Visual Artifacts in Software Engineering Processes

Hans-Werner Sehring
Department of Computer Science
Nordakademie
Elmshorn, Germany
e-mail: sehring@nordakademie.de

***Abstract*—Generating software solutions from (formal) models in Model-Driven Software Engineering (MDSE) approaches has many advantages, for example, model checking and traceability. Development processes that include a substantial amount of creative work rely on a manual creation of visual artifacts, and it is not feasible to work purely with formal models. As a first step towards the inclusion of graphical artifacts into MDSE, we propose two approaches: (1) Describe informal artifacts by formal model elements so that content of artifacts is related to models. (2) Generate artifacts from formal descriptions so that models become perceivable by domain experts. While the latter is not generally possible since visual artifacts are required as part of a creative process, this approach is applicable to prototyping. This way, testable designs are coherent with formal specifications.**

***Keywords*—Creative Process; Software Engineering; Software Development Processes; Model-driven Software Engineering**

I. Introduction

Designing software solutions from (formal) models and finally generating working software from these models in *Model-Driven Software Engineering* (*MDSE*) or *Model-Driven Software Development* (*MDSD*) approaches has many advantages. For example, they allow model checking of intermediate results, traceability of the results of the modeling steps [1], (partial) automation, etc.

There is class of software applications, though, that contains a substantial amount of creative work that cannot be captured by formal models for three possible reasons: (1) artifacts that result from a development step are informal in nature, for example, a graphical design sketch, (2) the work on the artifacts cannot be automated because of the creativity involved, or (3) the artifacts are preferably created using tools that do not fit into the model-driven tool chain, for example, visual prototyping tools.

In particular, graphical artifacts that provide descriptions of the solution play an important role in applications with user interaction like graphical clients, web-based systems, and mobile applications. These may be dynamic in nature. For example, prototypes exemplify the interaction with the software solution that is being developed under certain use cases and the navigation between parts of the software from a user interface perspective.

Prototypes basically provide a formal description since they consist of code. But code in itself is too expressive as to serve as a description since it cannot automatically be checked for completeness, consistency, etc.

In order to gain from MDSE advantages in creative software engineering processes, we are currently investigating means of managing visual artifacts into MDSE. To this end, model elements both serve as descriptions of visual artifacts and as formalizations that can be related to each other, checked, transformed, etc. Optimally, descriptions from creative tools can automatically be read in and related to formal descriptions, and artifacts can be updated from model elements.

In this paper, we present experiments with modeling two kinds of visual artifacts: models that can automatically be related to visual artifacts and thus co-exist with those artifacts, and models that are used to continuously generate artifacts in order to visually represent models.

We revisit model-driven as well as creative software development processes in Section II. In Section III, we introduce the Minimalistic Meta Modeling Language (M³L) as a basis for our proposal for managing visual artifacts. We study the two approaches that are proposed in this paper in Section IV. The paper concludes in Section V.

II. Artifacts in Software Engineering Processes

Software engineering processes consist of the generation of a series of artifacts, each describing the problem domain, the requirements to be addressed, or the solution.

Approaches differ in the degree of formalization of such artifacts and, therefore, in the degree of automation. Here, we specifically consider model-driven software engineering and software development processes that incorporate creative tasks in order to derive some requirements to their support.

A. Model-Driven Software Engineering

The artifacts created in *MDSE* processes are formal models that are derived from each other by means of model refinement and model transformations.

Figure 1 illustrates an MDSE process. It shows different modeling stages that correspond to different phases, here inspired by the *Model-Driven Architecture* (*MDA*) approach of the OMG [2]. A domain model represents the solution from the perspective of the subject domain, as domain concepts or requirements. In MDA, this was originally called a *Computation-Independent Model* (*CIM*; this term is not used in current specifications). It typically is an informal description, for example, done in natural language. A first formal model is a *Platform-Independent Model* (*PIM*) that gives a conceptual description of the software solution. It is transformed into a *Platform-Specific Model* (*PSM*) that adds implementation aspects. This model is used to generate a working implementation.

