Integrating Creative Artifacts into Software Engineering Processes

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

***Abstract*—Model-driven software engineering processes are based on formal models that are automatically transformed into each other. Many software development approaches involve creative activities that result in manually generated and informal documents that prevent automatic model transformations. The content of these documents must be accessed in a structured way to enable transformation steps. Manually maintained documents are subject to frequent changes, including modifications of their structure. To enable model-driven processes in the presence of creative activities and their documents, we are currently experimenting with parsing techniques that combine the structure of documents with domain knowledge about their content. First experiments are based on the Minimalistic Metamodeling Language and its ability to integrate semantic descriptions with syntactic representations.**

***Keywords*-software development; software engineering; computer aided software engineering; top-down programming; document handling.**

I. Introduction

Software engineering processes involve the creation and consumption of a series of documents. Such documents link different phases of activity in software creation processes, be they sequential work performed by experts in phase-oriented projects or simultaneous cooperation in cross-functional teams in agile approaches.

A class of software engineering processes that are based on documents that contain formal models are called *Model-Driven Software Engineering* (*MDSE*) or *Model-Driven Software Development* (*MDSD*) processes.

Some software engineering processes include creative activities, such as conceptual modeling or interaction design [1]. Such creative actitivies are supported by documents that have neither a common format [2] nor formal semantics. Instead, they reflect subjective impressions, case-based presentations, alternatives, and similar content directed at a human audience.

Documents that lack formal structure cannot participate in MDSE processes per se. However, they can be annotated by their creators with, for example, with references to relevant content that are sufficiently fine-grained to address well-formed content. Such annotations allow creative documents to participate in MDSE processes.

However, such annotations refer to specific document instances. Documents used in creative activities are, in particular, working documents that are subject to constant change. This includes changes in the structure of the documents. Therefore, any fixed reference to content in such a document will potentially become invalid and metadata may become inconsistent as work progresses.

In this paper, we investigate means of integrating informal documents, in particular ones that are subject to change, into (model-driven) software engineering processes. We are currently experimenting with linguistic means of recognizing the content of documents with changing structures. First experiments with document recognition are based on a modeling language and its special ability to integrate semantic descriptions with syntactic representations.

Preliminary results show that at least some content can be extracted from documents that lack formal representations. In this way, model-driven approaches can potentially be applied to software projects with creative aspects.

The remainder of this paper is organized as follows: In Section II, we revisit model-driven software engineering and discuss the need for incorporating informal documents. Section III presents typical ways of referencing content in single documents, and it addresses means of managing volatile references to content of mutable documents. Section IV briefly introduces a modeling language that is used for initial experiments in this paper. An experimental implementation of these concepts is presented in Section V. The paper concludes in Section VI with a summary and an outlook on future work.

II. Visual Software Engineering Artifacts

The discourse in this paper does not require a comprehensive introduction to model-driven approaches. However, this section introduces some basic terms and highlights the challenges of integrating creative work.

A. Model-Driven Software Engineering

In software development processes, a series of documents is created. The kinds of documents may differ depending on the kind of software being created and on the methodology used for the process. But all documents serve common purposes, such as linking activities by the results represented in them, allowing traceability of activities [3], and others.

MDSE formalizes the flow of documents and thus the connection of development steps. Documents are *models* with a formal semantics. Models are derived by means of *model-to-model transformations* and finally to code in *model-to-text transformations* on a (semi-) automatic basis. This way, development steps can be performed (semi-) automatically and changes to models can be propagated down the model chain.

One of the first prominent examples of MDSE is the Object Management Group's Model-Driven Architecture (MDA). Various other approaches have emerged that differ in the way in

