Towards Integrating Usability and Software Engineering Using the Mapache Approach

Alexander Behring, Andreas Petter, Max Mühlhäuser

{behring, a_petter, max}@tk.informatik.tu-darmstadt.de

Abstract: The recent success of various Apple products affirms that usability engineering can be a beneficial asset in software product development. But considerable problems exist in integrating usability engineering and software engineering. In this paper, we argue that the technical basis – the disciplines' artifacts and used tools – are a potential hurdle for integration. Model-driven development (*MDD*) is put forward to address these problems. The Mapache approach persued in our research specializes MDD for User Interface (UI) engineering. Four of its concepts are discussed in the light of integrating usability and software engineering.

1 Introduction

When integrating usability (*UE*) and software engineering (*SE*), most companies face a gap. Attempts were made to bridge it in several dimensions, as Göransson et al report in [GGB03]. Two IFIP[1] working groups investigated bridging the gap in more detail (cf. http://www.se-hci.org).

An integration of both aspects must, besides on a process level, be addressed on the technical level of artifacts and tools. Hereby, consistency checks crossing UE and SE become possible, modifications of artifacts can be propagated across the UE and SE border by (semi) automatic tools, changes can be tracked, and the other discipline's resources can be used (e.g., mock-ups in SE artifacts).

This paper discusses four concepts of *Mapache*, a model driven UI engineering framework developed in our group, in the light of the integration of UI and SE. We argue that using a model driven approach, with all artifacts using a common formalization (metamodel), integration of SE and UE artifacts on a technical level becomes more feasible.

2 Matching Usability and Software Engineering

Mapache is a model-driven UI engineering framework developed in our group and used as a research platform [BPM09]. It is based on our previous experience [BHWD07]. In Mapache, all components are centered around models (cf. figure 1). Tools operate on them,

[1] http://www.ifip.or.at/

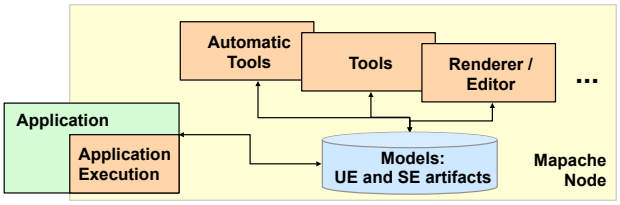

Abbildung 1: A sketch of the most important elements of the Mapache environment. All components are hosted inside a node and access the central model repository.

application logic is bound to UI model elements for capturing events. Renderers present the UI model in rendered form to the user at runtime and can also be used for editing (at run- and design time).

We propose to use models as an integration point of SE and UE. In the following, we will discuss four Mapache concepts in the light of this integration:

- **Use of models**: artifacts in Mapache are models complying to metamodels (schemata), supporting links between different artifacts and their elements.
- **Execution of models**: instead of transforming models into code or another intermediary artifact to be executed, they are directly interpreted at runtime, allowing the use of design time techniques at runtime.
- **Refinement of UIs**: tracking of interdependencies between different UI versions allows more flexible synchronization.
- **Integration of automatic and manual processing**: automatic approaches are combined with manual rework, supporting a more efficient synchronization of linked artifacts.

2.1 Using MDD to Link Usability and Software Engineering

Model-driven engineering's primary goal is to raise the level of abstraction [HT06], thereby aiding to the understanding of complex problems and their potential solutions [Sel03]. Göransson et al argue in [GGB03] that using an MDD approach even broadens the gap between UE and SE, because the users are taken out of the loop. We argue differently: users are taken into the loop by better integrating the different artifacts of the system. Instead of disconnected UE or SE artifacts, their relations are made transparent. When discussing with the user, UE artifacts can be used for easier communication: software engineers can trace the area of the discussion "in their view" through linked SE artifacts[2]. In our eyes, this argument also does not contradict the request by Göransson et al in [GGB03] to deepen the understanding by the development teams of the problem domain.

[2] also elements inside these artifacts can be linked

Mappings are a fundamental concept in MDD [SCF+06b] and link model elements on a technical level (the process level is out of the scope of this paper). They can be realized using explicit link-elements (associations) or described by transformations (e.g., using QVT[3]) in an imperative or declarative way.

Expressing UE and SE artifacts as models allows to leverage links between model elements for their integration. An artifact used in an MDD approach hereby must comply to a metamodel (schema), which precisely defines the syntax of the artifact. Ideally, the metamodels of the different UE and SE artifacts in turn have a common metamodel (i.e. the meta-metamodel of the artifacts). This allows to operate across different artifact types using a common tooling. In Mapache, the Eclipse Modeling Framework[4] (*EMF*) is used as this meta-metamodel.