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-159-6

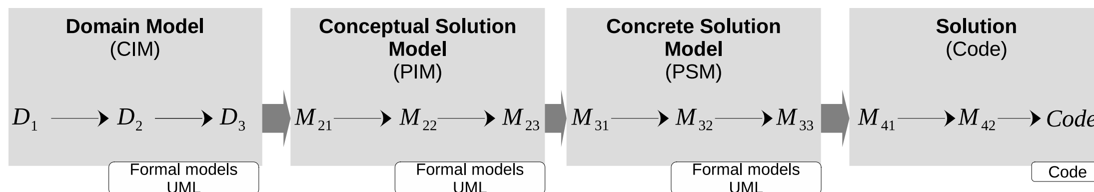

Figure 1. A typical MDSE / MDSD process.

In general, we see the typical stages of domain model, conceptual software model, concrete software model, and code.

The descriptions of the PIM D_i and the models M_{ij} of the other stages in the figure indicate the (formal) artifacts created in the various stages. Within one stage, models are refined until the next stage can be started. Models of a subsequent stage provide a description from a different perspective, but they make reference to the models of the preceding stage. *Code* is a formal model from which the working solution is automatically built.

In MDSE approaches, models have to be formal in order to be subject to transformations and to reference (parts of) each other. In the MDA, for example, model transformations are specified using *Query View Transform* based on MDA's *Meta Object Facility* [3].

B. Model-Assisted Software Creation

For some kinds of software implementations, practical processes are based on artifacts that cannot fully be formalized. This depends on the application domain and is particularly true for the development of interactive applications like websites or other web-based information systems and mobile apps.

The creation of such applications incorporates creative tasks. These are typically not just amending other development tasks by adding creative input to the use and presentation of an application. On top of that, the artifacts created in creative tasks are used to communicate ideas in the course of the process, they provide a conceptual domain model from a user perspective, and they provide prototypes for testing purposes.

Creative tasks are typically manual steps that involve creativity in generating artifacts and subjective opinions in tests. As such, they do not follow a formal notation. Instead, they are often either textual or graphical descriptions. Though certain communication principles are used, they cannot be processed automatically.

Figure 2 shows typical development steps and artifacts created to model aspects of a solution. On a conceptual level, requirements are studied from the users' perspective. Resulting artifacts serve to illustrate and verify the first insights, for example, by illustrating target groups as *personas* or typical usage patterns as *customer journeys* along which users interact with a software system at different *touchpoints*. These lead to a characterization of the general solution approach, the *solution hypothesis*.

First solution aspects are created conceptually in order to validate the solution hypothesis, for example, by studying the consistency of specific parts of the solution. Feedback can be collected from users by first visual impressions, like *wireframes*, and from software developers by estimating the implementation effort, for example, based on a *catalog of graphical modules*.

More elaborate prototypes, sometimes called high-fidelity prototypes, also address visual design aspects of the solution. This way, they allow test users to get a good impression of the final software and to verify the solution approach, or to identify points that need improvement.

Based on the insights from the creative phases, the more technical considerations of software engineering follow. In processes that are based on creative activities, software aspects should be derived from the previous artifacts that were created from a user perspective. For example, the data needed at certain touchpoints and the functionality that has to be provided in specific contexts are derived from customer journeys. This is done in a phase *solution architecture* that provides an abstract specification that brings together perspectives of users, designers, software developers, and others.

C. Conceptual Software Description Artifacts

Many of the various artifacts that are generated in creative software engineering processes are informal. This means that they are not expressed in a formal language, and that they do not have formal semantics attached to them. Instead, they are targeted at human perceivers that interpret the artifacts in a subjective way.

From the many artifacts, we consider screen designs and the user interactions that lead to different presentations and interactions. These allow test users to get an impression of they way the final software will work. An artifact that serves this purpose is a *click dummy*.

Figure 3 illustrates a flow between different *dialogs* and *view states*. A dialog is a means of interaction, for example, a user interface window, a webpage, or a screen of a mobile app. A view state is one of the states a dialog can be in, for example, a search dialog in initial form, after a search has been executed, and after the search result has been refined.

Storyboards are in wide-spread use as a metaphor for such flows of dialogs [4]. These can be used in a creative fashion, but also be formalized [5].