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

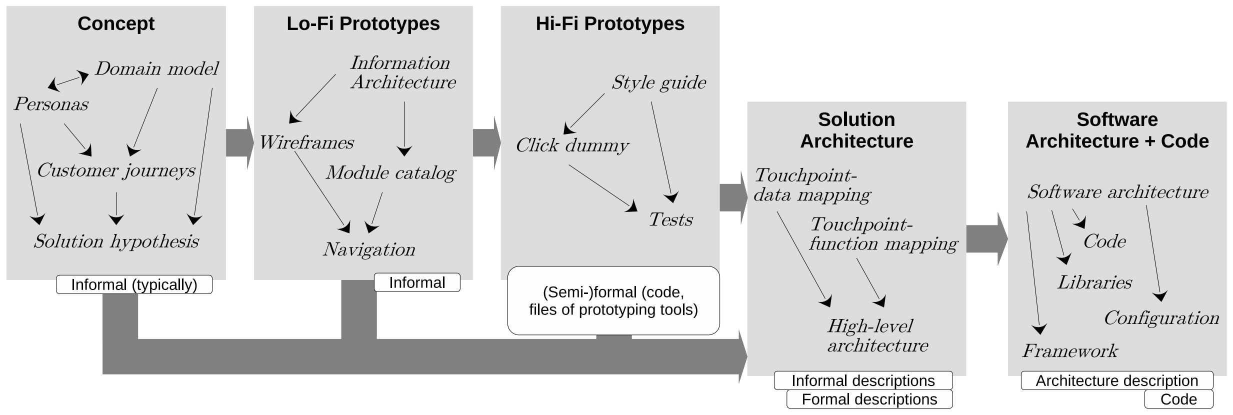


Figure 1. A typical software development process that integrates creative activities [4].

which they implement transformations, for example, by means of metaprogramming [5], code templates [6], or generative artificial intelligence [7].

B. Creative Software Development Activities

Certain kinds of software solutions, for example, one with a focus on the human-machine interface, include creative steps. Examples of creative activities are the definition of interaction patterns, of user experience in general, and user interface design in particular.

In [8], we use the term *Model-Supported Software Creation* (*MSSC*) to distinguish this kind of software development from general MDSE that relies purely on formal representations.

Figure 1 shows typical development steps and artifacts created to model aspects of a software solution. Models, such as the *Domain model*, the *High-level architecture*, and the *Software architecture* can typically be expressed in a suitably formal way as to be derived from each other by model-to-model transformations. However, other documents are typical representatives of informal documents, such as *Personas*, *Customer journeys*, and *Style guides*. There may even be dynamic artifacts, such as a *click dummy* that needs to be experienced by a human observer who interacts with it.

C. Creative Artifacts in Model-Driven Processes

Depending on the type of software, there are different steps in the development process that are of an informal nature. Some software solutions require creative development activities. Typical such activities are those from the disciplines of domain modeling, conceptual modeling, and visual design. Such development steps are typically performed manually and lead to subjective results. As a result, tools that support creative activities often produce informal representations and documents. Therefore, software projects that involve creative activities cannot be fully covered by model-driven processes in most approaches.

In order to include creative actitivities in model-driven processes, the informal documents that are generated have to be interpreted in such a way that their content can be referenced and can be extracted in a defined structure. Through such an interpretation, content may be used in software models or during model transformations.

Interpretations of documents that lack formal structure can be added explicitly. For example, their creators may provide annotations with *content references* and *metadata* to guide access to relevant content. Such annotations, however, refer to specific document instances.

Creative activities typically consist of numerous iterations. As a consequence, documents used in creative activities are subject to constant change. Changes include changes to the structure of the documents. Therefore, any fixed reference to content in such a document will potentially become invalid and metadata may become inconsistent as work progresses. As a consequence, documents are required to be constantly reinterpreted.

III. Referencing Content in Documents

In order to extract content from documents in a form that is suitable for use in a formal development process, parts of that content must be addressable. This requires documents to be structured, or to allow superimposed structures for content references.

Digital documents can be structured to varying degrees. Typically, document formats are categorized as *structured*, *semi-structured*, and *unstructured*.

A. Structured Documents

Structured documents are created according to a well-defined structure and they can be analyzed precisely according to that structure. This can be realized in three different ways. The structure of documents may be used to query for content, such as object paths based on JSON definitions. To be able to address specific parts of a document, structure elements must have stable names (paths) or stable IDs. A different approach is grammars, which can be used both to create documents of a certain form and to parse documents to identify structural elements according to linguistic constructs.

A common structure to which multiple documents conform calls for a schema or document format. Schemas of structured documents differ in the meaning they convey. A format may

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

reflect visual layout like, for example, in the case of HTML, it may use a generic semantics like, for example, XML formats for formal languages, or it may carry domain knowledge as, for example, application-specific XML formats.

B. Semistructured Documents

Documents that have a recognizable structure, but no common schema to which they conform, are called *semistructured*. Any interpretation rules applied to such documents are fragile in the sense that they may not be applicable to all document instances, or else all possible forms of documents must be considered.