When building tools to support engineers, beyond the syntax, the semantics of the artifacts must be well-defined. If even the pragmatics of artifacts can be made accessible, more complex relations can be exploited in these tools. These three levels, syntax, semantic and pragmatics, are discussed in the next section.

2.1.1 Semiotic Match

As described in the previous section, UE and SE artifacts (UML diagrams, UI sketches, requirements, etc.) in an MDD approach are formulated using a common meta-metamodel. To link them, a match has to be made on multiple levels: ideally, syntax, semantic and pragmatics of the artifacts are accessible and can be matched. By accessible, we refer to whether syntax, semantic and pragmatics can be used and exploited by automatic tools and not only by humans. For example, whether a tool to produce an initial UI suggestion can take into account the pragmatic knowledge encoded in the source artifacts.

Syntactic Level: the syntactic level is very well-formalized in the SE community. UML, SysML and other models[5] have a well-defined syntax that can be fed into compilers and interpreters. Models in MDD approaches have a formal syntax, defined through their meta- and meta-metamodels.

Usability Engineering artifacts mostly have a well-defined syntax, too – i.e. when using a standardized language. UML, for example, provides a (model-based) notation for Use Cases. UE artifacts that do not comply to a metamodel (schema) or/and are not available electronically (e.g., a sketch on paper) have no accessible syntax. Consequently, UE artifacts that should be linked must be available electronically and comply to a metamodel.

Semantic Level: when looking at the semantic level, formalization is more challenging. SE has a good stand with artifacts such as Java code or UML diagrams[6]. For example, Java programs can be executed. There are also initiatives to directly execute UML models.

Artifacts of UE tend to have less well-formalized semantics. Classes of artifacts like requirement documents are more challenging. They often are formulated in prosa and thus have

[3] QVT – Query View Transformation, an OMG standard, cf. http://www.omg.org/spec/QVT
[4] http://www.eclipse.org/emf
[5] For textual languages, this also holds if they comply to a defined syntax, e.g., given in Bachus-Naur-Form.
[6] UML does have ambiguities, but this is irrelevant for the sake of this argument

no formally specified semantic that can be accessed. Consequently, these artifacts can be read-in by machines (due to their well-defined syntax), but reasoning about their contents is not possible (lacking semantics). Sketches can also be read-in by machines, but their semantic is not accessible if they are represented as a bitmap.

Pragmatic Level: well formalized pragmatics are hard to specify. Contextual information has to be taken into account to identify the pragmatics of artifact-elements. For code-optimization, this has successfully been accomplished. The compiler identifies not only what a given single statement produces, but also in which context it is used (e.g., whether the statement's result is used at all, and if not, it can be removed for compilation).

As the semantics in UE artifacts are often not accessible, identifying pragmatics in UE is even harder. For example solving "what does using interaction device X in usage context Y mean?" can hardly be answered by a machine (and might further be subject to one's subjective interpretation).

In Summary, SE artifacts are more easily accessible than UE artifacts. For SE artifacts, sometimes even the pragmatics can be accessed. In contrast, for UE artifacts, this is less likely, because accessing the semantics already is challenging. For a successful and deep integration, it would be beneficial to have better accessible UE artifacts.

2.2 Executing Models

Da Silva identified two basic approaches [Pin00] to present the rendered UI model to the user. The *UI Generator* approach produces an intermediary artifact that is then rendered to the user. For example, a UI model that is first transformed into HTML, and then rendered for display. Da Silva notes that the generator approach only produces static descriptions of User-Interfaces [Pin00].

In contrast, Mapache takes the *UI Runtime System* approach [BPM09] and interprets the UI models directly at runtime. No intermediary artifact is used. Such an approach often is used to allow easy adaptation to varying contexts of use [Pin00, BLFA08, SCF06a]. Changes to the model are instantly reflected in the rendered UI, reducing roundtrip times when reworking the user interface, bringing design and runtime closer together. Links between model elements can be traced at runtime, allowing direct access to all relevant parts of the application when discussing an issue. Executing models thus allows to use design time techniques at runtime.

2.3 Refinement of User Interfaces

When designing different UI versions for one applications but different context of use, new challenges arise. In [BPFM08], we discussed observations in this respect. Mapache addresses these by providing a mechanism to describe interrelations between the different UI versions (conceptually a special form of inheritance). The relations are exposed to track

and make modifications across multiple UIs [BPM09].

Given the interrelations between the different UIs, synchronization between UE and SE artifacts is more flexible. Changes to the UI by usability and software engineers do not have to be synchronized manually for every UI version. New behavior is automatically added in new UI versions produced in parallel through an inheritance feature of the Mapache approach. Thus, usability tests of different (e.g., new) UI versions can be conducted, independent of the implementation of the application behavior. Furthermore, when working and discussing one UI version, UI refinement allows to automatically identify related elements in other UI versions.