In processes that center around the user and that incorporate creative activities, it is more common to use working software a user can interact with. There is no actual functionality underlying such software. But it gives test users the impression of working software so they can judge whether the final

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-159-6

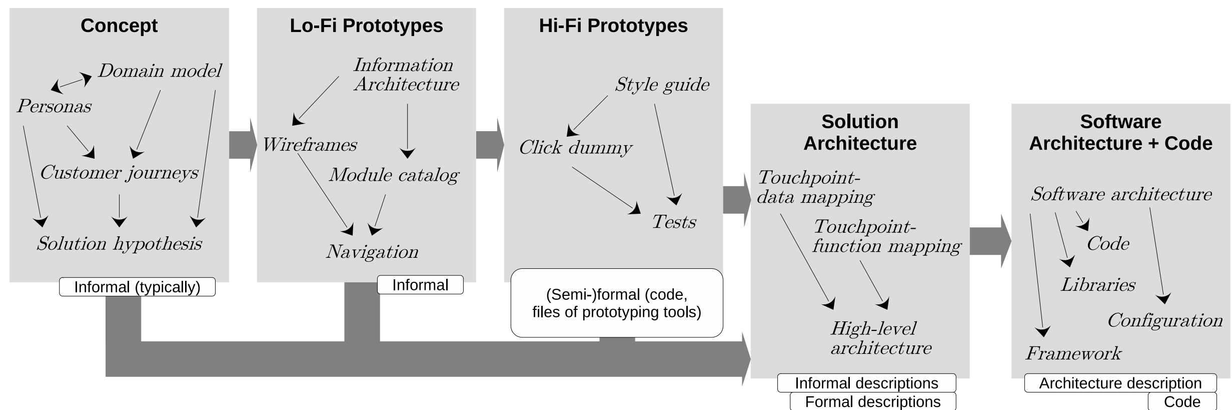

Figure 2. A typical software development process that integrates creative activities.

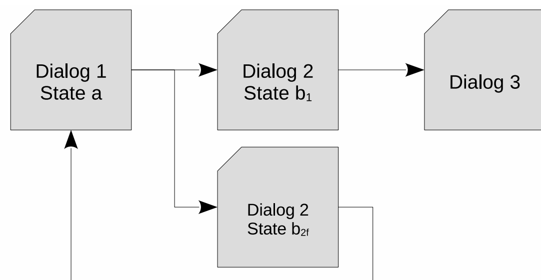

Figure 3. A sample flow of dialogs and dialog states.

software works as expected [6]. To this end, it simulates changes between dialogs and changing the status of a dialog.

In practice, such shallow prototypes are created as configurations in prototyping software tools. In such cases, the prototypes do not evolve into working software. They are maintained as singular artifacts that can later be analyzed.

In an MDSE scenario, prototypes can be generated from evolving models, though. In this case, they directly reflect the current modeling state. Models can be updated based on user feedback, and then a new revision of the prototype can be generated. This way, work on the client does not only lead to improved prototypes, but it directly leads to an improved model.

III. Minimalistic Meta Modeling Language (M³L)

For the discussion in this paper, we use the M³L since it proved to be a suitable language for the modeling of content of various artifacts. To this end, we briefly introduce the M³L, and we present some exemplary base models for the representation of artifacts.

A. Basic M³L Constructs

In this section, we briefly introduce the M³L by highlighting those features that are central to the underlying experiments.

The basic M³L statements are:

- `A`: the declaration of or reference to a *concept* named *A*
- `A is a B`: refinement of a concept *B* to a concept *A*. *A* is a *specialization* of *B*, *B* is a *generalization* of *A*.
- `A is a B { C }`: containment of concepts. *C* belongs to the *content* of *A*, *A* is the *context* of *C*.
- `A |= D`: the *semantic rule* of a concept. Whenever *A* is referenced, actually *D* is bound. If *D* does not exist, it is created in the same context as *A*.
- `A |- E F G .`: the *syntactic rule* of a concept that defines how a string is produced from a concept, respectively how a concept is recognized from a string. When the representation of A is requested, it is produced by a concatenation of the strings produced out of E, F, and G. When no syntactic rule is defined, a concept is represented by its name. Vice versa, an input that constitutes the name of a concept without a syntactic rule leads to that concept being recognized.

If a concept that is referenced by one of the statements exists or if some other concept evaluates (see below) to the reference, then this one is bound. Otherwise, the concept is created as defined by the statement.

Existing concepts can be redefined. For example, with the definitions above, a statement

```
A is an H { C is the I }
```

redefines A to have another generalization H and C (in the context of A) to be the only specialization of I.

Every context constitutes a *scope*. A redefinition of a concept in a context is only applied in that context. When a redefinition of a concept takes place in another context as the original definition, we call that redefinition a *derivation*.