If there is some technical structure that allows referencing parts of a document, then some pragmatics can be applied to interpret combinations of structure elements and content. For example, in a text document, there may be a recognizable structure of single-line terms written entirely in bold font. That may be interpreted as the term being a section heading. If the document is a software architecture description, and if the term is interpreted as a subsection of a section “Software Components”, then the term may be interpreted as the name of a software component.

In this way, semistructured documents are required to expose some recognizable structure, and interpreting them requires some known domain semantics and pragmatics to apply some interpretation rules.

C. Unstructured Documents

Unstructured documents, exhibit no structure that would allow referencing parts of a document. Typical examples of such documents are media files in binary format.

To reference parts of an unstructured document, some technical ways of addressing can be used, for example, pixel ranges in an image or timecode sequences in movies. Such references depend on the concrete document or, more precisely, on the actual presentation of it. For example, areas of an image that are defined by pixel coordinates relate to the resolution of that image. Such references are, therefore, volatile. For example, a selection of pixel coordinates is not valid for an equivalent image in different resolution.

There is no precise way to semantically reference content, although the semantics of unstructured documents can be analyzed by various algorithms.

D. Aggregating Documents from Different Sources

When accessing document collections that originate from different sources, the problem of different or varying schemas may aries. A typical approach to cope with such a situation is employing adapter components that allow accessing structured documents according to a common schema or by transforming them into a common schema [9].

E. Extracting Content from Mutable Documents

As mentioned earlier, documents created during creative activities in software engineering processes are subject to change, which means they have to be mutable (volatile, sometimes called *living* documents).

In MDSE processes, the contents of documents are used to create software models from them, or such models are in other ways related to the contents of documents. Changing documents can generally break such relationships.

One solution is to create copies of documents once they are referenced and to keep these copies stable. But this would exclude further work on those documents from the process.

Parsing is a standard approach to identifying meaningful content in a document. For formal languages, a parsing process operates on the syntactic structures of a document and applies a defined semantics to interpret those structures. Documents resulting from creative processes do not follow a fixed semantics. Therefore, classical parsing approaches based on formal languages alone do not work on them. In our current research, we augment document parsing with the application of domain knowledge.

Parsing of semistructured documents requires pragmatics since not all parts of the document have an identifiable structure. An open question is whether pragmatics can be provided by domain knowledge: two equally formatted expressions may be distinguished by some significant content. In general, domain knowledge may be necessary to decide on a parsing strategy.

Parsing is well understood for formal and, to a limited extent, semistructured representations, but it is usually applied once. Updating models based on subsequent parsing results of a modified document requires, according to our current findings, an additional relationship between document structure and domain semantics.

IV. The M³L as a Modeling Language

The *Minimalistic Metamodeling Language*, short *M³L*, is a metamodeling language. As such, it can be employed for models for different kinds of applications. We use it for first experiments in document recognition by capturing domain semantics as well as document formatting.

The M³L allows defining and deriving *concepts*. Definitions are of the general form
`A is a B { C is a D } |= E { F } |- G H .`
Such a statement matches or creates a concept *A*. All parts of such a statement except the concept name are optional.

In the course of this paper we use a graphical notation of the M³L as shown in Figure 2 for the different parts of a concept definition. For concept refinement we borrow notation from the *Unified Modeling Language* (*UML*), see Figure 2c for *is a* relationships and Figure 2d for *is the* relationships.

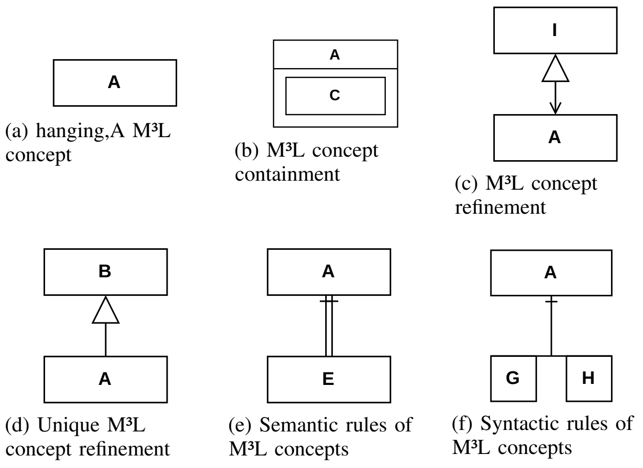
The concept *A* is a *refinement* of the concept *B*. Using the “is the” clause instead defines a concept as the only specialization of its base concept.

