed during the session. Data protection guidelines and the ethics guidelines of our responsible ethics board were considered beforehand and the study was deemed appropriate.

7.3.3 Analysis and Results

As seen in Figure 7.10, educators assign very high importance to *adaptability*, *dependability*, *intuitive use*, and *trust*; high importance to *perspicuity*, *stimulation*, *novelty*, and *attractiveness*; medium importance to *usefulness*, and low importance to *efficiency* – indicated by color saturation. Most results are positive, ranging between 0.87 for *efficiency* and 1.84 for *stimulation* (the standard interpretation considers > 0.8 as good), with *adaptability* (mean=0.59) and *dependability* (mean=0.62) being

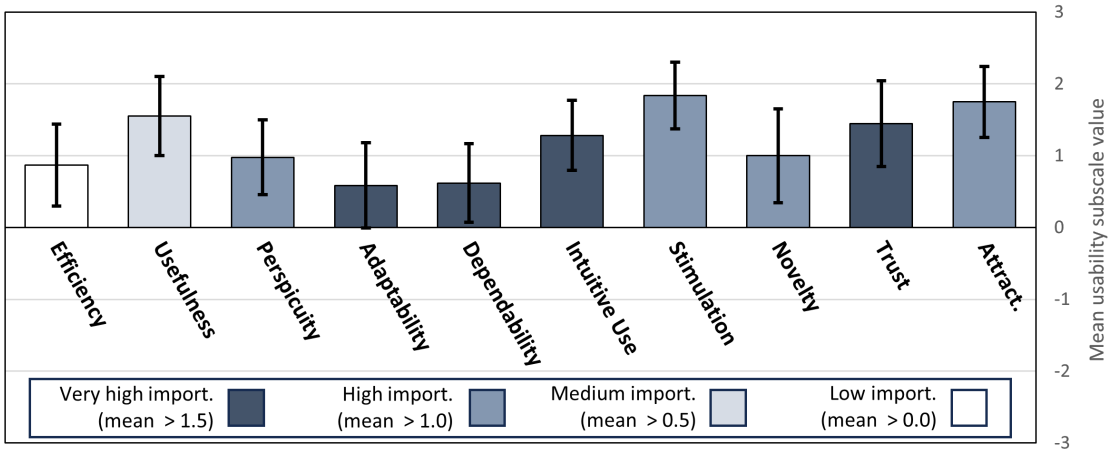


Figure 7.10: User Experience of authoring explorables with WebWriter (UEQ+, n=23). The height of the bars indicates the mean score (between -3 and 3) on each usability subscale, and the color indicates the mean importance score of the subscale.

ID	Lvl.	Field	Elem.	Widgets	ID	Lvl.	Field	Elem.	Widgets
01	1-2	Hum/SoSci	T----DM	-E-----	10	2-3	PhySci	T----DM	PE-----
02	2-6	Bio/Hea/Hyg	T-----M	-E-----	11	1-2	ICT	T----DM	-ESC---
03	2-5	Hum	T----DM	-E---I-	12	2-5	PhySci	TL-----	---SC-X
04	2-2	Math	T---FDM	-E-C---	13	1-5	Arts	T-----M	-E-----
05	1-2	Ltrcy/Lang	T----DM	-E-----	14	2-5	PhySci	T----DM	-ESCIX
06	2-2	Math	T-------	-E-C---	15	2-5	PhySci	T---FDM	----CI-
07	2-2	Math/ICT	TLB-DM	-E-----	16	2-5	Hum	T----DM	P----IX
08	2-5	PhySci	T----DM	-----I-	17	2-5	Eng	TL----M	---SC-X
09	3-6	Lang	T----DM	P-------	18	1-2	Ltrcy/Math	T---FDM	-ESCIX

Table 7.1: Finished explorables from the workshops (k=18) with coded metadata

more of a neutral result. High negative correlation coefficients for self-reported experience values and some sub scales were found, namely between education expertise and *perspicuity* (-0,57), and between digital learning expertise and *attractiveness* (-0,51).

During the workshops, each participant authored an explorable and submitted it in the end. The resulting explorables were classified using deductive content analysis [May14] by two researchers. The classes for coding were the ISCED education levels [UNE12] and ISCED-F fields [UNE14], also the contained types of elements according to the explorable authoring requirements (text, list, table, formula, disclosure, media) [Sal25a], with a distinct classification of widgets based on Churchill's taxonomy [Chu07]. Formally, only k=18 explorables were considered for coding as others were found to be incomplete or incoherent (byproducts of participants trying out features).

The results in Table 7.1 show that all types of elements and widgets were used by participants. More traditional content types such as text ($T = 18$ times coded) and media ($M = 16$) were used by almost all participants, with more advanced rich text content such as lists ($L = 4$), tables ($B = 1$), and formulas ($F = 1$) being used

more sparingly. For widgets, most explorables contained an exercise ($E = 13$) widget, i.e. a quiz. Second most common were conceptual widgets ($C = 8$), often combined with simulations ($S = 5$), i.e. the PhET simulations for physics. Information ($I = 5$), context ($C = 5$), and presentational widgets ($P = 3$) were used more rarely. For content, secondary education topics including non-tertiary post-secondary education (levels 2-5) in STEM fields were most common, but the humanities and languages were also represented.

7.3.4 Discussion

The results of our user study present first indicators that the concept of explorable authoring may be viable. In terms of usability, several areas like stimulation, usefulness and trust are very strong, considering ratings above 2 are very rare, and ratings above 1 are above average. This shows that participants consider the tool interesting and beneficial for real use cases. The rating for an overall impression of the tool, attractiveness, is also strong, which may indicate that the tool exceeds expectations compared to other solutions—although further investigation is needed to strengthen this interpretation. There are some notable weaknesses: Participants considered the tool somewhat unstable (dependability), did not always know how to configure it to their needs (adaptability), and had issues learning some concepts (perspicuity). These weaknesses are however to be expected for a research prototype, which is often less polished than commercial products. In terms of expressiveness, the broad range of workshop participants and explorables created show the potential of explorable authoring for all fields of education, but especially for STEM subjects.

However, looking at limitations, the sample size could be increased in further investigations to better support the claims made, and to get more reliable results on possible correlations. Due to the workshop setup, exposure to the concept of explorables and to the authoring tool was also very limited—long-term impressions and UX in real-world settings may differ. Also, EdTech is a vast field, and it is yet untested how explorables can work with other emerging technologies such as VR/AR, how to optimally integrate LA, what role LLMs can play in generating such content, and so on – this study can only speak to a subset of EdTech, not the whole field.

Overall, explorable authoring can contribute to a closer integration of the EdTech world—while there are many other dimensions to consider for cooperation between researchers, educators, and companies (e.g., organizational, legal, financial), technical challenges such as a lack of FAIRness in interactive learning resources form a barrier of entry to all kinds of cooperation. In other words: To increase cooperation, it can be helpful if there is a clear and easy technical path.

7.4 Summary: Explorable Authoring

In this chapter, we presented the concept of explorable authoring, an authoring paradigm for educators to create interactive learning resources according to the remix workflow.

We first (Section 7.1) defined interactive documents in a minimal fashion, based on the foundations laid previously. We then outlined the self editing paradigm (interactive elements that provide affordances to modify themselves inside the document) and integrated it with rich text editing in the overarching explorable authoring paradigm (rich text editing for static content and self editing for interactive content). We further defined the behavior of documents created with explorable authoring when read in browsers as explorable reading, a limited form of authoring as readers interact with the content.

Next (Section 7.2), we looked at three examples of the output of explorable authoring. The first example is a complete online textbook/course called `ExplorableJS` which features interactive code cells in the style of Jupyter and otherwise mostly static content. The last two examples are learning units for the local school lab InfoSphere aimed at students in grade 5 and 6. Both are blended learning scenarios where classroom instruction and activities are supplemented with explorables: The first unit (and second example) aims to teach the basics of cryptography to build a solid understanding so children are able to choose better passwords, for example. A cryptography widget is used so the learners can see how different cryptographic methods behave interactively. The second unit (and last example) is a kind of introduction to programming and algorithmic thinking – it mainly uses a block-based programming widget mixed with different quiz tasks, video and text content.

Finally (Section 7.3), we outlined the evaluation of our authoring, which is set up in a design parallel to the developer study. That involves the experimental setup (three hour workshops with educators), the data collection & analysis methods (UEQ+, QCA), and some inherent methodological limitations (no empirical measure of remixability, fit of experiment and theory, sampling issues, and biases of participants). We presented the results including demographic data (n=23 educators), and measurements (usability and expressiveness), also possible correlations. We discussed these results, stating that while broad expressiveness and very high usability are indicated, the sample's size limits conclusions.

RESEARCH SUMMARY

CT: In the context of a technical solution for web-based, interactive resources to support active learning, aimed at post-primary STEM learners:

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

ME: Create a prototype with participatory design. Evaluate it in experiments with authoring workshops and development projects, for each proving remixability theoretically and measuring usability (UEQ+) & expressiveness (QCA).

PR: *WebWriter*, a single page web app implemented with web technologies in the MVVM pattern, offering both an authoring and a development interface.

- **RQ_1**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
  - ◇ **HY_1**: Interactive elements are remixable if they are findable, accessible, interoperable, and reusable (FAIR), which is possible by applying the component and package patterns and adding a domain-specific API.
  - ◇ **PR_1**: *Widgets*, which extend Web Components with a minimal interface for blackbox reuse through authoring tools. They are published and accessed as Node Packages with learning-related metadata.
  - ◇ **EV_1**: Developer study over 4-6 months with n=11 participants creating widgets, with a post test consisting of interviews and survey. Results showed broad expressiveness and a positive developer experience, but the sample selected is limiting.

- **RQ_2**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?
  - ◇ **HY_2**: Authoring should support the remix workflow for educators by extending rich text editing with a new paradigm to cover morphological changes to interactive content.
  - ◇ **PR_2**: *Explorable Authoring*, which combines rich text editing, self editing and form fill-in to create web-based interactive documents.
  - ◇ **EV_2**: Authoring study with n=23 educators in three-hour workshops using WebWriter to create explorables. Results showed broad expressiveness and a very positive user experience, but the method and sample size are limiting factors.

CT=Context, RQ=Research Question, ME=Method, PR=Prototype, HY=Hypothesis, EV=Evaluation

Chapter 8 Conclusion

In this thesis, we set out to find a technical solution for the reusability paradox in the case of interactive learning resources (Chapter 1). We motivated the problem by showing the inherent contextuality of learning (behaviorism, cognitivism, and constructivism), which leads to a tension between reusability (desirable for cooperation between educators) and usefulness. We also suggested that any technical solution would need to consider both usability and expressiveness, which are difficult to reconcile (the danger of the "Turing Tar Pit"). For interactive content, we consider the standard approach, which is programming, highly expressive – but not usable for the average educator. This lead us to our two research questions, concerning an expressive and usable design pattern for interactive elements (RQ1), and an expressive and usable authoring paradigm for interactive learning resources (RQ2).

We explained a few foundations which helps define a scope for our work (Chapter 2). One part of the scope is educational technologies (EdTech), both a research field and a market producing a diverse set of insights and technologies – the object of what we want to make remixable. We looked at the history of EdTech, leading to a modern, pluralistic landscape constantly shifted by innovative technologies (the web, mobile computing, and LLMs). The other part of the scope is the web, the ubiquitous platform for networked documents and applications. The hypermodal nature of the web makes it an ideal technical fit for the remix of interactive learning resources, as both static and interactive content can be expressed.

We offered an insight into related work (Chapter 3), broken up into related ideas on design patterns for interactive elements (RQ1), authoring paradigms for interactive resources (RQ2), and explorable authoring systems which integrate both (`SCORM`, `MDX`, `H5P`, `Idyll`). This allowed us to formulate hypotheses: First (RQ1), that interactive elements could be remixable if they are FAIR, which is possible by combining the package and component design patterns. Second (RQ2), that remixability for educators means supporting the remix workflow by building on rich text editing.

For our methodology (Chapter 4), we outlined a research design applied in the rest of the project: We planned to develop a prototypical system which incorporates the ideas from both hypotheses, also considering results from the literature and a requirements study to assure a base level of usability and expressiveness. Discussing alternatives, we decided on an experimental setup using the developed prototype.

We showcased our prototype (Chapter 5), called `WebWriter`, by presenting the concept with an authoring interface intended for educators, and a development interface intended for EdTech developers. The authoring interface offers a WYSIWYG

experience with an editor, a palette with insertable elements, a toolbox with configuration options, an app bar, and a separate settings page. The development interface consists of the documentation and widget gallery on the WebWriter website, and the live package interface in the main application for development and debugging inside WebWriter. In terms of implementation, WebWriter is a SPA in the MVVM architecture, using TS, `Lit`, `MobX`, and `ProseMirror` as key technologies.

To answer the first research question (Chapter 6), we introduced the software design pattern of widgets, which combine the patterns of packages and components. We explained how widgets can be implemented on the web, using Node packages and web components. We then showed how reuse is possible in other code projects, on websites, or in authoring tools. We talked about how we evaluated the design pattern in development projects with student EdTech developers (n=11) across 5-7 months, where a diverse set of widgets was produced (large span of different educational levels, fields, and types of interactivity) and interviews and questionnaires showed a high degree of usability for the concept. We qualified the results by discussing several limitations, which are mainly connected to the quality and size of the sample.

For the second research question (Chapter 7), we introduced explorable authoring as a new authoring paradigm which extends rich text editing with the concept of self editing for interactive content (widgets). In self editing, widgets provide affordances to make changes to themselves or their content, and changes triggered by interactions are processed in the widget module, causing a new document state by modifying attributes or the nested content. This allows authoring tools to integrate interactive elements with the usual capabilities of selecting, copying/pasting, saving/loading, etc. The reader accessing the document can become a limited co-author, as they can use the same self editing capabilities. The paradigm fits the remix workflow observed in education, with educators becoming authors using the full power of authoring tools to craft a learning experience, and learners becoming readers which access that learning experience but also make it their own through interactions.

Research Contribution Our project produced several contributions to the EdTech field: Some contributions are small, such as WebWriter, a FAIR and open source prototype implementation now available to the research community. Uniquely, work on the widget concept lead to many interesting prototypes which are similarly FAIR and open source. Two exemplary widgets were published in papers, but more than 20 other widgets also follow the FAIR principles and are open source. They can be reused outside of WebWriter in other EdTech projects, on websites, or in future authoring tools. Of course, this text outlining the research process can also help researchers in several ways: The implementation can be an example for EdTech researchers pursuing similar projects, which is aided by the discussion of implementation challenges. The methodology would also be applicable for other explorable authoring systems, allowing for future work such as comparative studies between tools.

Mainly, we contribute the software design pattern of widgets and the explorable authoring paradigm to the admittedly narrow research area of remixability. Widgets are a novel solution to the reuse of interactive elements, which is widely applicable in the field of web-based EdTech. We consider the minimal nature of the concept as its

main strength: It is rooted in two well-established approaches (packages and components), and requires very little from EdTech developers both in learning the concept and in applying it. As such, many EdTech projects could be wrapped as widgets, increasing FAIRness and thus remixability without changing the core of the project – widgets are agnostic to architectures and even specific technologies. Explorable authoring offers a more usable, but still expressive way of authoring interactive learning resources. Whether it is an improvement over existing approaches such as markup languages (`MDX`), form fill-in (`H5P`), and limited rich text editing with markup (`Idyll`), remains to be investigated – but we have shown that it may be a promising alternative.

Limitations While we are convinced that our results are valuable, from a scientific point of view, our evaluation may not be sufficient to prove them.

First, we do not measure remixability directly, which would have to happen not at the point of developing/authoring the interactive element/resource, but at the point of reuse. We decided on our research design for practical reasons, as developing a design pattern (widgets), an authoring paradigm (explorable authoring) already provided a significant challenge. A further study would need an ecosystem of exchange between developers and educators building on widgets and explorable authoring.

Second, for both the author and the developer study, our sample is less than ideal. Our developers are not established practioners in the EdTech community (such as researchers or professional developers at EdTech companies), but students aspiring to develop EdTech. Also, the sample size of n=11 developers is sufficient for tentative conclusions, but more data points would be needed for accurate UEQ+ results and stronger evidence in terms of expressiveness. For the authoring study, the sample of n=23 educators is larger, but the margin of error is still big (for example, dependability of authoring explorables with WebWriter may rate at somewhere between 0 and 1, see Figure 7.10). Still, using these statistical methods, we can be cognizant of that margin of error when reviewing the results.