Scopes also define *visibility*. A concept is visible in the context it is (re-) defined in, and all contained contexts. Concepts from foreign contexts can be made visible with

```
ForeignConcept from ForeignContext
```

B. Concept Evaluation

The concepts that are defined by such statements are *evaluated* when used. Evaluation means looking up or creating concepts and applying semantic rules. This is done in the following form:

1) Evaluation is performed with respect to a context from which concept definitions are taken.

2) The effective definition of a concept in some context is the set of all definitions in that context and its surrounding base contexts (transitive).
3) A concept (transitively) inherits base concepts and content from its base concepts
4) A concept A is a derived base concept of a concept B if B *matches* A. Matching is defined as:
   - A and B share a common base concept (or A has none).
   - For every content C of A there exists content D of B where D *matches* C.
5) Let EC_A be the union of all base concepts and all derived base concepts of a concept A. If a concept in EC_A has a defined or inherited semantic rule, then the result of this rule defines candidates for the evaluation of A. If no concept in EC_A carries a semantic rule, then the whole set forms the set of candidates.
6) Finally, all candidate concepts are *narrowed down*: for each concept, the most specific matching subconcept is added to the result.

C. M³L Example

For a simple modeling example, consider the following M³L statements from the area of programming language design:

```
Boolean is a Type
Statement
IfThenElse is a Statement {
 Condition is a Boolean
 ThenStatement is a Statement
 ElseStatement is a Statement }
IfTrue is an IfThenElse {
 True is the Condition } |= ThenStatement
IfFalse is an IfThenElse {
 False is the Condition }|= ElseStatement
```

These describe the behavior of a conditional statement as it is found in typical imperative programming languages.

Adding an additional statement in order to provide a concrete program,

```
MyCond {
 ConceptEvaluatingToBool is the Condition
 SomeStatement is the ThenStatement
 SomeOtherStatement is the ElseStatement}
```

leads to the following evaluation: depending on the evaluation result of *ConceptEvaluatingToBool*, either *IfTrue* or *IfFalse* is a derived base concept of *MyCond* (shared base concept *Statement* and matching Content *True*). Thus, *MyCond* inherits the semantic rule from this derived base concept, making it evaluate to the right conditional branch. Candidates are either *ThenStatement* or *ElseStatement* that are finally narrowed down to *SomeStatement* or *SomeOtherStatement*, respectively.

IV. Managing Models of Visual Artifacts

To allow some of the creative artifacts to be integrated into a model-driven development process, we propose to define models that reflect the content of those artifacts. These models allow handling the representations in model transformation processes. This way, for example, traceability between models and thus between artifacts is achieved. It can be made explicit, which artifacts provided input to which other artifacts.

We see two basic approaches that are handled by the following two subsections: artifacts that are entities in their own right where models are representing the current state of the artifacts, and artifacts that are generated from models that provide one aspect of the development process.

A. Co-existence of Artifacts and Models

The M³L is universal and has many applications. Among other modeling tasks, it has proven useful for describing content and creating documents from content [7]. This applies both to content models, as well as content items since the M³L does not distinguish model layers, such as type and instance.

Using the M³L to define a content management structure, the content of artifacts can be maintained in models and be represented by documents generated out of the models. Changes to documents can be re-read in order to update the content concepts.

For an example of structured content, with a content model like:

```
UserStory is a Content {
 Title is a String
 Text is a FormattedString
 StoryPoints is a FibonacciNumber }
```

according content can be created:

```
UserStory123 is a UserStory {
 "As a sales person I ..." is the Title
 "When a sales person ..." is the Text
 8 is the StoryPoints }
```

For presentation purposes, content is added to document descriptions. For example, when creating text documents, there might be definitions like:

```
PublishedDocPart { Content }
UserStoryDetailPage is a PublishedDocPart{
 Content is a UserStory }
WebPage is a PublishedDocPart {
 Title is a String }
PrintDocumentPage is a PublishedDocPart {
 PageNumber is a Number }
```

For textual formats, like HTML and JSON, documents can be rendered from concepts through syntactic rules of content as introduced in the previous section. On the level of a content model, syntactic rules describe document templates, on the content item level they render single document instances.

For the sample content definitions above, an HTML template may look like outlined in Figure 4. In this example, there are some basic definitions for HTML code generation (or code parsing) in the context *HTML*. These include the redefinition of the concept *WebPage* from above with a syntactic rule for HTML generation. The sample code sketches a simple skeleton of HTML code.