The concept *C* is defined in the *context* of *A*; *C* is part of the *content* of *A*. Contexts define (hierarchical) scopes. Concepts, such as *A* are defined in an unnamed top-level context.

There can be multiple statements about a concept visible in a scope. Statements about a concept are cumulated. This allows concepts to be defined differently in different contexts.

For an example of modeling with the the M³L, consider the definition of a conditional statement found in imperative

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org



(a) hanging,A M³L concept
(b) M³L concept containment
(c) M³L concept refinement
(d) Unique M³L concept refinement
(e) Semantic rules of M³L concepts
(f) Syntactic rules of M³L concepts

Figure 2. A graphical notation of M³L concepts.

```
ConditionalStatement is a Statement {
  Condition     is a Boolean
  ThenStatement is a Statement
  ElseStatement is a Statement }
```

Figure 3. Sample base model of procedural programming.

```
IfTrueStmt is a ConditionalStatement {
  True is the Condition
} |= ThenStatement
IfFalseStmt is a ConditionalStatement {
  False is the Condition
} |= ElseStatement
```

Figure 4. Sample semantics of conditional statements.

programming languages in Figure 3. It consists of *Condition* to decide whether to execute *ThenStatement* or *ElseStatement*.

Semantic rules can be defined on concepts, denoted by “|=”. A semantic rule references another concept, that is returned when a concept with a semantic rule is referenced. Like for any other reference, a non-existent concept is created on demand.

Context, specializations, and semantic rules are employed for *concept evaluation*. A concept evaluates to the result of its syntactic rule, if defined, or itself, otherwhise. Syntactic rules are inherited from explicit base concepts (given by *is a*/*is the*) and implicit base concepts (concepts with matching content).

By means of concept evaluation, semantics can be assigned to concepts. The code in Figure 4 uses syntactic rules to assign semantics to the conditional statement from the example above. A concrete statement is matched against the two subconcepts *IfTrueStmt* and *IfFalseStmt*. If one of them is recognized as a derived base concept of the given statement, the semantic rule of the matching concept is inherited. This way, the “then” statement or the “else” statement is executed (evaluated next).

Concepts can be marshaled/unmarshaled as text by *syntactic rules*, denoted by “|-”. A syntactic rule names a sequence of concepts whose representations are concatenated. A concept without a syntactic rule is represented by its name. Syntactic

```
Java is a ProgrammingLanguage {
 ConditionalStatement
   |- if ( Condition ) ThenStatement
      else ElseStatement . }
Python is a ProgrammingLanguage {
 ConditionalStatement
   |- if Condition :
      "\n  " ThenStatement
      else:
      "\n  " ElseStatement . }
```

Figure 5. Sample syntax of the conditional statement.

rules are used to represent a concept as a string as well as to create a concept from a string.

Figure 5 shows syntactic rules that map the conditional statement from the example to different programming languages.

V. First Experiments Using the M³L

Describing static documents with metadata provided as concepts that make reference to relevant parts of the content has been researched in the past. Some initial experiments with simple documents have been conducted to investigate means of linguistic document interpretation.

A. Static Document References

As a first example of document descriptions using the M³L, Figure 6 illustrates static references to (fragments of) documents. It uses an example from art history. A picture of a painting shown on the bottom of Figure 6 is described using (M³L) concepts.

The concept hierarchy starting with the concept *Document-Reference* defines references to (fragments of) documents. A *DocumentId* defines some address of a document (file name, URL, or similar), and *FragmentSelector* defines a part of a document that holds interesting content. For the example, we see a sketch of a refinement hierarchy which specifies concepts for references to two-dimensional images, for those depicting paintings, and paintings that specifically show a ruler.

A second concept hierarchy starting with *DocumentDescription* contains concepts that describe the subject of a document. The two hierarchies meet at the *PaintingDescription*. An application-specific concept RulerPaintingDescription refines it for the area of interest, and *NapoleonCrossesTheAlps* finally provides an “instance” of a ruler painting.