Third, our scope of web-based EdTech and post-primary STEM learners is a limitation of our results. We assume that similar results could be attained for non-STEM subjects, but our evaluation does not cover non-STEM subjects, as only a few such widgets were developed and few non-STEM educators participated in the workshops.

Future Work As both widgets and explorable authoring are novel, there are nearly endless possibilities of future work – we will focus on those closest at hand.

First and most straightforward, the authoring and developer study could be repeated with more participants to achieve smaller margins of error and confirm the results. A step further, efforts could be made to recruit EdTech researchers instead of EdTech students to capture the perspective of the actual target group better. Many further studies are also easy to imagine: A comparative study between the explorable authoring systems already presented in our related work (`MDX`, `H5P`, `Idyll`, and now `WebWriter`) would help confirm our assumption that explorable authoring is an improvement over existing authoring paradigms – we can only state that is fairly usable and expressive in absolute terms. Similarly, if a study was conducted with EdTech practitioners familiar with other component systems (e.g. `H5P`), this may allow us to

compare the new approach. In FAIRness of EdTech, a lot of work is currently being conducted, which may shed more light on existing practices beyond component systems.

Second, the remix workflow outlines the three layers of sources, resources, and instances – while explorable authoring in `WebWriter` supports the resource and instance layer very well, the layer of sources is left largely unaddressed. In the future, technical solutions for exchange platforms so educators can share their own resources as sources for others could be found (this is being explored with student developers [Tro25; Möh25]). Similarly, searching for sources presents an interesting problem, as both content intended for educational purposes may become a source, but also non-educational content may be a source. If exchange and search platforms are implemented as prototypes or `WebWriter` integrates existing solutions in that area, further studies may be possible which observe reuse directly.

Third, there are many more features to be implemented in `WebWriter` to ease authoring and support different use case. For example, collaborative features both for authors working together on learning resources, and for learning groups of educators and learners using an educational learning resource together. Work-in-progress features such as the LLM integration also need to be developed beyond the prototype stage.

Fourth, on the level of remixing, LLMs could even perform more complex remixing operations: Sources used by educators may take many different forms, with text, images, PDFs, web content, and now even widgets that could be included in a new learning resource. LLMs can help process the more complex types of sources such as PDFs, and also offer prompt-based combinations of source collections. We may imagine a type of digital blender or mixer which creates a single learning resource out of many sources. That new resource may then be adjust by expert educators and be shared at the push of a button, mirroring back results of student interactions in real time on a dashboard. As such, the whole remix workflow can be digitally supported, with the crucial step of mixing sources done automatically through LLMs.

We conclude that our work yielded some promising results on the theoretical and technical level, namely our widget and explorable authoring concepts and their implementation in `WebWriter`. On the empirical level, further work is needed to validate our results, as limitations are numerous. Still, this project may bring the community a step closer towards solving the reusability paradox, helping increase cooperation between researchers and educators, furthering education in general.

RESEARCH SUMMARY

CT: In the context of a technical solution for web-based, interactive resources to support active learning, aimed at post-primary STEM learners:

RQ: How can the reusability paradox be solved for interactive learning resources, considering the perspectives of educators and developers?

ME: Create a prototype with participatory design. Evaluate it in experiments with authoring workshops and development projects, for each proving remixability theoretically and measuring usability (UEQ+) & expressiveness (QCA).

PR: *WebWriter*, a single page web app implemented with web technologies in the MVVM pattern, offering both an authoring and a development interface.

- **RQ$_1$**: Which domain-specific design pattern, both usable and expressive, can be used to implement interactive elements remixable by educators?
  - ◇ **HY$_1$**: Interactive elements are remixable if they are findable, accessible, interoperable, and reusable (FAIR), which is possible by applying the component and package patterns and adding a domain-specific API.
  - ◇ **PR$_1$**: *Widgets*, which extend Web Components with a minimal interface for blackbox reuse through authoring tools. They are published and accessed as Node Packages with learning-related metadata.
  - ◇ **EV$_1$**: Developer study over 4-6 months with n=11 participants creating widgets, with a post test consisting of interviews and survey. Results showed broad expressiveness and a positive developer experience, but the sample selected is limiting.

- **RQ$_2$**: Which authoring paradigm can be used to remix interactive documents in a usable and expressive way?
  - ◇ **HY$_2$**: Authoring should support the remix workflow for educators by extending rich text editing with a new paradigm to cover morphological changes to interactive content.
  - ◇ **PR$_2$**: *Explorable Authoring*, which combines rich text editing, self editing and form fill-in to create web-based interactive documents.
  - ◇ **EV$_2$**: Authoring study with n=23 educators in three-hour workshops using WebWriter to create explorables. Results showed broad expressiveness and a very positive user experience, but the method and sample size are limiting factors.

$\Longrightarrow$ Despite some limitations and with possible future work, Widgets and Explorable Authoring integrated in an explorable authoring system such as WebWriter may help solve the reusability paradox for interactive learning resources.

CT=Context, RQ=Research Question, ME=Method, PR=Prototype, HY=Hypothesis, EV=Evaluation

Appendix

Appendix A Instruments and Data

A digital appendix including all research data is publicly available at this URL: https://osf.io/znrv6/overview?view_only=bcb9421b6ce04721a8086d38260ba4d6[1]

If the data should become unavailable or questions arise, please contact salmen@cs.rwth-aachen.de.

[1] For the review version, this is a read-only access link to a private data repository. The data will be released publicly when the dissertation is published and this link replaced.

Appendix B Own Publications

[DSS25] Andreas Dymek, Frederic Salmen, and Ulrik Schroeder. “Web-Based Prototype of a Branching Scenario Editor for Adaptive Learning”. In: *23. Fachtagung Bildungstechnologien (DELFI 2025)*. Gesellschaft für Informatik eV. 2025, pp. 395–400. doi: 10.18420/delfi2025_48.

[KSS24] Christian Koch, Frederic Salmen, and Ulrik Schroeder. “Web-based prototype of a visual and interactive deep learning simulation”. In: *Proceedings of DELFI 2024*. Gesellschaft für Informatik eV. 2024, pp. 499–504. doi: 10.18420/delfi2024_48.

[Löb+23] Alexander Löbel et al. “Behind the Screens: The Security & Privacy Advice Landscape of Children in Grades 5 & 6”. In: *Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research*. 2023, pp. 1–2. doi: 10.1145/3605468.3609766.

[Sal+24] Frederic Salmen et al. “FAIR Learning Technologies with Web Components and Packages”. In: *Proceedings of DELFI 2024*. Gesellschaft für Informatik eV, 2024, pp. 10–18420.

[Sal+25] Frederic Salmen et al. “Packaged Widgets: Evaluating a Concept for FAIR Educational Technology Components”. In: *Proceedings of the 22nd International Conference on Cognition and Exploratory Learning in Digital Age (CELDA 2025)*. 2025.

[Sal25a] Frederic Salmen. “Evaluating Authoring Tools with the Explorable Authoring Requirements”. In: *Nachwuchskonferenz der Didaktik der Informatik (NakoDI) 2022*. Gesellschaft für Informatik eV. 2025, pp. 39–54. doi: 10.18420/nakodi2022_08.

[SRS23a] Frederic Salmen, René Roepke, and Ulrik Schroeder. “WebWriter: A System to Author and Remix Explorables – Requirements & First Prototype”. In: *Workshops Der 21. Fachtagung Bildungstechnologien (DELFI)*. Gesellschaft für Informatik eV. 2023, pp. 39–48. doi: 10.18420/wsdelfi2023-13.

[SRS23b] Frederic Salmen, René Roepke, and Ulrik Schroeder. “WebWriter: A System to Author and Remix Explorables–Requirements & First Prototype”. In: *Workshops Der 21. Fachtagung Bildungstechnologien (DELFI)*. Gesellschaft für Informatik eV, 2023, pp. 39–48. doi: 10.18420/wsdelfi2023-13.

[SRS25] Frederic Salmen, René Roepke, and Ulrik Schroeder. “Explorables: Evaluating a Concept for Remixable Interactive Learning Resources”. In: *Proceedings of the 22nd International Conference on Cognition and Exploratory Learning in Digital Age (CELDA 2025)*. 2025.

Appendix C Supervised Theses

[Bau23] Ferdinand Baumgarten. “Design and Implementation of a Widget for Graph Algorithm Visualization”. MA thesis. Aachen: RWTH Aachen University, Apr. 2023.

[Bog25] Alexia Bogarin Weiss. “Implementation of a Word Puzzle Widget”. BA thesis. Aachen: RWTH Aachen University, Mar. 2025.

[Dao23] Thu Van Dao. “Development of a Web-Based Computer Network Widget for Educational Purposes”. BA thesis. Aachen: RWTH Aachen University, Mar. 2023.

[Dym24] Andreas Dymek. “Design and Implementation of a Branching Scenario Widget”. MA thesis. Aachen: RWTH Aachen University, Dec. 2024.

[Fel26] Eric Feltgen. “Integration generativer AI zur Vereinfachung der Erstellung von Explorables mit WebWriter”. vathesis. Aachen: RWTH Aachen University, 2026.

[Fer23] Din Ferizović. “Konzeption und Erprobung einer Lerneinheit zum Thema Authentifizierung und Passwortsicherheit”. MA thesis. Aachen: RWTH Aachen University, Oct. 2023.

[Gen24] Jorik van Genuchten. “Automata Simulation in the Context of Explorables”. MA thesis. Aachen: RWTH Aachen University, June 2024.

[Got25] Max Gottwald. “Implementation of a Slide Presentation Widget for WebWriter”. BA thesis. Aachen: RWTH Aachen University, Mar. 2025.

[Gro24] Florin Große. “Implementation eines Widgets für Geometrieaufgaben”. BA thesis. Aachen: RWTH Aachen University, Mar. 2024.

[Koc23] Christian Gabriel Simon Koch. “Development of a Web-Based Simulation to Teach Deep Learning Fundamentals”. BA thesis. Aachen: RWTH Aachen University, Sept. 2023.

[Kre25] Robin Jonas Kreft. “Planung und Entwicklung eines Widgets zur Visualisierung chemischer Prozesse bei Experi- menten”. MA thesis. Aachen: RWTH Aachen University, Jan. 2025.

[Kug23] Luis Kugel. “Entwicklung eines Widgets zur Datenvisualisierung und -exploration für interaktive Arbeitsblätter”. BA thesis. Aachen: RWTH Aachen University, Sept. 2023.

[Kul24] Alexander Kulibaba. “Development of an LLM-based GEC Tool for Explorable Authoring”. BA thesis. Aachen: RWTH Aachen University, Dec. 2024.

[Kus23] Jens Thorben Kuschel. “Interaktives Concreteness-Fading - Konzipierung eines Widgets zur Visualisierung von Analogien zu Grundlagen der Technischen Informatik in personalisierten Lernmedien”. BA thesis. Aachen: RWTH Aachen University, June 2023.

[Lan26] Moritz Langenhan. “Development of a WebWriter Widget for Visualizing Object-Oriented Concepts Using UML Diagrams”. BA thesis. Aachen: RWTH Aachen University, 2026.

[Len24] Philipp Lentzen. “Gestaltung und Implementierung eines Widgets für blockbasierte Programmierung in WebWriter”. BA thesis. Aachen: RWTH Aachen University, Sept. 2024.

[Lu23] Kevin Lu. “Entwicklung einer Webanwendung für digitale Arbeitsblätter zur Konstruktion und Übersetzung von Programmablaufplänen in natürliche Sprache”. MA thesis. Aachen: RWTH Aachen University, Aug. 2023.

[Mar26] Laura Marten. “Eine Fallstudie zur Nutzung von Explorables bei der Erstellung von Unterrichtsmaterial für blockbasierte Programmierung”. MA thesis. Aachen: RWTH Aachen University, 2026.

[Möh25] Tim Möhring. “Entwurf und Umsetzung einer Systemarchitektur für die WebWriter-Dienste”. BA thesis. Aachen: RWTH Aachen University, Nov. 2025.

[Ngu24] Phuong Anh Nguyen. “Implementation eines Widgets für interaktive Graphen und Data Moves”. BA thesis. Aachen: RWTH Aachen University, Sept. 2024.

[Now24] Jan Nowak. “Welche Anforderungen stellen Physik-Lehrkräfte an ein Tool zur Entwicklung digitaler Materialien?” MA thesis. Aachen: RWTH Aachen University, July 2024.

[Phu24] Vu Nhat Quang Phung. “Conception and Development of a Flashcard Widget”. MA thesis. Aachen: RWTH Aachen University, Feb. 2024.

[San26] Moritz Sander. “Entwicklung eines Widgets zur visuellen und interaktiven Erstellung von statischen Webseiten in personalisierten Lernmedien”. BA thesis. Aachen: RWTH Aachen University, 2026.

[Sch24] Lukas Schneider. “Implementation eines interaktiven Widgets zur Vermittlung von Logikgattern”. BA thesis. Aachen: RWTH Aachen University, Mar. 2024.

[Sch25] Matthias Horst Schäfer. “Five Steps to Deep Learning: Implementing Widgets for No-Code Deep Learning Simulation”. BA thesis. Aachen: RWTH Aachen University, Oct. 2025.

[She23] Ashish Shetty. “Design and Implementation of a Virtual Chemistry Lab Widget for Explorables”. MA thesis. Aachen: RWTH Aachen University, May 2023.

[The23] Raphael Thelen. “Entwicklung eines Widgets für chemische Zeichnungen”. BA thesis. Aachen: RWTH Aachen University, Dec. 2023.

[Tro25] Marc Troll. “Development of an Exchange Platform for WebWriter-Explorables”. MA thesis. Aachen: RWTH Aachen University, Nov. 2025.

[Vie22] Erik Viere. “Entwicklung eines webbasierten Code-Widgets zur Vermittlung von Programmierkenntnissen anhand verschiedener Aufgabentypen”. BA thesis. Aachen: RWTH Aachen University, Dec. 2022.

[Vur24] Tarik Vural. “Design and Implementation of an Interactive Video Widget in WebWriter”. BA thesis. Aachen: RWTH Aachen University, Aug. 2024.

[Zim25] Nura Zimmermann. “Design and Implementation of an Interactive Timeline Widget for WebWriter”. BA thesis. Aachen: RWTH Aachen University, Mar. 2025.

Appendix D Software References

[Ado02] *Adobe Captivate*. Adobe Inc., 2002. https://adobe.com/captivate/ (accessed on 12/13/2025).

[AH11] Luis von Ahn and Severin Hacker. *Duolingo*. Duolingo, Inc., 2011. https://www.duolingo.com/ (accessed on 12/13/2025).

[Aku12] Dmitriy Akulov. *JSDelivr*. 2012. https://jsdelivr.com/ (accessed on 12/13/2025).

[App08] *App Store*. Apple, Inc., 2008. https://appstore.com/ (accessed on 12/13/2025).

[Art16] *Articulate 360*. Articulate Inc., 2016. https://articulate.com/360/ (accessed on 12/13/2025).

[Byt16] *TikTok*. ByteDance, 2016. https://www.tiktok.com/ (accessed on 12/13/2025).

[CCP16] Alexandre Chopin, Sebastien Chopin, and Parsa Pooya. *Nuxt*. 2016. https://nuxt.com/ (accessed on 12/13/2025).

[DAH03] Chris DeWolfe, Tom Anderson, and Jon Hart. *Myspace*. Viant Technology LLC, 2003. https://www.myspace.com/ (accessed on 12/13/2025).

[Doc13] *Docker*. Docker, Inc., 2013. https://www.docker.com/ (accessed on 08/18/2025).

[Dor+06] Jack Dorsey et al. *X*. Ed. by X Corp. 2006. https://www.x.com/ (accessed on 12/13/2025).

[Dou02] Martin Dougiamas. *Moodle*. Moodle HQ/Community, 2002. https://moodle.org/ (accessed on 12/13/2025).

[EPI95] *UnrealEngine*. EPIC Games, 1995. https://www.unrealengine.com/de/home (accessed on 08/19/2025).