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-159-6

```
HTML {
 <html> </html> <head> </head>
 <title> </title> <body> </body> ...
 WebPage
 |- <html>
     <head> <title> Title </title> </head>
     <body> Content </body>
    </html> . }
UserStoryHTML is an HTML {
 UserStoryDetailPage {
  Title is the Title from Content
  Content {
   Title |- "<h1>" Title "</h1>" .
   Text |- "<p>" Text "</p>" .
   StoryPoints |- "SP: " StoryPoints .
  } |- Title StoryPoints Text . }
```

Figure 4. Sketch of an HTML publication model.

Also, some tags for HTML elements are defined to indicate that several basic definitions will be provided here. Please note that, e.g., *<html>* is a valid concept name. Since new concepts are declared the first time they are referenced, and because they syntactically evaluate to their name by default, they can also be used in all syntactic rules like string literals.

The particular HTML generation of representations of user stories is defined in a subcontext *UserStoryHTML*. Here, the different parts of a user story's content are equipped with syntactic rules that produce (or recognize) page fragments.

HTML for a webpage that represents a user story is generated by creating a *UserStoryDetailPage* with a specific user story as content:

```
MyUserStoryPage is a UserStoryDetailPage {
 SomeUserStory is the Content }
```

The syntactic rule of *WebPage* from *HTML* is inherited by *UserStoryDetailPage*. Therefore, the page skeleton will be generated, and *Content* is embedded. For *Content*, a syntactic rule is defined in the context of *UserStoryDetailPage* that triggers the syntactic rule of the user story's content.

Manual changes to an HTML page can be re-read by the syntactic rules, leading to the M³L concepts being updated.

B. Generation of Artifacts from Models

Artifacts can be generated from models in many cases, but changes to the artifacts cannot automatically be re-engineered to models. In such scenarios, we can use "micro-MDSE" processes that drive the creation of but one artifact. Eventually, several such processes are running for co-evolving artifacts.

As an example, we consider click dummies in this section. Click dummies are shallow prototypes that give an impression of the final software product by offering a consistent part of a User Interface (UI) to test users. When click dummies are generated from models, any changes to the UI because of user feedback will be applied by reworking those models. This way, UI designs become manageable in MDSE processes.

```
UIElement { Id }
VisibleUIElement is a UIElement {
 Dimension is a ...
 ... }
Container is a UIElement {
 Components is a UIElement }
TextField is a VisibleUIElement {
 Text is a String }
```

Figure 5. Sketch of a base model of static UI elements.

```
ActivatableUIElement is a UIElement {
 ActionHandler }
Button is a VisibleUIElement,
          an ActivatableUIElement {
 Label is a String
 Icon is an Image }
ActionHandler {
 Condition is a Boolean
 Effect }
ExecutedActionHandler {
 True is the Condition
} |= Effect
UIEvent {
 Target is an ActivatableUIElement
} |= ActionHandler from Target {
       True is the Condition }
   |- "{" "\"target\"" ":"
          "\""Id from Target"\"" "}"
ClickEvent is a UIEvent
```

Figure 6. Sketch of a base model of dynamic UI elements.

The considerations of model-based UI generation are based on earlier work on the topic [8]. In the meantime, quite some related work emerged [9].

1) Base UI Model: Static aspects of a UI consist of a definition of UI components and possible combinations of them. Figure 5 outlines an example.

UIElement is a base definition of all UI components. Visible UI elements are categorized as *VisibleUIElement* that adds some parameters required to draw a component. A *Container* is an example of an invisible component that defines a UI layout.

To relate input to components, we assume that every UI component carries an ID. Alternative approaches might, for example, use a component's path in the UI layout hierarchy.

2) Dynamic UI Aspects: To define the dynamic behavior of a UI, user interactions have to be modeled. A UI description needs a runtime environment that visualizes the defined UI elements, and that handles user input. For an example, assume a runtime environment that hands over events as JSON strings, for example, for mouse clicks, and that receives UI updates by marshaled/HTML components.