B. Interpretation of Semistructured Documents

As a foundation of the interpretation of some kind of documents, some general concepts are defined first. Figure 7 shows an example of documents that represent customer journeys and that are exported from a (hypothetical) whiteboard software. A concept *Board* allows to reference a whiteboard, a concept *Page* some page (assuming the whiteboard software allows to subdivide whiteboards). On a whiteboard, there is no recognizable structure below the page level. Starting with the concept *CustomerJourney*, we look for semantic structures on a

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

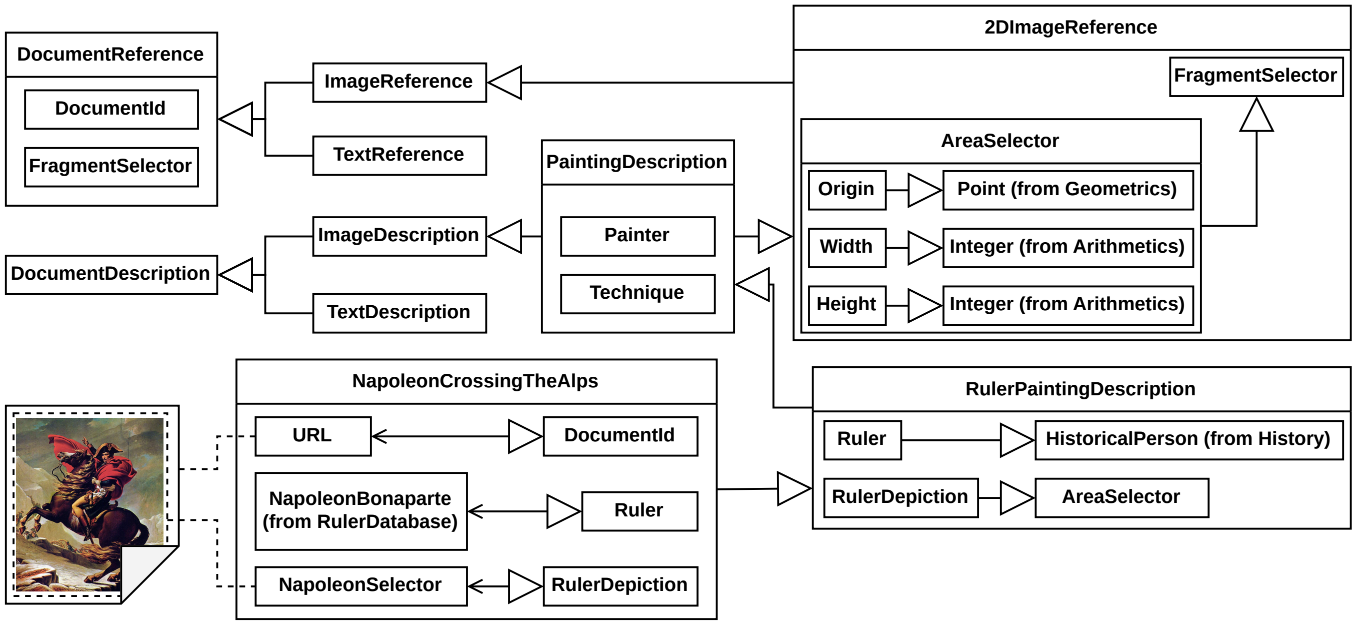
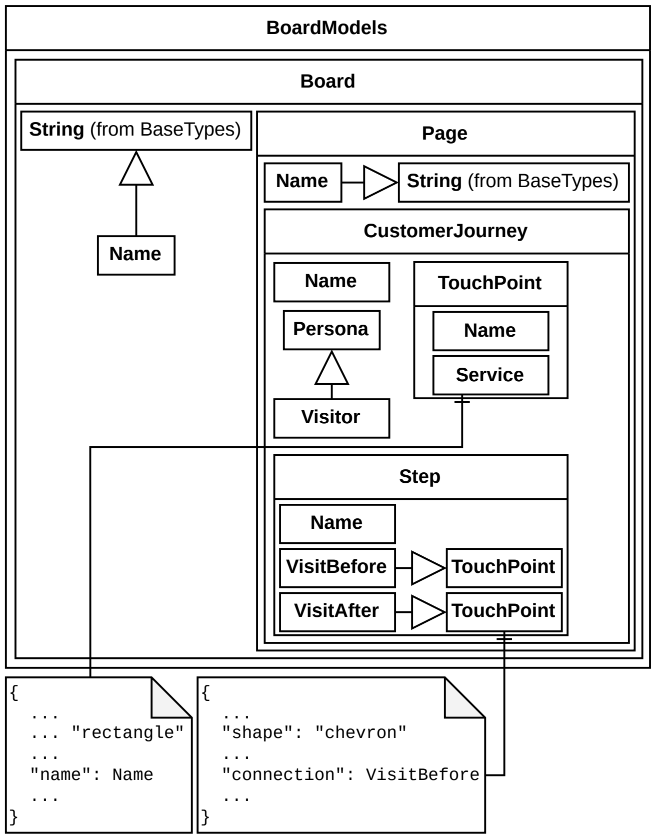