[Fan17] Justin Fangnani. *Lit*. OpenJS Foundation, 2017. https://www.lit.dev/ (accessed on 12/13/2025).

[Fig16] *Figma*. Figma Inc., 2016. https://www.figma.com/ (accessed on 12/13/2025).

[Fra+12] Neil Fraser et al. *Blockly*. Google, 2012. https://developers.google.com/blockly (accessed on 12/13/2025).

[Fra+16] Max Franz et al. *Cytoscape.js*. H5P Group, 2016. https://js.cytoscape.org/ (accessed on 12/13/2025).

[Fra13] Steve Francia. *Hugo*. 2013. https://gohugo.io/ (accessed on 12/13/2025).

[Geo23] Gani Georgiev. *PocketBase*. 2023. https://pocketbase.io/ (accessed on 12/13/2025).

[Goo06] *Google Workspace*. Google, 2006. https://workspace.google.com/ (accessed on 12/13/2025).

[Goo08a] *Google Chrome*. Google, 2008. https://google.com/chrome (accessed on 12/13/2025).

[Goo08b] *Play Store*. Google, 2008. https://play.google.com/ (accessed on 12/13/2025).

[Goo16] *Angular*. Google, 2016. https://angular.dev/ (accessed on 08/18/2025).

[Goo18] *Tensorflow.js*. Google, 2018. https://www.tensorflow.org/js (accessed on 12/13/2025).

[H5P13] *H5P*. H5P Group, 2013. https://h5p.org/ (accessed on 12/13/2025).

[H5P19] *H5P Branching Scenario*. H5P Group, 2019. https://h5p.org/branching-scenario (accessed on 12/13/2025).

[Har+16] Rich Harris et al. *Svelte*. 2016. https://www.svelte.dev/ (accessed on 12/13/2025).

[Hav11] Marijn Haverbeke. *CodeMirror*. 2011. https://codemirror.net/ (accessed on 12/13/2025).

[Hav17] Marijn Haverbeke. *ProseMirror*. 2017. https://prosemirror.net/ (accessed on 12/13/2025).

[Hil08] Raymond Hill. *uBlock Origin*. 2008. https://google.com/chrome (accessed on 12/13/2025).

[Hip00] Richard Hipp. *SQLite*. 2000. https://sqlite.org/ (accessed on 12/13/2025).

[HKC05] Chad Hurley, Jawed Karim, and Steve Chen. *YouTube*. Google, 2005. https://www.youtube.com/ (accessed on 12/13/2025).

[Hoh01] Markus Hohenwarter. *GeoGebra*. 2001. https://geogebra.org/ (accessed on 12/13/2025).

[Hol10] TJ Holowaychuk. *Express.js*. OpenJS Foundation, 2010. https://expressjs.com/ (accessed on 12/13/2025).

[Jup14] *Project Jupyter*. Jupyter, 2014. https://jupyter.org (accessed on 04/28/2022).

[Kha08] Salman Khan. *Khan Academy*. Khan Academy, Inc., 2008. https://www.khanacademy.org/ (accessed on 12/13/2025).

[Kut17] Jarosław Kutyłowski. *DeepL Translator*. 2017. https://deepl.com/ (accessed on 12/13/2025).

[Lea10] *Gomo Learning*. Learning Technologies Group, 2010. https://www.gomolearning.com/ (accessed on 12/13/2025).

[Lea13] *Elucidat*. Learning Pool, 2013. https://www.elucidat.com/ (accessed on 12/13/2025).

[LM03] Mike Little and Matt Mullenweg. *WordPress*. WordPress Foundation, 2003. https://wordpress.com/ (accessed on 12/13/2025).

[Met13] *React*. Meta, 2013. https://react.dev/ (accessed on 08/18/2025).

[MF17] Philipp Mayring and Thomas Fenzl. *QCAmap*. 2017. https://www.qcamap.org/ (accessed on 12/13/2025).

[Mic08] *Microsoft Azure*. Microsoft, 2008. https://azure.microsoft.com/ (accessed on 12/13/2025).

[Mic12] *Microsoft Store*. Microsoft, 2012. https://apps.microsoft.com/ (accessed on 12/13/2025).

[Mic20] *Microsoft Edge*. Microsoft, 2020. https://microsoft.com/edge (accessed on 12/13/2025).

[Mic21] *Github Copilot*. Microsoft, 2021. https://github.com/features/copilot (accessed on 12/13/2025).

[Mic90] *Microsoft Office*. Microsoft, 1990. https://www.office.com/ (accessed on 12/13/2025).

[Nis16] Maël Nison. *Yarn*. 2016. https://yarnkg.com/ (accessed on 12/13/2025).

[Ope06] *jQuery*. OpenJS Foundation, 2006. https://jquery.com/ (accessed on 08/18/2025).

[Ope13] *Electron*. OpenJS Foundation, 2013. https://electronjs.org/ (accessed on 12/13/2025).

[Ope21] *Custom Elements Manifest Analyzer*. Open WC, 2021. https://www.npmjs.com/package/@custom-elements-manifest/analyzer (accessed on 12/13/2025).

[Ope22] *ChatGPT*. OpenAI, 2022. https://chatgpt.com/ (accessed on 12/13/2025).

[Pic17] Juan Picado. *Verdaccio*. 2017. https://www.verdaccio.org/ (accessed on 12/13/2025).

[Pos96] *PostgreSQL*. PostgrSQL Global Development Group, 1996. https://www.postgresql.org/ (accessed on 08/18/2025).

[Pre+08a] Tom Preston-Werner et al. *GitHub*. Microsoft, 2008. https://github.com/ (accessed on 12/13/2025).

[Pre+08b] Tom Preston-Werner et al. *Jekyll*. 2008. https://jekyllrb.com/ (accessed on 12/13/2025).

[Rau+16] Guillermo Rauch et al. *NextJS*. Vercel, 2016. https://nextjs.org/ (accessed on 12/13/2025).

[Red19] *Podman*. Red Hat, Inc., 2019. https://podman.io/ (accessed on 08/18/2025).

[Ron10] Armin Ronacher. *Flask*. 2010. https://flask.palletsprojects.com/en/stable/ (accessed on 08/18/2025).

[Sal25b] Frederic Salmen. *WebWriter*. 2025. https://webwriter.app/ (accessed on 12/13/2025).

[Sal25c] Frederic Salmen. *WebWriter Build Tool*. 2025. https://www.npmjs.com/package/@webwriter/build (accessed on 12/13/2025).

[Sal25d] Frederic Salmen. *WebWriter Lit Helper*. 2025. https://www.npmjs.com/package/@webwriter/lit (accessed on 12/13/2025).

[SLL09] Alex Shevchenko, Max Lytvyn, and Dmytro Lider. *Grammarly*. Superhuman Platform Inc., 2009. https://grammarly.com/ (accessed on 12/13/2025).

[SM21] Fred Schott and Nate Moore. *Astro*. 2021. https://astro.build/ (accessed on 12/13/2025).

[Sol20] Jero Soler. *Drawflow*. 2020. https://jerosoler.github.io/Drawflow/ (accessed on 12/13/2025).

[SR20] Jan Philip Schellenberg and Sebastian Rettig. *Lumi*. 2020. https://lumi.education/ (accessed on 12/13/2025).

[Sum21] Jarred Sumner. *Bun*. 2021. https://bun.com/ (accessed on 12/13/2025).

[Sys04] Igor Sysoev. *NGINX*. 2004. https://nginx.org/ (accessed on 08/18/2025).

[TN20] Daniel Thompson-Yvetot and Lucas Nogueira. *Tauri*. 2020. https://tauri.app/ (accessed on 12/13/2025).

[Uni05] *Unity Engine*. Unity Technologies, 2005. https://unity.com (accessed on 08/19/2025).

[Ver17] *MDX*. Vercel, 2017. https://mdxjs.com/ (accessed on 12/13/2025).

[Wal13] Jordan Walke. *React*. Facebook, 2013. https://www.react.dev/ (accessed on 12/13/2025).

[Wal20] Evan Wallace. *esbuild*. 2020. https://esbuild.github.io/ (accessed on 12/13/2025).

[Wan08] Alf Inge Wang. *Kahoot*. Kahoot ASA, 2008. https://www.kahoot.com/ (accessed on 12/13/2025).

[Wes15] Michel Weststrate. *MobX*. 2015. https://mobx.js.org/ (accessed on 12/13/2025).

[Xia16] Lunny Xiao. *Gitea*. 2016. https://gitea.com/ (accessed on 12/13/2025).

[You14] Evan You. *Vue.js*. 2014. https://vuejs.org/ (accessed on 08/18/2025).

[You20] Evan You. *Vite*. VoidZero Inc., 2020. https://www.vite.dev/ (accessed on 12/13/2025).

[ZS11] Dmytro Zaporozhets and Sytse Sijbrandij. *GitLab*. GitLab Inc., 2011. https://gitlab.com/ (accessed on 12/13/2025).

[Zuc+04] Mark Zuckerberg et al. *Facebook*. Facebook, 2004. https://www.facebook.com/ (accessed on 12/13/2025).

Appendix E Literature References

[Abb18] Janet Abbate. “Popularizing the Internet”. In: *Communication in History*. 7th ed. Routledge, 2018.

[AFQ89] J. André, R. Furuta, and V. Quint. *Structured Documents*. Cambridge: Cambridge University Press, Apr. 1989.

[Ale77] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[Ale79] Christopher Alexander. *The timeless way of building*. Vol. 1. New york: Oxford university press, 1979.

[And11] Marc Andreessen. “Why software is eating the world”. In: *Wall Street Journal* 20.2011 (2011), p. C2.

[AS12] Terry Anderson and Julie Shattuck. “Design-Based Research: A Decade of Progress in Education Research?” In: *Educational Researcher* 41.1 (Jan. 2012), pp. 16–25. doi: 10.3102/0013189X11428813.

[AS22] Vibhor Agarwal and Nishanth Sastry. ““Way back then”: A Data-driven View of 25+ years of Web Evolution”. In: *Proceedings of the ACM Web Conference 2022*. Virtual Event, Lyon France: ACM, Apr. 2022, pp. 3471–3479. doi: 10.1145/3485447.3512283.

[ASR02] Pekka Abrahamsson, Outi Salo, and Jussi Ronkainen. “Agile Software Development Methods: Review and Analysis”. In: (2002).

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Berlin, Heidelberg: Springer, Feb. 2003.

[Aßm05] Uwe Aßmann. “Architectural styles for active documents”. In: *Science of Computer Programming*. New Software Composition Concepts 56.1 (Apr. 2005), pp. 79–98. doi: 10.1016/j.scico.2004.11.006.

[AV06] Frans van Assche and Riina Vuorikari. “A framework for quality of learning resources”. In: *Handbook on Quality and Standardisation in E-Learning*. Ed. by Ulf-Daniel Ehlers and Jan Martin Pawlowski. Berlin, Heidelberg: Springer, 2006, pp. 443–456. doi: 10.1007/3-540-32788-6_29.

[Bab+23] Parvaneh Babari et al. “A literature review of children’s and youth’s conceptions of the internet”. In: *International Journal of Child-Computer Interaction* 37 (2023). doi: 10.1016/j.ijcci.2023.100595.

[Bad00] Alan Baddeley. “The episodic buffer: a new component of working memory?” In: *Trends in cognitive sciences* 4.11 (2000), pp. 417–423.

[Bad92] A. Baddeley. “Working memory”. In: *Science (New York, N.Y.)* 255.5044 (Jan. 1992), pp. 556–559. doi: 10.1126/science.1736359.

[Bar67] John Barson. *Instructional Systems Development – A Demonstration And Evaluation Project – Final Report*. Tech. rep. June 1967.

[Bea18] Tita Beaven. “‘Dark reuse’: an empirical study of teachers’ OER engagement”. In: *Open Praxis* 10.4 (2018), pp. 377–391.

[Ber23] Tim Berners-Lee. *Frequently asked questions by the Press*. 2023. https://www.w3.org/People/Berners-Lee/FAQ.html#Influences (accessed on 08/09/2025).

[Ber89] Tim Berners-Lee. *Information management: A proposal*. Tech. rep. 1989.

[Ber91] Tim Berners-Lee. *Tags used in HTML*. 1991. https://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html (accessed on 08/09/2025).

[BLE65] Donald L. Bitzer, Elisabeth R. Lyman, and John A. Easley Jr. “The uses of PLATO: A computer-controlled teaching system”. In: *Coordinated Science Laboratory Report no. R-268* (1965).

[Blo+64] Benjamin Samuel Bloom et al. *Taxonomy of educational objectives*. Vol. 2. Longmans, Green New York, 1964.

[Bor93] Jorge Luis Borges. *El jardín de senderos que se bifurcan*. Barcelona: Bibliotex, S.L., 1993.

[BR89] T. J. Biggerstaff and C. Richter. “Reusability framework, assessment, and directions”. In: *Software reusability: vol. 1, concepts and models*. New York, NY, USA: Association for Computing Machinery, 1989, pp. 1–17.

[Bra+75] Robert K. Branson et al. “Interservice procedures for instructional systems development. executive summary and model”. In: (1975).

[Bra16] Cynthia J. Brame. “Effective Educational Videos: Principles and Guidelines for Maximizing Student Learning from Video Content”. In: *CBE—Life Sciences Education* 15.4 (Dec. 2016). Ed. by Kathryn E. Perez, es6. doi: 10.1187/cbe.16-03-0125.

[Bro+21] Jessica E. Brodsky et al. ““It’s just everywhere now”: Middle-school and college students’ mental models of the Internet”. In: *Human Behavior and Emerging Technologies* (2021). doi: 10.1002/hbe2.281.

[Bro+23] Cyril Brom et al. “‘The internet is in the satellites!’: A systematic review of 3–15-year-olds’ conceptions about the internet”. In: *Education and Information Technologies* 28 (2023). doi: 10.1007/s10639-023-11775-9.

[Bro92] Ann L. Brown. “Design Experiments: Theoretical and Methodological Challenges in Creating Complex Interventions in Classroom Settings”. In: *Journal of the Learning Sciences* 2.2 (Apr. 1992), pp. 141–178. doi: 10.1207/s15327809jls0202_2.

[Bro96] John Brooke. “SUS: A ’Quick and Dirty’ Usability Scale”. In: *Usability Evaluation In Industry*. CRC Press, 1996.

[BT90] Dennis Bowers and Chia Tsai. “HyperCard in educational research: An introduction and case study”. In: *Educational Technology* 30.2 (1990), pp. 19–24.

[BTD16] Stefan Buschmann, Matthias Trapp, and Jürgen Döllner. “Animated visualization of spatial–temporal trajectory data for air-traffic analysis”. In: *The Visual Computer* 32 (Mar. 2016), pp. 371–381. doi: 10.1007/s00371-015-1185-9.

[Bur24] Brendan Burns. *Designing distributed systems*. O’Reilly Media, Inc., 2024.

[Bus45] Vannevar Bush. “As we may think”. In: *The Atlantic Monthly* 176.1 (1945), pp. 101–108.

[BW21] Eva Wennås Brante and Robert Walldén. ““Internet? That’s an app you can download”. First-graders use linguistic resources to describe internet and digital information”. In: *Education Inquiry* 14 (2021). doi: 10.1080/20004508.2021.1950273.

[BZF05] S. Boyd, D. Zowghi, and A. Farroukh. “Measuring the expressiveness of a constrained natural language: an empirical study”. In: *13th IEEE International Conference on Requirements Engineering (RE’05)*. Aug. 2005, pp. 339–349. doi: 10.1109/RE.2005.39.

[Car+20] Michelle Carney et al. “Teachable Machine: Approachable Web-Based Tool for Exploring Machine Learning Classification”. In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. Honolulu HI USA: ACM, Apr. 2020, pp. 1–8. doi: 10.1145/3334480.3382839.

[CCN97] Gaëlle Calvary, Joëlle Coutaz, and Laurence Nigay. “From single-user architectural design to PAC*: a generic software architecture model for CSCW”. In: *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. Atlanta Georgia USA: ACM, Mar. 1997, pp. 242–249. doi: 10.1145/258549.258717.

[CH18] Matthew Conlen and Jeffrey Heer. “Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web”. In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. Berlin Germany: ACM, Oct. 2018, pp. 977–989. doi: 10.1145/3242587.3242600.