Events are handled by *ActionHandler*s as lined out in Figure 6. Dynamic UI components are *ActivatableComponent*s

```
SearchDialog is a UIModel {
 SearchView is a Panel {
  1 is the Id
  QueryField is a TextField
  SearchButton is a Button {
   2 is the Id
   Search is the Label }
  ...
} } |- SearchView
SearchDialogInitial is a SearchDialog {
 SearchView {
  SearchButton {
   Action1 is an ActionHandler {
    "manyres" is the Text from QueryField
    SearchDialogManyResults is the Effect}
   Action2 is an ActionHandler {
    "fewres" is the Text from QueryField
    SearchDialogFewResults is the Effect }
} } }
SearchDialogManyResults is a SearchDialog{
 ... }
SearchDialogFewResults is a SearchDialog {
 ... }
```

Figure 7. Example of views and view transitions.

that may have a handler. A handler that shall execute its action is activated by its *Condition* set to true, making it become a derived subconcept of *ExecutedActionHandler* and its *Effect* to be the result.

JSON messages from the environment are parsed using the syntactic rule of *UIEvent*. The *Id* of a resulting *UIEvent*'s *Target* is set to the ID contained in the JSON message. As a consequence, *Target* evaluates to the actual target component and the target's *Action* is activated.

3) Example of a User Interaction Model: Figure 7 shows a sample application of such a UI model. When an event is sent to the *SearchButton*, the *Condition* of all action handlers, here *Action1* and *Action2*, is set to *True*. When evaluating *ActionHandler* in the semantic rule of *UIEvent*, it evaluates to those actions for whom also the text of the query field matches. This way, different paths through the sequence of dialogs are selected by test users. If none is applicable, the event just evaluates to the target's base *ActionHandler* that has no defined *Effect*. The *Effect* of the search actions (*Action1* and *Action2*) is a new *SearchDialog*, that "replaces" the current one. The assumed surrounding runtime environment is responsible for rendering the new search dialog through the syntactic rule defined for *SearchDialog*.

V. CONCLUSION

We conclude with a summary and an outlook.

A. Summary

We presented two approaches to the management of informal artifacts by formal modeling elements. The first experiment demonstrates the possibility to generate textual descriptions from abstract descriptions by using typical content management functionality.

Prototypical implementations can be generated the same way as the final software solution. A second experiment showed that this can be done in the same modeling framework as the first experiment.

Therefore, potentially different ways of including creative work can be combined. For example, a graphical design specification can be created using the first approach of coevolving documentation and models, and can finally be used to related to a prototype description on the level of models.

B. Outlook

Additional research is need to fully investigate round-trip engineering that consists of alternating (manual) creative and (automated) modeling steps that operate on the same set of artifacts.

Implementation work, for example extending the M³L, is required to include other formats of visual artifacts into the process so that models do not prescribe the form of those artifacts. With the incorporation of such formats, practical processes in which proprietary design tools are used can be subject to further experiments.

ACKNOWLEDGMENT

The author thanks NORDAKADEMIE gAG for funding of this work. Various colleagues from the fields of user experience design and software engineering helped define the initial experiments.

REFERENCES

[1] I. Galvao and A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineering," Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, 2007, pp. 313-313.

[2] Object Management Group. *Model Driven Architecture (MDA)*, MDA Guide rev. 2.0, OMG Document ormsc/2014-06-01, [Online] Available from: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf. 2024.2.10.

[3] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, OMG Document Number formal/2008-04-03, [Online] Available from: https://www.omg.org/spec/QVT/1.0/PDF/. 2024.2.10.

[4] C. van der Lelie, "The value of storyboards in the product design process," Personal and Ubiquitous Computing, volume 10, pp. 159–162, 2006.

[5] K.-D. Schewe and B. Thalheim, "Storyboarding," Design and Development of Web Information Systems, Springer, pp. 61–109, 2019.

[6] S. Böhm and S. Graser, "AI-based Mobile App Prototyping: Status Quo, Perspectives and Preliminary Insights from Experimental Case Studies," Proceedings Sixteenth International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services, ThinkMind, 2023, pp. 29–37.

[7] H.-W. Sehring, "On Integrated Models for Coherent Content Management and Document Dissemination," Proceedings of the Thirteenth International Conference on Creative Content Technologies, ThinkMind, 2021, pp. 6–11.

[8] H.-W. Sehring, "Adaptable and adaptive visualizations in concept-oriented content management systems," International Journal on Advances in Software, volume 3, numbers 1 and 2, ThinkMind, pp. 265–279, 2010.

[9] J. C. Mejias, N. Silega, M. Noguera, Y. I. Rogozov, and V. S. Lapshin, "Model-Driven User Interface Development: A Systematic Mapping," Proceedings of the 8th Iberoamerican Workshop on Human-Computer Interaction, Springer, 2022, pp. 114–129.