Figure 6. Static references to documents and document fragments.



```
{
  ...
  ... "rectangle"
  ...
  "name": Name
  ...
}
```

```
{
  ...
  "shape": "chevron"
  ...
  "connection": VisitBefore
  ...
}
```

Figure 7. Example of a pattern for mutable documents.

whiteboard page. A customer journey is some named (graphical) object that consists of elements that represent *Touchpoint*s and ones that represent *Step*s. A touchpoint is characterized by a *Name* and a *Service*. Syntactic rules define how such concepts are represented on a whiteboard page. Figure 7 sketches some rules that generate/recognize JSON code as it might come out of a whiteboard software that is provided as a Cloud service.

Once a customer journey has been developed on a whiteboard of that form, the syntactic rules can be used to recognize the structure and to extract the content of it. Figure 8a shows a sample customer journey. Also in the example of that figure, a (M^{3}L) concept for the board has been created as a subconcept of *Board* from Figure 7 with a reference to the board document.

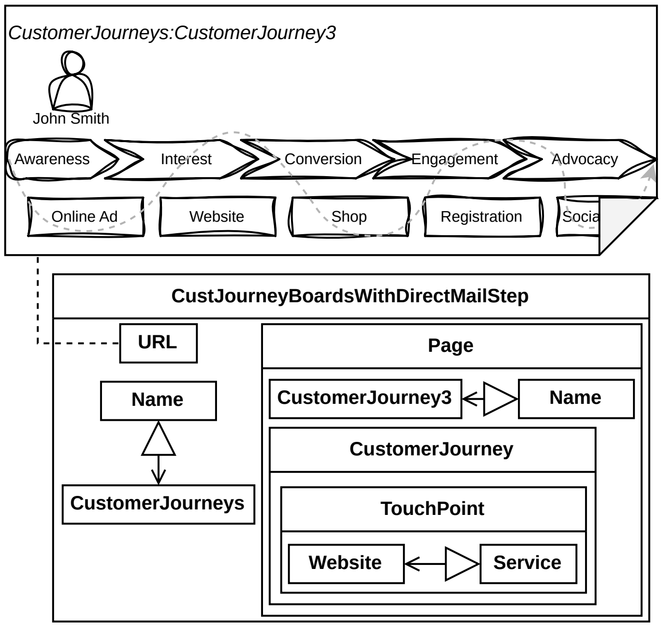
When the board is interpreted according to the syntactic rules for *Board*s, the result is the concept structure from Figure 8b. The concepts that have been created from the board reflect some of the design decisions contained in the customer journey representation, such as the participating persona and the relationships to the touchpoints it visits and the sequence of touchpoints along the customer journey.

The extracted information can be used in subsequent activities of the software development process. Using the M^{3}L, the resulting concepts can be related to concepts that represent models created in such subsequent activities.

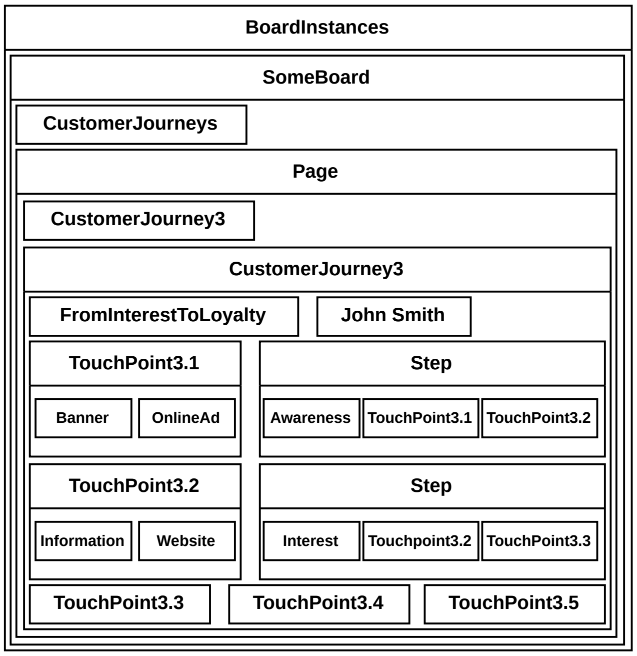
C. Reinterpretation of Mutable Documents