[Che+24] Xiangping Chen et al. “An empirical study of code reuse between GitHub and stack overflow during software development”. In: *Journal of Systems and Software* 210 (2024), p. 111964.

[Chu+22] Neil P. Chue Hong et al. “FAIR principles for research software (FAIR4RS principles)”. In: *Zenodo* (2022).

[Chu07] Daniel Churchill. “Towards a useful classification of learning objects”. In: *Educational Technology Research and Development* 55.5 (Sept. 2007), pp. 479–497. doi: 10.1007/s11423-006-9000-y.

[CJB04] Allan Collins, Diana Joseph, and Katerine Bielaczyc. “Design Research: Theoretical and Methodological Issues”. In: *Journal of the Learning Sciences* 13.1 (Jan. 2004), pp. 15–42. doi: 10.1207/s15327809jls1301_2.

[CK24] Will Crichton and Shriram Krishnamurthi. “A Core Calculus for Documents: Or, Lambda: The Ultimate Document”. In: *Proceedings of the ACM on Programming Languages* 8.POPL (Jan. 2024), pp. 667–694. doi: 10.1145/3632865.

[Cla90] Jon Claerbout. “Active documents and reproducible results”. In: *Stanford Exploration Project Report* 67 (1990), pp. 139–144.

[CMM17] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research Methods in Education*. 8th ed. London: Routledge, Oct. 2017. doi: 10.4324/9781315456539.

[CN23] Sridhar Chimalakonda and Kesav V. Nori. “A patterns based approach for the design of educational technologies”. In: *Interactive Learning Environments* 31.4 (May 2023), pp. 2114–2133. doi: 10.1080/10494820.2021.1875000.

[Col92] Allan Collins. “Toward a Design Science of Education”. In: *New Directions in Educational Technology*. Ed. by Eileen Scanlon and Tim O’Shea. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 15–22. doi: 10.1007/978-3-642-77750-9_2.

[Con+21] Matthew Conlen et al. “Idyll Studio: A Structured Editor for Authoring Interactive & Data-Driven Articles”. In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. Virtual Event USA: ACM, Oct. 2021, pp. 1–12. doi: 10.1145/3472749.3474731.

[Con21] Matthew Conlen. *Authoring and Publishing Interactive Articles*. University of Washington, 2021.

[CP12] K.i. Clements and J.m. Pawlowski. “User-oriented quality for OER: understanding teachers’ views on re-use, quality, and trust”. In: *Journal of Computer Assisted Learning* 28.1 (2012), pp. 4–14. doi: 10.1111/j.1365-2729.2011.00450.x.

[CS59] Noam Chomsky and Marcel P. Schützenberger. “The algebraic theory of context-free languages”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 1959, pp. 118–161.

[CW98] Reidar Conradi and Bernhard Westfechtel. “Version models for software configuration management”. In: *ACM Computing Surveys* 30.2 (June 1998), pp. 232–282. doi: 10.1145/280277.280280.

[Die+19] Jens Dietrich et al. “Dependency Versioning in the Wild”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. May 2019, pp. 349–359. doi: 10.1109/MSR.2019.00061.

[Die19] André Dietrich. *LiaScript: A Domain-Specific-Language for Interactive Online Courses*. Tech. rep. Publication Title: International Association for Development of the Information Society ERIC Number: ED621589. International Association for the Development of the Information Society, 2019.

[Dij82] Edsger W. Dijkstra. “On the Role of Scientific Thought”. In: *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer New York, 1982, pp. 60–66. doi: 10.1007/978-1-4612-5695-3_12.

[Dod01] Philip Dodds. *SCORM 1.2*. 2001.

[EK19] Sirpa Eskelä-Haapanen and Carita Kiili. “‘It Goes Around the World’ – Children’s Understanding of the Internet”. In: *Nordic Journal of Digital Literacy* 14 (2019). doi: 10.18261/issn.1891-943x-2019-03-04-07.

[Ell68] Douglas G. Ellson. “A Report on Programed Tutoring.” In: (1968).

[FB18] Eckhardt Fuchs and Annekatrin Bock, eds. *The Palgrave Handbook of Textbook Studies*. New York: Palgrave Macmillan US, 2018. doi: 10.1057/978-1-137-53142-1.

[Fel91] Matthias Felleisen. “On the expressive power of programming languages”. In: *Science of Computer Programming* 17.1 (Dec. 1991), pp. 35–75. doi: 10.1016/0167-6423(91)90036-W.

[FG95] Claire M. Fletcher-Flinn and Breon Gravatt. “The efficacy of computer assisted instruction (CAI): A meta-analysis”. In: *Journal of educational computing research* 12.3 (1995), pp. 219–241.

[Fin+05] Noah D. Finkelstein et al. “When Learning About the Real World Is Better Done Virtually: a Study of Substituting Computer Simulations for Laboratory Equipment”. In: *Phys. Rev. ST Phys. Educ. Res.* 1 (1 2005), p. 010103. doi: 10.1103/PhysRevSTPER.1.010103.

[FM12] Fabian Fagerholm and Jürgen Münch. “Developer experience: Concept and definition”. In: *2012 international conference on software and system process (ICSSP)*. IEEE, 2012, pp. 73–77.

[Fou25] OpenJS Foundation. *Modules: Packages | Node.js v24.5.0 Documentation*. 2025. https://nodejs.org/api/packages.html (accessed on 08/10/2025).

[Fow20] Simon Fowler. *Model-View-Update-Communicate: Session Types meet the Elm Architecture*. May 2020. doi: 10.48550/arXiv.1910.11108.

[Fre+14] Scott Freeman et al. “Active learning increases student performance in science, engineering, and mathematics”. In: *Proceedings of the National Academy of Sciences* 111.23 (June 2014), pp. 8410–8415. doi: 10.1073/pnas.1319030111.

[Fri04] Norm Friesen. “Three objections to learning objects”. In: *Online education using learning objects*. Routledge, 2004, pp. 78–88.

[FSB25] Mario Fuksa, Sandro Speth, and Steffen Becker. “MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern”. In: *Enterprise Design, Operations, and Computing*. Ed. by José Borbinha et al. Vol. 15409. Cham: Springer Nature Switzerland, 2025, pp. 163–181. doi: 10.1007/978-3-031-78338-8_9.

[Fur89] Richard Furuta. “Concepts and models for structured documents”. In: *Structured documents* 2 (1989), p. 7.

[Gam+95] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[Gar18] Marc Garreau. *Redux in action*. Simon and Schuster, 2018.

[Ges08] “Bildungsstandards Informatik - Sekundarstufe I”. In: (2008). Ed. by Gesellschaft für Informatik, p. 72.

[GHC18] Daniel Green, Charlie Hargood, and Fred Charles. “Contemporary Issues in Interactive Storytelling Authoring Systems”. In: *Interactive Storytelling*. Springer International Publishing, 2018, pp. 501–513. doi: 10.1007/978-3-030-04028-4_59.

[GHC21] Daniel Green, Charlie Hargood, and Fred Charles. “Use of Tools: UX Principles for Interactive Narrative Authoring Tools”. In: *J. Comput. Cult. Herit.* 14.3 (2021). doi: 10.1145/3458769.

[Gra18] Julie Hooft Graafland. “New technologies and 21st century children: Recent trends and outcomes”. In: *OECD Education Working Papers* 179 (2018). doi: 10.1787/e071a505-en.

[Gur08] Seth Gurell. *Open Educational Resources for Educators Version 1.0*. The Center for Open and Sustainable Learning, Aug. 2008.

[Gut30] E. R. Guthrie. “Conditioning as a principle of learning”. In: *Psychological Review* 37.5 (1930), pp. 412–428. doi: 10.1037/h0072172.

[Han+19] Tereza Hannemann et al. “Eight-Year-Olds’ Conceptions of Computer Viruses: A Quantitative Study”. In: *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*. WiPSCE. 2019. doi: 10.1145/3361721.3361726.

[Hav19] Marijn Haverbeke. *Eloquent JavaScript: a modern introduction to programming*. Third edition. San Francisco: No Starch Press, 2019.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. “A universal modular actor formalism for artificial intelligence”. In: *Advance papers of the conference*. Vol. 3. Stanford Research Institute Menlo Park, CA, 1973, p. 235.

[Hea16] Michael R. Heafford. *Pestalozzi: His thought and its relevance today*. Routledge, 2016.

[Hen+07] Jakob Henriksson et al. “How dark should a component black box be? The Reuseware Answer”. In: *Proc. of the 12th International Workshop on Component-Oriented Programming (WCOP) co-located with 21st European Conf. on Object-Oriented Programming (ECOOP’07)(to appear)*. 2007.

[Hin+19] Andreas Hinderks et al. “Developing a UX KPI based on the user experience questionnaire”. In: *Computer Standards & Interfaces* 65 (July 2019), pp. 38–44. doi: 10.1016/j.csi.2019.01.007.

[Hin18] Eric S. Hintz. *The Mother of All Demos | Lemelson*. 2018.

[Hoh+20] Fred Hohman et al. “Communicating with interactive articles”. In: *Distill* 5.9 (2020), e28.

[ISO18] ISO. *ISO 9241-11:2018*. Standard. 2018.

[ISO23] ISO. *ISO 25010:2023*. Standard. 2023.

[Jam97] William James. *The Meaning of Truth*. Amherst, N.Y.: Prometheus Books, 1997.

[JC24] Tarosh Jacob and Stephanie Centofanti. “Effectiveness of H5P in improving student learning outcomes in an online tertiary education setting”. In: *Journal of Computing in Higher Education* 36.2 (Aug. 2024), pp. 469–485. doi: 10.1007/s12528-023-09361-6.

[Jes09] Paul Jesukiewicz. *SCORM 2004 (4th Edition)*. 2009.

[Jol89] Vania Joloboff. “Document Represenation: Concepts and standards”. In: *Structured documents* 2 (1989), p. 75.

[Kau+17] Manfred Kaul et al. “Game Changer for Online Learning Driven by Advances in Web Technology.” In: *International Association for Development of the Information Society* (2017).

[Kel74] Fred S. Keller. “Ten Years of Personalized Instruction”. In: *Teaching of Psychology* 1.1 (Oct. 1974), pp. 4–9. doi: 10.1177/009862837400100102.

[Ken12] Karsten Kenklies. “Educational Theory as Topological Rhetoric: The Concepts of Pedagogy of Johann Friedrich Herbart and Friedrich Schleiermacher”. In: *Studies in Philosophy and Education* 31.3 (May 2012), pp. 265–273. doi: 10.1007/s11217-012-9287-6.

[Ker95] N. Kerth. “Caterpillar’s fate: A pattern language for transformation from analysis to design”. In: *Pattern Languages of Program Design. Addison-Wesley* (1995).

[KF98] Shriram Krishnamurthi and Matthias Felleisen. “Toward a formal theory of extensible software”. In: *SIGSOFT Softw. Eng. Notes* 23.6 (Nov. 1998), pp. 88–98. doi: 10.1145/291252.288269.

[Kle+12] Sebastian Kleinschmager et al. “Do static type systems improve the maintainability of software systems? An empirical study”. In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. Passau, Germany: IEEE, June 2012, pp. 153–162. doi: 10.1109/ICPC.2012.6240483.

[Knu78] Donald E. Knuth. *Tau Epsilon Chi: A System for Technical Text.* Computer Science Department Report STAN-CS-78-675.1. Stanford, 1978.

[Knu84a] Donald E. Knuth. *The TeXbook*. Reading, Mass.: Addison-Wesley Professional, Jan. 1984.

[Knu84b] Donald Ervin Knuth. “Literate programming”. In: *The computer journal* 27.2 (1984), pp. 97–111.

[Koe18] Hartmut Koenitz. “Thoughts on a Discipline for the Study of Interactive Digital Narratives”. In: *Interactive Storytelling*. Cham: Springer International Publishing, 2018, pp. 36–49. doi: 10.1007/978-3-030-04028-4_3.

[Kra02] David R. Krathwohl. “A Revision of Bloom’s Taxonomy: An Overview”. In: *Theory Into Practice* 41.4 (Nov. 2002), pp. 212–218. doi: 10.1207/s15430421tip4104_2.

[Kru92] Charles W. Krueger. “Software reuse”. In: *ACM Computing Surveys* 24.2 (June 1992), pp. 131–183. doi: 10.1145/130844.130856.

[KS22] Natalie Kiesler and Daniel Schiffner. “On the Lack of Recognition of Software Artifacts and IT Infrastructure in Educational Technology Research”. In: Gesellschaft für Informatik e.V., 2022, pp. 201–206.

[Kub62] George Kubler. *Shape of Time*. New Haven: Yale University Press, 1962.

[Lam94] Leslie Lamport. *LATEX: a document preparation system: user’s guide and reference manual*. 2nd ed. Reading, Mass: Addison-Wesley Pub. Co, 1994.

[LC62] J. C. R. Licklider and Welden E. Clark. “On-line man-computer communication”. In: *Proceedings of the May 1-3, 1962, spring joint computer conference on - AIEE-IRE ’62 (Spring)*. San Francisco, California: ACM Press, 1962, p. 113. doi: 10.1145/1460833.1460847.

[Leh18] Interkantonale Lehrmittelzentrale. “Lehrmittel in einer digitalen Welt”. In: (2018).

[Lei+09] Barry M. Leiner et al. “A brief history of the internet”. In: *ACM SIGCOMM Computer Communication Review* 39.5 (Oct. 2009), pp. 22–31. doi: 10.1145/1629607.1629613.

[Lew02] James R. Lewis. “Psychometric Evaluation of the PSSUQ Using Data from Five Years of Usability Studies”. In: *International Journal of Human-Computer Interaction* 14.3-4 (Sept. 2002), pp. 463–488. doi: 10.1080/10447318.2002.9669130.

[Lew95] James R. Lewis. “IBM computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use”. In: *International Journal of Human-Computer Interaction* 7.1 (Jan. 1995), pp. 57–78. doi: 10.1080/10447319509526110.

[LHS08] Bettina Laugwitz, Theo Held, and Martin Schrepp. “Construction and Evaluation of a User Experience Questionnaire”. In: *HCI and Usability for Education and Work*. Ed. by Andreas Holzinger. Vol. 5298. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 63–76. doi: 10.1007/978-3-540-89350-9_6.

[Lin92] Ogden R. Lindsley. “Precision teaching: Discoveries and effects”. In: *Journal of Applied Behavior Analysis* 25.1 (1992), p. 51.

[LM09] J. Loeliger and M. McCullough. “Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development”. In: May 2009.

[LM20] Anke Lindmeier and Andreas Mühling. “Keeping Secrets: K-12 Students’ Understanding of Cryptography”. In: *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. 2020. doi: 10.1145/3421590.3421630.

[LP21] Heeyoung Lee and Younghwan Pan. “Evaluation of the Nomological Validity of Cognitive, Emotional, and Behavioral Factors for the Measurement of Developer Experience”. In: *Applied Sciences* 11.17 (Aug. 2021), p. 7805. doi: 10.3390/app11177805.

[LW23] Heather Leary and Richard West. *Foundations of Learning and Instructional Design Technology*. 2nd ed. EdTech Books, 2023. doi: 10.59668/473.

[Mac92] I. Scott MacKenzie. “Fitts’ Law as a Research and Design Tool in Human-Computer Interaction”. In: *Human–Computer Interaction* 7.1 (Mar. 1992), pp. 91–139. doi: 10.1207/s15327051hci0701_3.

[Mag21] Juliana Magro. “H5P”. In: *Journal of the Medical Library Association: JMLA* 109.2 (2021), p. 351.

[Mar17] Robert C. Martin. *Clean architecture: a craftsman’s guide to software structure and design*. Prentice Hall Press, 2017.

[MAS23] A. M. Mutawa, Jamil Abdul Kareem Al Muttawa, and Sai Sruthi. “The effectiveness of using H5P for undergraduate students in the asynchronous distance learning environment”. In: *Applied Sciences* 13.8 (2023), p. 4983.

[May14] Philipp Mayring. “Qualitative content analysis: theoretical foundation, basic procedures and software solution”. In: (2014).

[May20] Richard Mayer. *Multimedia Learning*. 3rd ed. Cambridge University Press, July 2020. doi: 10.1017/9781316941355.