2.4 Integration of Manual and Automatic Processing

There exists a tradeoff between automatization and quality in UI creation. Automatic approaches ease the work of engineers, but have proven to be inadequate to produce UI with good usability. On the other hand, when creating all artifacts manually, some target platforms and contexts of use will not be important enough to be covered. Mapache integrates automatic and manual processing following the 80-20 rule: use automatic approaches to increase the range and only invest manual work in the artifacts that are of importance. Mapache provides such integrated mechanisms: UIs can be generated automatically [PBM09]. The generated UIs then can (if needed) be adapted by developers. Furthermore, semi-automatic propagation of modifications through multiple UI versions eases applying modifications in Mapache [BPM09].

This concept can be extended for bridging the UE-SE-gap. For example, changes to requirement documents are automatically propagated to UI models and SE models. These are either applied automatically or can be hinted to developers for manual processing. The work of usability and software engineers can be synchronized more efficiently using this concept.

3 Summary and Conclusion

In this paper, we discussed the integration of usability and software engineering on a technical level. We put forward the use of concepts from Mapache, a model-based UI engineering approach. Hereby we emphasized the use of four concepts to more closely integrate usabiliy and software engineering: *i)* models can provide links between artifacts, *ii)* direct execution of modeled artifacts facilitates the use of design time techniques at runtime, *iii)* UI refinements tracks interdependecies between UI versions for more flexible synchronization, and *iv)* the integration of automatic and manual processing yielding the choice how much effort to put into which artifact and more efficient synchronization.

But such an integration on the technical level alone will, in our eyes, not be the solution. Various other issues have to be addressed, like the integration on the process level and education [GGB03, FC00]. In our opinion, further research should be conducted to

what degree usability engineering artifacts should be formalized. Furthermore, tooling for usability, software engineers and their integration should be addressed.

Literatur

[BHWD07] Alexander Behring, Matthias Heinrich, Matthias Winkler und Waltenegus Dargie. Werkzeugunterstützte Modellierung multimodaler, adaptiver Benutzerschnittstellen. *i-com - Zeitschrift für interaktive und kooperative Medien*, 6(3):31–36, Dec 2007.

[BLFA08] Marco Blumendorf, Grzegorz Lehmann, Sebastian Feuerstack und Sahin Albayrak. Executable Models for Human-Computer Interaction. Seiten 238–251, 2008.

[BPFM08] Alexander Behring, Andreas Petter, Felix Flentge und Max Mühlhäuser. Towards Multi-Level Dialogue Refinement for User Interfaces. In *CHI Workshop on User Interface Description Languages*, Apr 2008.

[BPM09] Alexander Behring, Andreas Petter und Max Mühlhäuser. Rapidly Modifying Multiple User Interfaces of one Application. In *ICSOFT (SE)*. INSTICC Press, 2009. to appear.

[FC00] Xristine Faulkner und Fintan Culwin. Enter the usability engineer: integrating HCI and software engineering. *SIGCSE Bull.*, 32(3):61–64, 2000.

[GGB03] Bengt Göransson, Jan Gulliksen und Inger Boivie. The usability design process - integrating user-centered systems design in the software development process. *Software Process: Improvement and Practice*, 8(2):111–131, 2003.

[HT06] B. Hailpern und P. Tarr. Model-driven development: the good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461, 2006.

[PBM09] Andreas Petter, Alexander Behring und Max Mühlhäuser. Constraint Solving in Model Transformations. In Richard F. Paige, Hrsg., *International Conference on Model Transformation, ICMT 2009*. Springer, 2009. to appear.

[Pin00] Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. *Lecture Notes in Computer Science*, 1946:207–226, 2000.

[SCF06a] Jean-Sébastien Sottet, Gaëlle Calvary und Jean-Marie Favre. Models at Runtime for Sustaining User Interface Plasticity. In *Workshop Models@run.time in conjunction with MoDELS 2006*, Genova, Italy, October 2006.

[SCF+06b] Jean-Sébastien Sottet, Gaëlle Calvary, Jean-Marie Favre, Joëlle Coutaz und Alexandre Demeure. Towards Mapping and Model Transformation for Consistency of Plastic User Interfaces. In Kai Richter, Jeffrey Nichols, Krzysztof Gajos und Ahmed Seffah, Hrsg., *Workshop on The Many Faces of Consistency in Cross-platform Design, ACM conf. on Computer Human Interaction, CHI 2006*. ACM Press, 2006.

[Sel03] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, Sept.-Oct. 2003.