Mutable documents are handled by repeatedly applying the parsing process. When reinterpreting a document after a change, fresh concept definitions are made in the M^{3}L. Due to M^{3}L's way of matching definitions against existing concepts before creating new ones, previous interpretations are found and used in the parsing process. Depending on the concept model, existing concept references that were established by model-to-model transformations are preserved. In this way, documents can be modified even if they have already been interpreted and related to other models during an MDSE process.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org



(a) Example of a query to mutable documents.



(b) Example result of mutable document recognition.

Figure 8. Parsing of documents and document fragments.

Recognition of existing concepts requires some stable information. These may, for example, be unique names as well as a certain location in the document structure where it is placed. In the example of the digital whiteboards above, names might be given in a specially positioned text field. As a consequence, the documents are not completely mutable, at least not in terms of content.

An agreement on some recognizable information constitutes a restriction to the idea of mutable documents. Finding ways of leveraging this situation is subject to future work.

VI. Conclusion and Future Work

In this paper, we investigate an approach to integrate semi-structured documents supporting creative activities into MDSE processes. Using the M³L, documents can be parsed based on their syntactic structure in conjunction with the semantics of the concepts represented in such documents. A first simple experiment shows that content can be extracted from a document in a suitably formal form if the document follows some conventions. The concepts recognized in a document can serve as model elements that link the documents to the chain of model-to-model transformations of MDSE processes.

Future work will need to test this approach with a range of existing file formats and service APIs to further investigate the limits of document interpretation and possibly identify additional requirements for parsing technology. There are limits to the extent to which documents can be modified without losing existing links to software models. These limits are not well researched. We need to find the limits, ways to extend them, and notations to describe parts of documents that must not be altered. Another future research direction concerns a form of roundtrip engineering in which documents are not only interpreted, but also generated from models that need to be presented in a form suitable for non-technical stakeholders.

Acknowledgement

The author thanks the Nordakademie for granting the opportunity to publish this work.

References

[1] G. Liebel *et al.*, “Human factors in model-driven engineering: Future research goals and initiatives for mde”, *Software and Systems Modeling*, vol. 23, no. 4, pp. 801–819, 2024.
[2] E. Herac, L. Marchezan, W. Assunção, R. Haas, and A. Egyed, “A flexible operation-based infrastructure for collaborative model-driven engineering”, in *Modellierung 2024*, ser. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik e.V., 2024.
[3] I. Galvao and A. Goknil, “Survey of traceability approaches in model-driven engineering”, in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, 2007, pp. 313–313.
[4] H.-W. Sehring, “Visual artifacts in software engineering processes”, in *Proceedings of the Sixteenth International Conference on Creative Content Technologies*, ThinkMind, 2024, pp. 1–6.
[5] S. Trujillo, M. Azanza, and O. Diaz, “Generative metaprogramming”, in *Proceedings of the 6th international conference on Generative programming and component engineering GPCE ’07*, Association for Computing Machinery, 2007, pp. 105–114.
[6] J. Arnoldus, M. Van den Brand, A. Serebrenik, and J. J. Brunekreef, *Code generation with templates*. Springer Science & Business Media, 2012, vol. 1.
[7] K. Lano and Q. Xue, “Code generation by example using symbolic machine learning”, *SN Computer Science*, vol. 4, Jan. 2023.
[8] H.-W. Sehring, “Model-supported software creation: Towards holistic model-driven software engineering”, in *Proceedings of the 2023 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications*, ThinkMind, 2023, pp. 113–118.
[9] I. Amous, A. Jedidi, and F. Sèdes, “A contribution to multimedia document modeling and querying”, *Multimedia Tools and Applications*, vol. 25, pp. 391–404, 3 Oct. 2005.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org