[MB18] Bodo Möslein-Tröppner and Willi Bernhard. *Digitale Gamebooks in der Bildung*. Springer, Jan. 2018. doi: 10.1007/978-3-658-21349-7.

[McI+68] M. Douglas McIlroy et al. “Mass-produced software components”. In: *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*. 1968, pp. 88–98.

[Mer19] Pekka Mertala. “Young children’s conceptions of computers, code, and the Internet”. In: *International Journal of Child-Computer Interaction* 19 (2019). doi: 10.1016/j.ijcci.2018.11.003.

[Mey86] Norman Meyrowitz. “Intermedia: The architecture and construction of an object-oriented hypemerdia system and applications framework”. In: *Conference proceedings on Object-oriented programming systems, languages and applications - OOPLSA ’86*. Portland, Oregon, United States: ACM Press, 1986, pp. 186–201. doi: 10.1145/28697.28716.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. “Past, present, and future of user interface software tools”. In: *ACM Trans. Comput.-Hum. Interact.* 7.1 (2000), pp. 3–28. doi: 10.1145/344949.344959.

[ML94] Marshall McLuhan and Lewis H. Lapham. *Understanding Media: The Extensions of Man*. Cambridge, Mass.: The MIT Press, 1994.

[MMK25] Alexey Melnikov, Darrel Miller, and Murray Kucherawy. *Media Types*. 2025.

[MMM95] H. Mili, F. Mili, and A. Mili. “Reusing software: issues and research directions”. In: *IEEE Transactions on Software Engineering* 21.6 (June 1995), pp. 528–562. doi: 10.1109/32.391379.

[MS23] John T. Murray and Anastasia Salter. “Mapping the Unmappable: Reimagining Visual Representations of Interactive Narrative”. In: *The Authoring Problem: Challenges in Supporting Authoring for Interactive Digital Narratives*. Springer, 2023, pp. 171–190. doi: 10.1007/978-3-031-05214-9_11.

[MSF80] M. David Merrill, Edward W. Schneider, and Kathie A. Fletcher. *Ticcit*. Vol. 2. Educational Technology, 1980.

[MT10] Morten Moshagen and Meinald T. Thielsch. “Facets of visual aesthetics”. In: *International journal of human-computer studies* 68.10 (2010), pp. 689–709.

[Mül20] Philipp Müller. “Understanding history: hermeneutics and source criticism in historical scholarship”. In: *Reading Primary Sources*. Routledge, 2020, pp. 23–40.

[Nat23] National Center for Education Statistics. *Children’s Internet Access at Home. Condition of Education. U.S. Department of Education, Institute of Education Sciences*. 2023.

[Nel65] T. H. Nelson. “Complex information processing: a file structure for the complex, the changing and the indeterminate”. In: *Proceedings of the 1965 20th national conference*. ACM ’65. New York, NY, USA: Association for Computing Machinery, Aug. 1965, pp. 84–100. doi: 10.1145/800197.806036.

[Nel99a] Ted Nelson. *Ted’s ComParadigm in OneLiners*. 1999. https://xanadu.com.au/ted/TN/WRITINGS/TCOMPARADIGM/tedCompOneLiners.html (accessed on 08/09/2025).

[Nel99b] Theodor Holm Nelson. “The unfinished revolution and Xanadu”. In: *ACM Computing Surveys* 31.4es (Dec. 1999), p. 37. doi: 10.1145/345966.346039.

[Nie93] Jakob Nielsen. *Usability engineering*. Boston: Academic Press, 1993.

[Nor02] Donald A. Norman. *The Design of Everyday Things*. [New York]: Basic Books, 2002.

[Nor89] Pat Norrish. “Semantic structures of text”. In: *Structured documents*. 1989, pp. 143–160.

[NS23] Svenja Noichl and Ulrik Schroeder. “Interaktionsmöglichkeiten und Potenziale digitaler Gamebooks unter Berücksichtigung der CELG-Taxonomie”. In: *21. Fachtagung Bildungstechnologien (DELFI)*. Bonn: Gesellschaft für Informatik e.V., 2023, pp. 305–306. doi: 10.18420/delfi2023-67.

[Nys14] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[OMG06] OMG. *CORBA Component Model Specification Version 4.0*. 2006.

[OMG21] OMG. *Common Object Request Broker Architecture Specification Version 3.4*. 2021.

[OMG91] OMG. *Common Object Request Broker Architecture Specification Version 1.1*. 1991.

[ORe21] Gerard O’Regan. “History of Operating Systems”. In: *A Brief History of Computing*. Ed. by Gerard O’Regan. Cham: Springer International Publishing, 2021, pp. 143–154. doi: 10.1007/978-3-030-66599-9_13.

[Par+19] Jennifer Paris et al. *Child growth and development*. College of the Canyons Santa Clarita, CA, 2019.

[Pav06] Ivan P. Pavlov. “The Scientific Investigation of the Psychical Faculties or Processes in the Higher Animals”. In: *Science* 24.620 (Nov. 1906), pp. 613–619. doi: 10.1126/science.24.620.613.

[Per82] Alan J. Perlis. “Special Feature: Epigrams on programming”. In: *ACM SIGPLAN Notices* 17.9 (Sept. 1982), pp. 7–13. doi: 10.1145/947955.1083808.

[Phi23] Carrie Anne Philbin. “Impact of Generative AI on K-12 Students’ Perceptions of Computing: A Research Proposal”. In: *Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research*. WiPSCE ’23. New York, NY, USA: Association for Computing Machinery, Sept. 2023, pp. 1–2. doi: 10.1145/3605468.3609775.

[Pia76] Jean Piaget. “Piaget’s Theory”. In: *Piaget and His School*. Ed. by Bärbel Inhelder, Harold H. Chipman, and Charles Zwingmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1976, pp. 11–23. doi: 10.1007/978-3-642-46323-5_2.

[Pil04] M. Pilato. “Version Control with Subversion”. In: July 2004.

[Plu24] Tsvetelina Plummer. *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*. 2024.

[Pop63] Karl R Popper. “Science as falsification”. In: *Conjectures and refutations* 1.1963 (1963), pp. 33–39.

[Pot96] Mike Potel. “MVP: Model-View-Presenter the Taligent programming model for C++ and Java”. In: *Taligent Inc* 20 (1996).

[PS19] Camille Paloque-Bergès and Valérie Schafer. “Arpanet (1969–2019)”. In: *Internet Histories* 3.1 (Jan. 2019), pp. 1–14. doi: 10.1080/24701475.2018.1560921.

[PT98] L. Prechelt and W.F. Tichy. “A controlled experiment to assess the benefits of procedure argument type checking”. In: *IEEE Transactions on Software Engineering* 24.4 (Apr. 1998), pp. 302–312. doi: 10.1109/32.677186.

[QCJ21] Farzana Quayyum, Daniela S Cruzes, and Letizia Jaccheri. “Cybersecurity awareness for children: A systematic literature review”. In: *International Journal of Child-Computer Interaction* 30 (2021). doi: 10.1016/j.ijcci.2021.100343.

[QV94] Vincent Quint and Irène Vatton. “Making structured documents active”. In: *Electronic publishing* 7.2 (1994), pp. 55–74.

[Rab+23] Anna Rabe et al. “How to Deal with Students’ Conceptions: Conceptual Change Texts on Artificial Intelligence”. In: *Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research*. WiPSCE ’23. New York, NY, USA: Association for Computing Machinery, Sept. 2023, pp. 1–2. doi: 10.1145/3605468.3609781.

[Rad+23] Alec Radford et al. “Robust speech recognition via large-scale weak supervision”. In: *International conference on machine learning*. PMLR, 2023, pp. 28492–28518.

[Ray+14] Baishakhi Ray et al. “A large scale study of programming languages and code quality in github”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 155–165. doi: 10.1145/2635868.2635922.

[Ray99] Eric Raymond. “The cathedral and the bazaar”. In: *Knowledge, Technology & Policy* 12.3 (Sept. 1999), pp. 23–49. doi: 10.1007/s12130-999-1026-0.

[Rei03] Frederick F. Reichheld. “The one number you need to grow”. In: *Harvard business review* 81.12 (2003), pp. 46–55.

[Ren+24] Karen Renaud et al. ““We’re Not That Gullible!” Revealing Dark Pattern Mental Models of 11-12-Year-Old Scottish Children”. In: *ACM Transactions on Computer-Human Interaction* (2024). doi: 10.1145/3660342.

[RFA24] Ulfia Rahmi, Bayu Ramadhani Fajri, and Azrul Azrul. “Effectiveness of interactive content with H5P for Moodle-learning management system in blended learning”. In: *Journal of Learning for Development* 11.1 (2024), pp. 66–81.

[RR03] T. Ravichandran and Marcus A. Rothenberger. “Software reuse strategies and component markets”. In: *Communications of the ACM* 46.8 (Aug. 2003), pp. 109–114. doi: 10.1145/859670.859678.

[RSP23] Yvonne Rogers, Helen Sharp, and Jennifer Preece. *Interaction Design: Beyond Human-Computer Interaction*. Hoboken: Wiley, Apr. 2023.

[Sal24] Frederic Salmen. *ExplorableJS*. 2024. https://explorablejs.elearn.rwth-aachen.de/ (accessed on 12/13/2025).

[SC20] Rachelle Singleton and Amanda Charlton. “Creating H5P content for active learning”. In: *Pacific Journal of Technology Enhanced Learning* 2.1 (2020), pp. 13–14.

[Sch13] Peter Schaefer. “Why is" ether" in Ethernet?” In: *International Journal of Communication (19328036)* 7 (2013).

[Sch19] Dale Schunk. *Learning Theories: An Educational Perspective*. Hoboken, NJ: Pearson, Jan. 2019.

[Sch21] Martin Schrepp. “Measuring User Experience with Modular Questionnaires”. In: *2021 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. Oct. 2021, pp. 1–6. doi: 10.1109/ICACSIS53237.2021.9631321.

[SD09] Jeff Sauro and Joseph S. Dumas. “Comparison of three one-question, post-task usability questionnaires”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Boston MA USA: ACM, Apr. 2009, pp. 1599–1608. doi: 10.1145/1518701.1518946.

[Sei14] Niels Seidel. “Interaction design patterns for design and development of video learning environments”. In: *Proceedings of the 19th European Conference on Pattern Languages of Programs*. EuroPLoP ’14. New York, NY, USA: Association for Computing Machinery, July 2014, pp. 1–12. doi: 10.1145/2721956.2721981.

[Ser+06] Isabel Seruca et al., eds. *Enterprise Information Systems VI*. Dordrecht: Springer Netherlands, 2006. doi: 10.1007/1-4020-3675-2.

[Ser+19] Sebastian Serth et al. “Evaluating Digital Worksheets With Interactive Programming Exercises for K-12 Education”. In: *2019 IEEE Frontiers in Education Conference (FIE)*. Covington, KY, USA: IEEE Press, 2019, pp. 1–9. doi: 10.1109/FIE43999.2019.9028680.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[Shn+17] Ben Shneiderman et al. *Designing the user interface: strategies for effective human-computer interaction*. Sixth Edition. Boston: Pearson, 2017.

[Shn83] Ben Shneiderman. “Direct manipulation: A step beyond programming languages”. In: *Computer* 16.08 (1983), pp. 57–69.

[Ski65] Burrhus Frederic Skinner. *Science and human behavior*. Simon and Schuster, 1965.

[Smi+17] Daniel Smilkov et al. *Direct-Manipulation Visualization of Deep Networks*. Aug. 2017. doi: 10.48550/arXiv.1708.03788.

[Smu09] Aaron Smuts. “What is interactivity?” In: *The Journal of Aesthetic Education* 43.4 (2009), pp. 53–73.

[Soe25] Mads Soegaard. *Interaction Styles*. 2025. https://www.interaction-design.org/literature/book/the-glossary-of-human-computer-interaction/interaction-styles (accessed on 09/01/2025).

[Som11] Ian Sommerville. *Software engineering*. 9th ed. Boston: Pearson, 2011.

[Spi05] Clay Spinuzzi. “The methodology of participatory design”. In: *Technical communication* 52.2 (2005), pp. 163–174.

[Spi88] Robert Spinrad. “Dynamic documents”. In: *Harvard University Information Technology Quarterly* 7.1 (1988), pp. 15–18.

[SS20] Atikah Shemshack and Jonathan Michael Spector. “A systematic literature review of personalized learning terms”. In: *Smart Learning Environments* 7.1 (Oct. 2020), p. 33. doi: 10.1186/s40561-020-00140-9.

[SSR21] P. Sinnayah, A. Salcedo, and S. Rekhari. “Reimagining physiology education with interactive content developed in H5P”. In: *Advances in Physiology Education* 45.1 (Mar. 2021), pp. 71–76. doi: 10.1152/advan.00021.2020.

[ST19] Martin Schrepp and Jorg Thomaschewski. “Design and validation of a framework for the creation of user experience questionnaires”. In: *IJIMAI* 5.7 (2019), pp. 88–95.

[Sta15] André Staltz. *Unidirectional User Interface Architectures*. 2015. https://staltz.com/unidirectional-user-interface-architectures (accessed on 09/20/2025).

[Sta24] UX Stalin. *The Working Memory Model*. June 2024. https://commons.wikimedia.org/wiki/File:The_Working_Memory_Model.svg (accessed on 08/11/2025).

[Sta25] Inc. Stack Exchange. *2025 Stack Overflow Developer Survey*. 2025. https://survey.stackoverflow.co/2025/ (accessed on 08/18/2025).

[Str04] Jennifer Stromer-Galley. “Interactivity-as-product and interactivity-as-process”. In: *The Information Society* 20.5 (2004), pp. 391–394.

[Swe11] John Sweller. “Cognitive load theory”. In: *Psychology of learning and motivation*. Vol. 55. Elsevier, 2011, pp. 37–76.

[Syk17] Martin Sykora. “Web 1.0 to Web 2.0: an observational study and empirical evidence for the historical r(evolution) of the social web”. In: *International Journal of Web Engineering and Technology* 12.1 (2017), p. 70. doi: 10.1504/IJWET.2017.084024.

[Szy99] Clemens Szyperski. “Components vs. objects vs. component objects”. In: *Proceedings of OOP*. Vol. 99. 1999.

[TB90] Douglas B Terry and Donald G Baker. “Active Tioga documents: an exploration of two paradigms”. In: (1990).

[TDT21] Matti Tedre, Peter Denning, and Tapani Toivonen. “CT 2.0”. In: Nov. 2021, pp. 1–8. doi: 10.1145/3488042.3488053.

[Ted+20] Matti Tedre et al. “Machine learning introduces new perspectives to data agency in K—12 computing education”. In: *2020 IEEE frontiers in education conference (FIE)*. IEEE, 2020, pp. 1–8.

[Tho98] Edward L. Thorndike. *Animal intelligence*. 1898.

[Tic85] Walter F. Tichy. “Rcs — a system for version control”. In: *Software: Practice and Experience* 15.7 (July 1985), pp. 637–654. doi: 10.1002/spe.4380150703.

[Tou+23] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. Feb. 2023. doi: 10.48550/arXiv.2302.13971.

[Tsa+20] Katerina Tsarava et al. “Fostering Knowledge of Computer Viruses among Children: The Effects of a Lesson with a Cartoon Series”. In: *Proceedings of the 20th Koli calling international conference on computing education research*. 2020. doi: 10.1145/3428029.3428033.

[Tsi+16] Andreas Tsirtsis et al. “Cyber Security Risks for Minors: A Taxonomy and a Software Architecture”. In: *11th International Workshop on Semantic and Social Media Adaptation and Personalization*. 2016. doi: 10.1109/SMAP.2016.7753391.

[Tur37] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. doi: 10.1112/plms/s2-42.1.230.

[Tur78] Murray Turoff. “The EIES Experience: Electronic Information Exchange System.” In: *Bulletin of the American Society for Information Science* 4.5 (1978), pp. 9–10.

[UK13] GOV UK. *National curriculum in England: computing programmes of study*. 2013.

[UNE12] UNESCO, ed. *International standard classification of education: ISCED 2011*. Montreal, Quebec: UNESCO Institute for Statistics, 2012.

[UNE14] UNESCO, ed. *ISCED Fields of education and training 2013 (ISCED-F 2013): Manual to accompany the International Standard Classification of Education 2011*. UNESCO Institute for Statistics, 2014.

[UP22] Amanda J. Unsworth and Mareike G. Posner. “Case Study: Using H5P to design and deliver interactive laboratory practicals”. In: *Essays in Biochemistry* 66.1 (2022), pp. 19–27.

[Vai23] Lionel Sujay Vailshery. *Global developer population 2018-2024*. 2023.

[Val02] Gilbert A. Valverde. *According to the book: using TIMSS to investigate the translation of policy into practice through the world of textbooks*. New York, [New York]: Springer-Science+Business Media, LLC, 2002.

[Van88] Andries Van Dam. “Hypertext ’87: keynote address”. In: *Communications of the ACM* 31.7 (July 1988), pp. 887–895. doi: 10.1145/48511.48519.

[Vas+23] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 2023. doi: 10.48550/arXiv.1706.03762.

[VC78] Lev Semenovich Vygotsky and Michael Cole. *Mind in society: Development of higher psychological processes*. Harvard university press, 1978.

[Ver87] Christian Verlaque. “Les espaces du Minitel”. In: *NETCOM: Réseaux, communication et territoires/Networks and communication studies* 1.2 (1987), pp. 316–327.

[Vic11a] Bret Victor. *Explorable Explanations*. 2011. http://worrydream.com/ExplorableExplanations/ (accessed on 08/23/2022).

[Vic11b] Bret Victor. *Up and Down the Ladder of Abstraction*. 2011. https://worrydream.com/LadderOfAbstraction/ (accessed on 08/03/2025).

[Von93] John Von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75.

[VP23] Gia Minh Vo and Nils Pancratz. “AI Education in German K-10 Computer Science Curricula”. In: *Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research*. WiPSCE ’23. New York, NY, USA: Association for Computing Machinery, Sept. 2023, pp. 1–4. doi: 10.1145/3605468.3605471.

[WAP08] Carl E. Wieman, Wendy K. Adams, and Katherine K. Perkins. “PhET: Simulations That Enhance Learning”. In: *Science* 322.5902 (Oct. 2008), pp. 682–683. doi: 10.1126/science.1161948.

[War93] Martin Ward. “Formal Methods and Transformations in Software Reuse”. In: *Integrated Software Reuse: Management and Techniques*. Ed. by Paul Walton and Neil Maiden. 1st ed. Routledge, 1993, pp. 77–92. doi: 10.4324/9780429455520-7.

[WC22] Robert E. White and Karyn Cooper. “Grounded Theory”. In: *Qualitative Research in the Post-Modern Era*. Cham: Springer International Publishing, 2022, pp. 339–385. doi: 10.1007/978-3-030-85124-8_9.

[WC65] Richard E. Walther and Norman Crowder. *A guide to preparing intrinsically programmed instructional materials*. Tech. rep. 1965.

[Wel20] Martin Weller. *25 Years of Ed Tech*. Issues in Distance Education. Canada: Athabasca University Press, 2020.

[Wel37] Herbert G. Wells. “World brain: The idea of a permanent world encyclopaedia”. In: *Encyclopédie Française* 18 (1937), pp. 24–11.

[WH18] David Wiley and John Levi Hilton III. “Defining OER-enabled pedagogy”. In: *The International Review of Research in Open and Distributed Learning* 19.4 (2018).

[WHA25] WHATWG. *HTML Standard*. 2025. https://html.spec.whatwg.org/ (accessed on 08/18/2025).

[Wic+21] Julien Arief Wicaksono et al. “The use of H5P in teaching English”. In: *The First International Conference on Social Science, Humanity, and Public Health (ICOSHIP 2020)*. Atlantis Press, 2021, pp. 227–230.

[Wil+04] David Wiley et al. “Overcoming the Limitations of Learning Objects”. In: *Journal of Educational Multimedia and Hypermedia* 13.4 (Oct. 2004), pp. 507–521.

[Wil+16] Mark D. Wilkinson et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Scientific data* 3.1 (2016), pp. 1–9.

[Wil00] David A. Wiley. “Connecting learning objects to instructional design theory. A definition, a metaphor, and a taxonomy”. In: (2000).

[Wil08] David A. Wiley. “The learning objects literature”. In: *Handbook of research on educational communications and technology*. Routledge, 2008, pp. 345–353.

[Wil09] David A. Wiley. *Dark Matter, Dark Reuse, and the Irrational Zeal of a Believer. Iterating Toward Openness*. 2009.

[Win06] Jeannette M Wing. “Computational thinking”. In: *Communications of the ACM* 49.3 (2006), pp. 33–35.

[Wou+13] Pieter Wouters et al. “A meta-analysis of the cognitive and motivational effects of serious games.” In: *Journal of Educational Psychology* 105.2 (May 2013), pp. 249–265. doi: 10.1037/a0031311.

[WP25] James M. Wells and Warren E. Preece. *Typography*. Aug. 2025. https://www.britannica.com/technology/typography/History-of-typography (accessed on 08/27/2025).

[Wu+19] Yuhao Wu et al. “How do developers utilize source code from stack overflow?” In: *Empirical Software Engineering* 24.2 (Apr. 2019), pp. 637–673. doi: 10.1007/s10664-018-9634-5.

[WU17] Tom Warnke and Adelinde M. Uhrmacher. “Practical expressiveness of internal and external domain-specific modeling languages”. In: *2017 Winter Simulation Conference (WSC)*. Dec. 2017, pp. 4566–4567. doi: 10.1109/WSC.2017.8248207.

[YG23] Joseph Yoder and Richard Gabriel. *Proceedings of the 30th Conference on Pattern Languages of Programs*. 2023.

[YHL16] Di Yang, Aftab Hussain, and Cristina Videira Lopes. “From query to usable code: an analysis of stack overflow code snippets”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. Austin Texas: ACM, May 2016, pp. 391–402. doi: 10.1145/2901739.2901767.

Appendix F Glossary

Terms which are assumed to be common knowledge (e.g. web and internet) are not defined here. Description of simple terms (e.g. STEM) in this glossary are sourced from the English Wikipedia (`https://en.wikipedia.org` as of December 22, 2025) articles of the same name. Domain standard technologies such as programming languages are listed here, as well (e.g. Python). See Appendix D for more specific technologies such as software and programming libraries. Terms from the standardized web platform are partially sourced from the Mozilla Development Network MDN (`https://developer.mozilla.org/` as of December 22, 2025) and are referenced as such. Complex research terms (e.g. reusability paradox) include a reference back to the section in which they are defined, which contains literature references.

A

Active learning Active learning is a method of learning in which students are actively or experientially involved in the learning process and where there are different levels of active learning, depending on student involvement.

ActiveX ActiveX is a deprecated software framework created by Microsoft that adapts its earlier Component Object Model (COM) and Object Linking and Embedding (OLE) technologies for content downloaded from a network, particularly from the World Wide Web.

Adaptive learning Adaptive learning, also known as adaptive teaching, is an educational method which uses computer algorithms as well as artificial intelligence to orchestrate the interaction with the learner and deliver customized resources and learning activities to address the unique needs of each learner.

API An application programming interface (API) is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software.

AR Augmented reality (AR), also known as mixed reality (MR), is a form of 3D human–computer interaction that overlays real-time 3D-rendered computer graphics into the real world through a display, such as a handheld device or head-mounted display.

ARPANET The Advanced Research Projects Agency Network (ARPANET) was the first wide-area packet-switched network with distributed control and one of the first computer networks to implement the TCP/IP protocol suite. Both technologies became the technical foundation of the Internet. The ARPANET was established by the Advanced Research Projects Agency (now DARPA) of the United States Department of Defense.

Artifact An artifact is one of many kinds of tangible by-products produced during the development of software. Artifact occasionally may refer to the released code (in the case of a code library) or released executable (in the case of a program) produced.

Audiovisual education Audiovisual (AV) education or multimedia-based education (MBE) is an instruction method where particular attention is paid to the audiovisual or multimedia presentation of the material to improve comprehension and retention.

Authoring paradigm An authoring paradigm combines interaction styles to modify elements (morphology), to rearrange elements into a composite resource (syntax), and to annotate the resource with metadata (pragmatics). [section 3.2]

B

Base64 Base64 is a binary-to-text encoding that uses 64 printable characters to represent each 6-bit segment of a sequence of byte values. As for all binary-to-text encodings, Base64 encoding enables transmitting binary data on a communication channel that only supports text.

BBCode BBCode ("Bulletin Board Code") is a lightweight markup language used to format messages in many Internet forum software. It was first introduced in 1998. The available "tags" of BBCode are usually indicated by square brackets surrounding a keyword, and are parsed before being translated into HTML.

Behaviorism Behaviorism is a systematic approach to understanding the behavior of humans and other animals. It assumes that behavior is either a reflex elicited by the pairing of certain antecedent stimuli in the environment, or a consequence of that individual's history, including especially reinforcement and punishment contingencies, together with the individual's current motivational state and controlling stimuli.

Betamax Betamax is a discontinued consumer analog videocassette recording format developed by Sony. It was one of the main competitors in the videotape format war against its primary rival, VHS.

Bundler A module bundler is a software development tool that takes multiple source files and their dependencies (modules) and combines them into one or more output files called bundles. Module bundlers are commonly used in front-end web development to package JavaScript, CSS and other assets for efficient delivery to web browsers.

C

C++ C++ is a high-level, general-purpose programming language created by Danish computer scientist Bjarne Stroustrup. First released in 1985 as an extension of the C programming language, adding object-oriented (OOP) features, it has since expanded significantly over time adding more OOP and other features.

Canvas The Canvas API provides a means for drawing graphics via JavaScript and the HTML `<canvas>` element. Among other things, it can be used for animation, game graphics, data visualization, photo manipulation, and real-time video processing. The Canvas API largely focuses on 2D graphics. [MDN]

CCM CORBA Component Model (CCM) is an addition to the family of CORBA definitions.[5] It was introduced with CORBA 3 and it describes a standard application framework for CORBA components.

CDN A content delivery network (CDN) or content distribution network is a geographically distributed network of proxy servers and corresponding data centers. CDNs provide high availability and performance ("speed") through geographical distribution relative to end users.

CD-ROM A CD-ROM is a type of read-only memory consisting of a pre-pressed optical compact disc that contains data computers can read, but not write or erase.

Class In programming, a class is a syntactic entity structure used to create objects. The capabilities of a class differ between programming languages, but generally the shared aspects consist of state (variables) and behavior (methods) that are each either associated with a particular object or with all objects of that class.

Cluefinders The ClueFinders is an educational software series aimed at children aged 8–12 that features a group of mystery-solving teenagers.

CMS A content management system (CMS) is a computer software used to manage the creation and modification of digital content (content management). It is typically used for enterprise content management (ECM) and web content management (WCM). ECM typically supports multiple users in a collaborative environment, by integrating document management, digital asset management, and record retention.

Cognitivism In psychology, cognitivism is a theoretical framework for understanding the mind that gained credence in the 1950s. Cognitive psychology derived its name from the Latin cognoscere, referring to knowing and information, thus cognitive psychology is an information-processing psychology derived in part from earlier traditions of the investigation of thought and problem solving.

COM Component Object Model (COM) is a binary-interface technology for software components from Microsoft that enables using objects in a language-neutral way between different programming languages, programming contexts, processes and machines.

COM+ COM+ is a now deprecated successor of COM with support for distributed transactions, resource pooling, disconnected applications, event publication and subscription, bad better memory and processor (thread) management.

Command language Interaction style where expert users type in prompts with parameters to instruct the system. This is usually done in a command line interface (CLI) or terminal, where previous commands and outputs are shown in sequence, with a command prompt at the bottom. [section 3.2]

CommonJS module CommonJS is a project to standardize the module ecosystem for JavaScript outside of web browsers (e.g. on web servers or native desktop applications). CommonJS's specification of how modules should work is widely used today for server-side JavaScript with Node.js.

Component A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties. [section 3.1]

Connected Connected content would be active resources with runtimes that communicate with other runtimes through APIs, similar to the original idea of the Semantic Web. [section 3.2]

Constructivism Constructivism is a theory that suggests that learners do not passively acquire knowledge through direct instruction. Instead, they construct their understanding through experiences and social interaction, integrating new information with their existing knowledge.

Container In software engineering, containerization is operating-system-level virtualization or application-level virtualization over multiple network resources so that software applications can run in isolated user spaces called containers in any cloud or non-cloud environment, regardless of type or vendor.

Content repository A content repository or content store is a database of digital content with an associated set of data management, search and access methods allowing application-independent access to the content, rather like a digital library, but with the ability to store and modify content in addition to searching and retrieving.

CORBA The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) designed to facilitate the communication of systems that are deployed on diverse platforms.

CORS Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. [MDN]

CRUD In computer programming, create, read, update, and delete (CRUD) are the four basic operations (actions) of persistent storage. CRUD is also sometimes used to describe user interface conventions that facilitate viewing, searching, and changing information using computer-based forms and reports.

CSS Cascading Style Sheets (CSS) is a style sheet language used for specifying the presentation and styling of a document written in a markup language such as HTML or XML.

CUI A conversational user interface (CUI) is a user interface for computers that emulates a conversation with a human. CUIs provide opportunity for the user to communicate with the computer in their natural language rather than in a syntax specific commands. [section 7.1]

Custom element There are two parts to Custom Elements: autonomous custom elements and customized built-in elements. Autonomous custom elements are HTML elements that are entirely separated from native HTML elements; they are essentially built from the bottom up using the Custom Elements API. Customized built-in elements are elements that are built upon native HTML elements to reuse their functionality.

Custom element registry The CustomElementRegistry interface provides methods for registering custom elements and querying registered elements. [MDN]

D

DARPA The Defense Advanced Research Projects Agency (DARPA) is a research and development agency of the United States Department of Defense responsible for the development of emerging technologies for use by the military.

Dart Dart is a programming language designed by Lars Bak and Kasper Lund and developed by Google. It can be used to develop web and mobile apps as well as server and desktop applications. It is an object-oriented, class-based, garbage-collected language with C-style syntax.

Data URL Data URLs, URLs prefixed with the data: scheme, allow content creators to embed small files inline in documents. They were formerly known as "data URIs" until that name was retired by the WHATWG. [MDN]

DBR Design-based research (DBR) is a type of research methodology used by researchers in the learning sciences. The basic process of DBR involves developing solutions (called "interventions") to problems. Then, the interventions are put to use to test how well they work. The iterations may then be adapted and re-tested to gather more data.

Design pattern In software engineering, a software design pattern or design pattern is a general, reusable solution to a commonly occurring problem in many contexts in software design. A design pattern is not a rigid structure to be transplanted directly into source code. Rather, it is a description or a template for solving a particular type of problem that can be deployed in many different situations.

Direct manipulation An interaction style where a user is presented with a set of predefined form fields to fill out. Usually, digital forms consist of a set of input fields with fixed labels. [section 3.2]

DNS The Domain Name System (DNS) is a hierarchical and distributed name service that provides a naming system for computers, services, and other resources on the Internet or other Internet Protocol (IP) networks.

DOM The Document Object Model (DOM) is a cross-platform and language-independent API that treats an HTML or XML document as a tree structure wherein each node is an object representing a part of the document. The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects.

DVD DVD (digital video disc or digital versatile disc) is a digital optical disc data storage format. The medium can store any kind of digital data and has been widely used to store video programs (watched using DVD players), software and other computer files.

DX Developer Experience (DX or DevEx) is a field within software engineering that examines how developers think, feel, and interact with their work environments, tools, and processes.

E

EC-TEL This conference is one of the conferences with the longest trajectory in the field of Technology Enhanced Learning. It has become the main conference in Europe in this topic, and it usually attracts the main researchers in the field within Europe, as well as researchers from across the world.

EJB Jakarta Enterprise Beans (EJB; formerly Enterprise JavaBeans) is one of several Java APIs for modular construction of enterprise software. EJB is a server-side software component that encapsulates business logic of an application.

Elm Elm is a domain-specific programming language for declaratively creating web browser-based graphical user interfaces. Elm is purely functional, and is developed with emphasis on usability, performance, and robustness.

Erlang Erlang is a general-purpose, concurrent, functional high-level programming language, and a garbage-collected runtime system.

ES module ECMAScript modules are the official standard format to package JavaScript code for reuse. Modules are defined using a variety of import and export statements.

Explorable An explorable explanation (often shortened to explorable) is a form of informational media where an interactive computer simulation of a given concept is presented, along with some form of guidance (usually prose) that suggests ways that the audience can learn from the simulation.

Explorable authoring Authoring paradigm for interactive content: Treat interactive documents as rich text, with rich text editing behavior for static elements. For interactive elements, extend this behavior through self editing, where interactive element may provide affordances to modify themselves depending on the author's interaction. [section 7.1]

Explorable authoring requirements A systematic catalog of requirements for explorable authoring systems. It has has two main perspectives: Authoring and development. Authoring refers to the editing of documents, and development refers to the editing of extensions. For each perspective, there is both the quality of functionality (which enables expressiveness) and the quality of usability to consider, resulting in the four categories of authoring functionality, authoring experience, development functionality, and development experience. [section 5.2]

Explorable authoring system A system which integrates both a design pattern for interactive elements and an authoring paradigm to enable the remix of interactive, hypermodal and static content into a structured document. [section 3.3]

Explorable reading Counterpart to explorable authoring, modeling reading of the authored document as a type of limited authoring. Render a document and allow self editing for interactive elements, where those elements may provide affordances to modify themselves depending on the reader's interaction. [section 7.1]

Expressiveness Practical expressiveness of a language or approach means that actors of a certain domain can make use of that language or approach to express any product desired (see section 1.3).

F

FAANG Big Tech (FAANG), also referred to as the tech giants or tech titans, is a collective term for the largest and most influential technology companies in the world. It commonly denotes the five dominant firms in the U.S. technology industry—Microsoft, Apple, Alphabet (Google), Amazon, and Meta (Facebook).

FAIR An acronym for the four principles of findability, accessibility, interoperability, and reusability (FAIR), which describe practices make research data reusable. Note that accessibility is meant literally as access to the data, not in the usual sense of accessibility as inclusive practices regarding disabilities. [section 3.1]

FAIR4RS An adaptation of FAIR that considers software (FAIR for Research Software). Compared to the original FAIR principles, most are adapted as-is. Some software-typical concerns are also addressed, such as versioning. Additionally: FAIR4RS introduces an idea of a hierarchy of components inside the software, as it mandates distinct IDs for components. It includes a concept of the software actively reading/writing/exchanging data.

File System Access API The File System API — with extensions provided via the File System Access API to access files on the device file system — allows read, write and file management capabilities. [MDN]

Flash Adobe Flash (formerly Macromedia Flash and FutureSplash) is a mostly discontinued multimedia software platform used for production of animations, rich internet applications, desktop applications, mobile apps, mobile games, and embedded web browser video players.

Form fill-in Form fill-in means presenting a user with a set of predefined form fields to fill out. Usually, digital forms consist of a set of input fields with fixed labels. Input fields can be different depending on which type of data should be entered. [section 3.2]

Friendster Friendster was a social networking service originally based in Mountain View, California, founded by Jonathan Abrams and launched in March 2002. The service allowed users to contact other members, maintain those contacts, and share online content and media with those contacts.

G

Git Git is a distributed version control software system that is capable of managing versions of source code or data. It is often used to control source code by programmers who are developing software collaboratively.

GML Generalized Markup Language (GML) is a set of macros that implement intent-based (procedural) markup tags for the IBM text formatter, SCRIPT. SCRIPT/VS is the main component of IBM's Document Composition Facility (DCF).

Go Go is a high-level, general-purpose programming language that is statically-typed and compiled. It is known for the simplicity of its syntax and the efficiency of development that it enables through the inclusion of a large standard library supplying many needs for common projects.

Groff groff (also called GNU troff) is a typesetting system that creates formatted output when given plain text mixed with formatting commands. It is the GNU replacement for the troff and nroff text formatters, which were both developed from the original roff.

Grounded theory Grounded theory is a systematic methodology that has been largely applied to qualitative research conducted by social scientists. The methodology involves the construction of hypotheses and theories through the collection and analysis of data.

GUI A graphical user interface, or GUI[a], is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation.

H

Haxe Haxe is a high-level cross-platform programming language and compiler that can produce applications and source code for many different computing platforms from one code-base. It is free and open-source software, released under an MIT License.

HTML Hypertext Markup Language (HTML) is the standard markup language for documents designed to be displayed in a web browser. It defines the content and structure of web content.

HTTP HTTP (Hypertext Transfer Protocol) is an application layer protocol in the Internet protocol suite for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access, for example by a mouse click or by tapping the screen in a web browser.

Hypermedia Hypermedia, an extension of hypertext, is a nonlinear medium of information that includes graphics, audio, video, plain text and hyperlinks. This designation contrasts with the broader term multimedia, which may include non-interactive linear presentations as well as hypermedia.

Hypertext Hypertext is text displayed on a computer display or other electronic devices with references (hyperlinks) to other text that the reader can immediately access. Hypertext documents are interconnected by hyperlinks, which are typically activated by a mouse click, keypress set, or screen touch.

I

ICT Information and communications technology (ICT) is an extensional term for information technology (IT) that stresses the role of unified communications and the integration of telecommunications (telephone lines and wireless signals) and computers, as well as necessary enterprise software, middleware, storage and audiovisual, that enable users to access, store, transmit, understand and manipulate information.

IDN Interactive digital narrative editors (IDNs) are authoring tools for adaptive storytelling experiences. [section 7.2]

Instructional design Instructional design (ID), also known as instructional systems design and originally known as instructional systems development (ISD), is the practice of systematically designing, developing and delivering instructional materials and experiences, both digital and physical, in a consistent and reliable fashion toward an efficient, effective, appealing, engaging and inspiring acquisition of knowledge.

Interaction style In UX, an interaction style defines how a user interacts with a system. [section 3.2]

Interactive element Interactive elements make up interactive multimedia resources. They by definition cannot be divided further. They are programmed using the methods of software engineering and reused in contexts such as in authoring tools, which help with composition.

Interactive multimedia Interactive media refers to digital experiences that dynamically respond to user input, delivering content such as text, images, animations, video, audio, and even AI-driven interactions.

Invasive composition Principle for composing active documents where the included component has the power to "invasively" change the document itself, as it is not sandboxed.

IPhone The iPhone is a line of smartphones developed and marketed by Apple that run iOS, the company's own mobile operating system. The first-generation iPhone was announced by then–Apple CEO and co-founder Steve Jobs on January 9, 2007, at Macworld 2007, and launched later that year.

ISCED The International Standard Classification of Education (ISCED) is a statistical framework for organizing information on education maintained by the United Nations Educational, Scientific and Cultural Organization (UNESCO). It is a member of the international family of economic and social classifications of the United Nations. It defines levels of education and also fields of education (ISCED-F).

ISTE The International Society for Technology in Education (ISTE) is a nonprofit organization that focuses on accelerating innovation in education through the smart use of technology in education. ISTE provides a variety of services to support professional learning for educators and education leaders, including ISTELive — an ed tech event, the ISTE Standards for learning, teaching and leading with technology, and ISTE Certification.

J

Java Java is a high-level, general-purpose, memory-safe, object-oriented programming language. It is intended to let programmers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need to recompile.

JavaBeans In computing based on the Java Platform, JavaBeans is a technology developed by Sun Microsystems and released in 1996, as part of JDK 1.1. The 'beans' of JavaBeans are classes that encapsulate one or more objects into a single standardized object (the bean).

JS JavaScript (JS) is a programming language and core technology of the Web, alongside HTML and CSS. It was created by Brendan Eich in 1995. As of 2025, the overwhelming majority of websites (98.9%) uses JavaScript on the client side for webpage behavior.

JSON JSON (JavaScript Object Notation) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of name–value pairs and arrays. It is a commonly used data format with diverse uses in electronic data interchange, including that of web applications with servers.

JSX JSX (sometimes referred to as JavaScript XML) is an XML-like extension to the JavaScript language syntax. Initially for use with React, JSX has been adopted by multiple web frameworks. Being syntactic sugar, JSX is generally transpiled into nested JavaScript function calls structurally similar to the original JSX.

JVM A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

L

LA Learning analytics (LA) is the measurement, collection, analysis and reporting of data about learners and their contexts, for purposes of understanding and optimizing learning and the environments in which it occurs.

LAN A local area network (LAN) is a computer network that interconnects computers within a limited area such as a residence, campus, or building, and has its network equipment and interconnects locally managed. LANs facilitate the distribution of data and sharing network devices, such as printers.

LaTeX LaTeX is a software system for typesetting documents, based on TeX. LaTeX provides a high-level, descriptive markup language to utilize TeX more easily: TeX handles the document layout, while LaTeX handles the content side for document processing.

Learning object A learning object is a collection of content items, practice items, and assessment items that are combined based on a single learning objective.

Learning simulation A simulation is an imitative representation of a process or system that could exist in the real world, and allows experimentation with the help of a model.

Literate programming Literate programming (LP) is a programming paradigm introduced in 1984 by Donald Knuth in which a computer program is given as an explanation of how it works in a natural language, such as English, interspersed (embedded) with snippets of macros and traditional source code, from which compilable source code can be generated.

LLM A large language model (LLM) is a language model trained with self-supervised machine learning on a vast amount of text, designed for natural language processing tasks, especially language generation. The largest and most capable LLMs are generative pre-trained transformers (GPTs) and provide the core capabilities of modern chatbots.

LMS A learning management system (LMS) is a software application for the administration, documentation, tracking, reporting, automation, and delivery of educational courses, training programs, materials or learning and development programs.

LocalStorage Local storage is a browser functionality for storing data persistently. The stored data is saved across browser sessions. [MDN]

M

Markdown Markdown is a lightweight markup language for creating formatted text using a plain-text editor. Markdown is widely used for blogging, instant messaging, and large language models, and also used elsewhere in online forums, collaborative software, documentation pages, and readme files.

Markup A markup language is a text-encoding system which specifies the structure and formatting of a document and potentially the relationships among its parts. Markup can control the display of a document or enrich its content to facilitate automated processing.

MathML Mathematical Markup Language (MathML) is a pair of mathematical markup languages, an application of XML for describing mathematical notation and capturing both its structure and content. Its aim is to natively integrate mathematical formulae into World Wide Web pages and other documents.

Media Capture and Streams API The Media Capture and Streams API, often called the Media Streams API or MediaStream API, is an API related to WebRTC which provides support for streaming audio and video data. It provides the interfaces and methods for working with the streams and their constituent tracks. [MDN]

Media type A media type (formerly known as a Multipurpose Internet Mail Extensions or MIME type) indicates the nature and format of a document, file, or assortment of bytes. MIME types are defined and standardized in IETF's RFC 6838. [MDN]

Menu selection An interaction style where available commands are labeled, grouped and shown to the user (for example in a menu), who can select them and observe a change in the system. Commonly, this is coupled with keyboard shortcuts to execute commands quickly. [section 3.2]

Mixin In object-oriented programming languages, a mixin (or mix-in) is a class that contains methods for use by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language. Mixins are sometimes described as being "included" rather than "inherited".

Module Modular programming is a programming paradigm that emphasizes organizing the functions of a codebase into independent modules – each providing an aspect of a computer program in its entirety without providing other aspects.

Monotype The Monotype typefaces were developed by the Lanston Monotype Machine Company and its successors, originally as metal type for use in printing presses.

Morphology In our learning resource context, the layer of morphology deals with properties of the small units that make up a resource. [section 3.2]

Multimedia Multimedia is a form of communication that uses a combination of different content forms, such as writing, audio, images, animations, or video, into a single presentation. This is in contrast to traditional mass media, such as printed material or audio recordings, which only feature one form of media content.

Multimedia learning Multimedia learning is an educational approach that combines words and pictures (visuals and audio/text) to create deeper understanding, leveraging our brain's dual channels for processing information, rather than just words alone, through active selection, organization, and integration of these elements with prior knowledge.

MVVM Model–view–viewmodel (MVVM) is an architectural pattern in computer software that facilitates the separation of the development of a graphical user interface (GUI; the view), be it via a markup language or GUI code, from the development of the business logic or back-end logic (the model) such that the view is not dependent upon any specific model platform.

N

.NET Framework The .NET Framework is a proprietary software framework developed by Microsoft that runs primarily on Microsoft Windows. It was the predominant implementation of the Common Language Infrastructure (CLI) until being superseded by the cross-platform .NET project.

Node package A package is a folder tree described by a package.json file. The package consists of the folder containing the package.json file and all subfolders until the next folder containing another package.json file, or a folder named node_modules.

NodeJS Node.js is a cross-platform, open-source JavaScript runtime environment that can run on Windows, Linux, Unix, macOS, and more. Node.js runs on the V8 JavaScript engine, and executes JavaScript code outside a web browser.

NPM npm is a package manager for the JavaScript programming language maintained by npm, Inc., a subsidiary of GitHub. npm is the default package manager for the JavaScript runtime environment Node.js and is included as a recommended feature in the Node.js installer.

O

Object In software development, an object is an entity semantic that has state, behavior, and identity. An object can model some part of reality or can be an invention of the design process whose collaborations with other such objects serve as the mechanisms that provide some higher-level behavior.

OER Open educational resources (OER) are teaching, learning, and research materials intentionally created and licensed to be free for the end user to own, share, and in most cases, modify. The term "OER" describes publicly accessible materials and resources for any user to use, re-mix, improve, and redistribute under some licenses.

Office Open XML Office Open XML (also informally known as OOXML) is a zipped, XML-based file format developed by Microsoft for representing spreadsheets, charts, presentations and word processing documents.

OLE Object Linking and Embedding (OLE) is a proprietary technology developed by Microsoft that allows embedding and linking to documents and other objects. For developers, it brought OLE Control Extension (OCX), a way to develop and use custom user interface elements.

Open source Open source is source code that is made freely available for possible modification and redistribution. Products include permission to use and view the source code, design documents, or content of the product. The open source model is a decentralized software development model that encourages open collaboration.

OSI The Open Systems Interconnection (OSI) model is a reference model that provides a common basis for the coordination of standards development for the purpose of systems interconnection. In the OSI reference model, the components of a communication system are distinguished in seven abstraction layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application.

P

Pac Man Pac-Man is a 1980 maze video game developed and published by Namco for arcades. The player controls Pac-Man, who must eat all the dots inside an enclosed maze while avoiding four colored ghosts. Eating large flashing dots called "Power Pellets" causes the ghosts to temporarily turn blue, allowing Pac-Man to eat the ghosts for bonus points.

Package A package is a collection of different resources, hierarchically organized and with metadata attached, requiring certain other packages as dependencies. A package has a unique name and version, which is enforced by a registry, through which package can be accessed and dependencies resolved. Packages can be accessed either for reuse by other developers, or by end users. [section 3.1]

Package manager Package managers allow automatically installing and updating packages of registries.

Part-of-speech In grammar, a part of speech or part-of-speech (abbreviated as POS or PoS, also known as word class or grammatical category) is a category of words (or, more generally, of lexical items) that have similar grammatical properties.

Perl Perl is a high-level, general-purpose, interpreted, dynamic programming language. Though Perl is not officially an acronym, there are various backronyms in use, including "Practical Extraction and Reporting Language".

Personalized learning Personalized learning (also named individualized instruction, personal learning place or direct instruction) refers to a type of learning where learners are provided with experiences that uniquely meet their needs, interests and educational outcomes.

PhET PhET Interactive Simulations, a project at the University of Colorado Boulder, is a non-profit open educational resource project that creates and hosts explorable explanations. It was founded in 2002 by Nobel Laureate Carl Wieman.

PHP PHP is a general-purpose scripting language geared towards web development. It was created by Danish-Canadian programmer Rasmus Lerdorf in 1993 and released in 1995. The PHP reference implementation is now produced by the PHP Group. PHP was originally an abbreviation of Personal Home Page, but it now stands for the recursive backronym PHP: Hypertext Preprocessor.

Pragmatics In our learning resource context, the layer of pragmatics deals with who and what a learning resource is intended for. [section 3.2]

Programmed technology Programmed learning (or programmed instruction) is a research-based system which helps learners work successfully. The learning material is in a kind of textbook or teaching machine or computer. The medium presents the material in a logical and tested sequence. The text is in small steps or larger chunks. After each step, learners are given a question to test their comprehension. Then immediately the correct answer is shown.

Programming Computer programming or coding is the composition of sequences of instructions, called programs, that computers can follow to perform tasks. It involves designing and implementing algorithms, step-by-step specifications of procedures, by writing code in one or more programming languages.

PureScript PureScript is a strongly typed, purely functional programming language that transpiles to JavaScript and other targets. It can be used to develop web applications, server side apps, and also desktop applications. Its syntax is mostly comparable to that of Haskell.

Python Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically type-checked and garbage-collected. It supports multiple programming paradigms, including structured, object-oriented and functional programming.

Q

QCA Qualitative content analysis (QCA) is a family of methods for the systematic analysis of content for research purposes. The family contains eight process models which differ in the direction of inferences (inductive/deductive) and how theory is incorporated. It is commonly used to analyze large amounts of natural language data, such as interview transcripts. Deductive content analysis is one of the process models, which emphasizes applying classifications derived from a theoretical background. [section 6.3]

R

R R is a programming language for statistical computing and data visualization. It has been widely adopted in the fields of data mining, bioinformatics, data analysis, and data science.

Reason Reason is a general-purpose, high-level, multi-paradigm, functional and object-oriented programming language and syntax extension and toolchain for OCaml. Reason uses many syntax elements from JavaScript, compiles to native code using OCaml's compiler toolchain, and can compile to JavaScript using the ReScript compiler.

Registry A registry is a storage location for software packages. Often a table of contents is also stored, along with metadata.

Remix workflow The remix of learning resources involves original authors, educators, and learners. Authors create sources, which are composed into resources by educators and further differentiated. They are then shared as instances with learners, who interact with them, creating a learning experience. [section 3.2]

Remixability The degree to which the structure of an interactive element or resource allows it to be reused. [chapter 4]

Repository In version control systems, a repository is a data structure that stores metadata for a set of files or directory structure.

Responsive web design Responsive web design (RWD) or responsive design is an approach to web design that aims to make web pages render well on a variety of devices and window or screen sizes from minimum to maximum display size to ensure usability and satisfaction.

REST REST (Representational State Transfer) is a software architectural style that was created to describe the design and guide the development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave.

Reusability paradox Intentionally generalizing learning resources so they can be reused elsewhere decontextualizes the learning resource, but the value of the resource lies in its contextualization, the degree to which it is fitted to the context. So the more generalized and thus reusable a learning resource is made to be, the less useful and likely to be reused it actually is. Put shortly: Maximizing reusability minimizes usefulness [section 1.2].

Reverse proxy In computer networks, a reverse proxy or surrogate server is a proxy server that appears to any client to be an ordinary web server, but in reality merely acts as an intermediary that forwards the client's requests to one or more ordinary web servers.

Rich text editing Rich text editing systems are usually direct manipulation systems coupled with form fill-in and menu selection for certain interactions. Through the notion of a selection with an anchor and a head, objects in the tree can quickly be grouped and actions performed on that group. With selection, copy & paste alone, syntactical changes can be performed at any granularity. [section 3.2]

RPC In distributed computing, a remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in a different address space of the current process (commonly on another computer on a shared computer network), which is written as if it were a normal (local) procedure call, without the programmer explicitly writing the details for the remote interaction.

Ruby Ruby is a general-purpose programming language. It was designed with an emphasis on programming productivity and simplicity. In Ruby, everything is an object, including primitive data types.

Runtime In computer programming, a runtime system or runtime environment is a sub-system that exists in the computer where a program is created, as well as in the computers where the program is intended to be run.

RUP The Rational Unified Process (RUP) is an iterative software development process framework created by the Rational Software Corporation, a division of IBM since 2003. RUP is not a single concrete prescriptive process, but rather an adaptable process framework, intended to be tailored by the development organizations and software project teams that will select the elements of the process that are appropriate for their needs.

Rust Rust is a general-purpose programming language. It is noted for its emphasis on performance, type safety, concurrency, and memory safety. Rust supports multiple programming paradigms. It was influenced by ideas from functional programming, including immutability, higher-order functions, algebraic data types, and pattern matching.

S

SCORM Sharable Content Object Reference Model (SCORM) is a collection of standards and specifications for web-based electronic educational technology. It defines communications between client side content and a host system, which is commonly supported by a learning management system.

Scratch Scratch is a high-level, block-based visual programming language and website aimed primarily at children as an educational tool, with a target audience of ages 5 to 16. Users can create projects on the website using a block-like interface.

Self editing Core part of the explorable authoring paradigm, where interactive elements may provide affordances for direct manipulation of themselves and their content, and reactively update when their state in the document is changed. [section 7.1]

Semantic versioning Software versioning is the process of assigning unique version names or unique version numbers to unique states of computer software. The most widely adopted[citation needed] scheme for version numbers is known as semantic versioning (SemVer), which comprises a three-part version number (Major.Minor.Patch), an optional pre-release tag (e.g. alpha, beta), and an optional build meta tag.

Serialization In computing, serialization is the process of translating a data structure or object state into a format that can be stored (e.g. files in secondary storage devices, data buffers in primary storage devices) or transmitted (e.g. data streams over computer networks) and reconstructed later.

Serious game A serious game or applied game is a game designed for a primary purpose other than pure entertainment. The idea shares aspects with simulation generally, including flight simulation and medical simulation, but explicitly emphasizes the added pedagogical value of fun and competition.

Service worker Service workers essentially act as proxy servers that sit between web applications, the browser, and the network (when available). They are intended, among other things, to enable the creation of effective offline experiences, intercept network requests, and take appropriate action based on whether the network is available, and update assets residing on the server. [MDN]

SGML The Standard Generalized Markup Language (SGML; ISO 8879:1986) is a standard for defining generalized markup languages for documents.

Shadow DOM The Shadow DOM is a functionality that allows the web browser to render DOM elements without putting them into the main document DOM tree. This creates a barrier between what the developer and the browser can reach; the developer cannot access the Shadow DOM in the same way they would with nested elements, while the browser can render and modify that code the same way it would with nested elements.

Signal Programming pattern for values that vary over continuous time.

Singleton pattern In object-oriented programming, the singleton pattern is a software design pattern that restricts the instantiation of a class to a singular instance.

Snippet Snippets parts of a document. On the web, a snippet would be a HTML fragment which can be inserted into a web document. [section 6.1]

Source In computing, source code, or simply code or source, is human readable plain text that can eventually result in controlling the behavior of a computer.

SPA A single-page application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current web page with new data from the web server, instead of the default method of loading entire new pages (multi-page application or MPA). The goal is faster transitions that make the website feel more like a native app.

Space Invaders Space Invaders is a 1978 shoot 'em up video game developed for arcades. Space Invaders was the first video game with endless gameplay and the first fixed shooter, setting the template for the genre. The goal is to defeat waves of descending aliens with a horizontally moving laser cannon to earn as many points as possible.

Static Static content includes formatted text with rich content such as images and mathematical formulas. As a rule of thumb, all printable content is static. [section 3.2]

Static program analysis In computer science, static program analysis (also known as static analysis or static simulation) is the analysis of computer programs performed without executing them, in contrast with dynamic program analysis, which is performed on programs during their execution in the integrated environment.

STEM Science, technology, engineering, and mathematics (STEM) is an umbrella term used to group together the related technical disciplines of science, technology, engineering, and mathematics. It represents a broad and interconnected set of fields that are crucial for innovation and technological advancement.

Structured document A form of representation where documents are not simply a sequence of characters, but a tree of objects, which are in turn made up of simpler objects. [section 3.2]

SUS The System Usability Scale (SUS) is a standardized, single scale of 10 items to measure the usability of a system. [section 5.2]

SVG Scalable Vector Graphics (SVG) is an XML-based vector graphics format for defining two-dimensional graphics, having support for interactivity and animation.

Syntax In our learning resource context, the layer of syntax deals with the composition of units into a larger whole, and the relationship of units with each other. [section 3.2]

T

TCP/IP The Internet protocol suite, commonly known as TCP/IP, is a framework for organizing the communication protocols used in the Internet and similar computer networks according to functional criteria. The foundational protocols in the suite are the Transmission Control Protocol (TCP), the User Datagram Protocol (UDP), and the Internet Protocol (IP).

TEI-XML The TEI Guidelines collectively define a type of XML format. It differs from other well-known open formats for text (such as HTML and OpenDocument) in that it is primarily semantic rather than presentational: the semantics and interpretation of every tag and attribute are specified.

TeX TeX is a typesetting program which was designed and written by computer scientist and Stanford University professor Donald Knuth and first released in 1978.

The Oregon Trail The Oregon Trail is a series of strategy computer games. The original game was designed to teach eighth grade schoolchildren about the realities of 19th-century pioneer life on the Oregon Trail.

Theme Themes are stylesheets to be applied to a whole document. On the web, a theme would be a CSS stylesheet which can be included in a web document. [section 6.1]

Transaction processing In computer science, transaction processing is information processing that is divided into individual, indivisible operations called transactions. Each transaction must succeed or fail as a complete unit; it can never be only partially complete.

TS TypeScript (TS) is a high-level programming language that adds static typing with optional type annotations to JavaScript. It is designed for developing large applications. It transpiles to JavaScript.

Turing completeness In computability theory, a system of data-manipulation rules (such as a model of computation, a computer's instruction set, a programming language, or a cellular automaton) is said to be Turing-complete or computationally universal if it can be used to simulate any Turing machine.

U

UEQ The User Experience Questionnaire (UEQ) is a standardized questionnaire with six subscales to measure the usability of a system. For more specific applications, a modular framework to construct questionnaires based on the UEQ is available, named UEQ+. [section 5.2]

UI In the industrial design field of human–computer interaction, a user interface (UI) is the space where interactions between humans and machines occur. The goal of this interaction is to allow effective operation and control of the machine from the human end, while the machine simultaneously feeds back information that aids the operators' decision-making process.

UNESCO Institute for Statistics The UNESCO Institute for Statistics (UIS) is the statistical office of UNESCO and is the UN depository for cross-nationally comparable statistics on education, science and technology, culture, and communication.

URL A uniform resource locator (URL), colloquially known as web address, is a reference to a resource on the World Wide Web. A URL specifies the location of a resource on a computer network and a mechanism for retrieving it.

Usability A quality attribute that assesses how easy user interfaces are to use with components such as learnability, efficiency, memorability, low error rate with easy recovery, and satisfaction (see section 1.3).

UUID A universally unique identifier (UUID) is a 128-bit number designed to be a unique identifier for objects in computer systems. UUIDs are designed to be large enough that any randomly-generated UUID will, in practice, be unique from all other UUIDs.

UX User experience (UX) is how a user interacts with and experiences a product, system, or service. It includes a person's perceptions of utility, ease of use, and efficiency.

V

Version control system A version control system is a software tool that automates version control. Version control (also known as revision control, source control, and source code management) is the software engineering practice of controlling, organizing, and tracking different versions in history of computer files.

VHS VHS (Video Home System) is a discontinued standard for consumer-level analog video recording on tape cassettes, introduced in 1976 by JVC. It was the dominant home video format throughout the tape media period of the 1980s and 1990s.

Visual Basic Visual Basic (VB), sometimes referred to as Classic Visual Basic, is a third-generation programming language based on BASIC, as well as an associated integrated development environment (IDE).

VR Virtual reality (VR) is a simulated experience that employs 3D near-eye displays and pose tracking to give the user an immersive feel of a virtual world. Applications of virtual reality include entertainment (particularly video games), education (such as medical, safety, or military training), research and business (such as virtual meetings).

W

W3C The World Wide Web Consortium (W3C) is the main international standards organization for the World Wide Web. Founded in 1994 by Tim Berners-Lee, the consortium is made up of member organizations that maintain full-time staff working together in the development of technical standards for the World Wide Web.

Web 2.0 Web 2.0 (also known as participative web and social web) refers to websites that emphasize user-generated content, ease of use, participatory culture, and interoperability (i.e., compatibility with other products, systems, and devices) for end users.

Web component Web Components are a set of features that provide a standard component model for the web allowing for encapsulation and interoperability of individual HTML elements. Web Components are a popular approach when building microfrontends.

Web view A WebView is a web browser that is embedded within an app. Thus a WebView is a large-scale software component, enabling the use of web content within apps.[1] In some cases, the entire functionality of the app is implemented this way.

Web worker Web Workers makes it possible to run a script operation in a background thread separate from the main execution thread of a web application. The advantage of this is that laborious processing can be performed in a separate thread, allowing the main (usually the UI) thread to run without being blocked/slowed down. [MDN]

WebAssembly WebAssembly (Wasm) defines a portable binary-code format and a corresponding text format for executable programs as well as software interfaces for facilitating communication between such programs and their host environment.

WHATWG The Web Hypertext Application Technology Working Group (WHATWG) was founded by representatives from Apple Inc., the Mozilla Foundation and Opera Software, leading web browser vendors in 2004. WHATWG is responsible for maintaining multiple web-related technical standards, including the specifications for the HyperText Markup Language (HTML) and the Document Object Model (DOM).

Widget The design pattern of widgets wraps components in packages to achieve FAIRness, enabling flexible reuse for code projects, through authoring tools, or in (web) documents. [section 6.1]

Wikitext Wikitext, also known as wiki markup or wikicode, is the markup language used by the MediaWiki software to format pages.

WYSIWYG In computing, WYSIWYG (what you see is what you get) is software that allows content to be edited in a form that resembles its appearance when printed or displayed as a finished product, such as a printed document, web page, or slide presentation. WYSIWYG implies a user interface that allows the user to view something very similar to the result while the document is being created.

X

XAPI The Experience API (xAPI) is an e-learning software specification that records and tracks various types of learning experiences for learning systems. Learning experiences are recorded in a Learning Record Store (LRS), which can exist within traditional learning management systems (LMSs) or on their own.

Xerox PARC Future Concepts division (formerly Palo Alto Research Center, PARC and Xerox PARC) is a research and development company in Palo Alto, California. It was founded in 1969 by Jacob E. "Jack" Goldman, chief scientist of Xerox Corporation, as a division of Xerox, tasked with creating computer technology-related products and hardware systems.

XLIFF XLIFF (XML Localization Interchange File Format) is an XML-based bitext format created to standardize the way localizable data are passed between and among tools during a localization process and a common format for CAT tool exchange.

XML Extensible Markup Language (XML) is a markup language and file format for storing, transmitting, and reconstructing data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

Z

ZIP ZIP is an archive file format that supports lossless data compression. A ZIP file may contain one or more files or directories that may have been compressed. The ZIP file format permits a number of compression algorithms, though DEFLATE is the most common.

Acknowledgements

Acknowledgements are probably boring for most readers, but I hope they will delight a few select ones.

First of all, I want to thank my parents, Reinhild and Karlo, my sister Karolin, and my partner Marie. Simply, I am who I am because of you all, so my achievement is your achievement. You also dealt with me through the whole writing process, which cannot be understated. I also thank all my wonderful close friends, who supported me and never took me too seriously.

Professionally, I am very grateful to my advisor, Ulrik Schroeder. For a dissertation, this is a mandatory "thank you", but for me, this comes easily and from the heart – I was granted all the freedom, but also all the support and resources to pursue my project. You shaped the project and gave me the confidence to complete it, but you never interfered. The team you built at the Learning Technologies Research Group, past and present, is just amazing and I am grateful to them, as well! Our discussions and collaborations all became a part of this result. I appreciated everyone's support throughout, and especially want to highlight René Röpke's mentorship, the technical discussions with Sergej, and all the delicious cake served over the years. This includes the work of my student assistants over the years (thank you Jorik, Robin, Peter, Constantin, Johannes, and Nura), and the work of the many students who decided to complete their theses under my supervision. I also thank my second examiner Niels Pinkwart for devoting so much time to someone else's work! Curiously, I need to thank Carsten Schulte, professor in Paderborn, who a few years ago said "you can't do that topic at our group, but there might be a place in Aachen for it...". This is the best *no* I have received so far.

I have to thank everyone taking the time to read this dissertation in its draft form, since my sentences can be long-winded. With your help, I hope that the text has become readable. Thanks go out to Alex (also my partner in crime for the SCOUT project, and the source of endless funny distractions sorely needed), to my sister Karolin and partner Marie, to my colleagues Sergej and Jasper, to my good friend Lars, and many more who provided feedback.

Finally, I need to thank the state of North-Rhine Westphalia which supported my work through research project SCOUT in the consortium NERD II (005-2201-0014).