Erschienen in: Proceedings of the IADIS International Conference on Interfaces and Human Computer Interaction 2007 / Palma dos Reis, António et al. (Hrsg.). - Lisbon : IADIS Press, 2007. - S. 3-10. - ISBN 9789728924393

MODEL-BASED VISUAL SOFTWARE SPECIFICATION

Thomas Memmel, Mathias Heilig, Harald Reiterer
Human-Computer Interaction Lab, University of Konstanz
Universitätsstrasse 10, Box D73

ABSTRACT

Collaborative corporate requirements engineering and system specification demands modelling approaches that all stakeholders can understand and apply. Using traditional methods, an IT organization often experiences frustrating communication and competency issues. Current processes lack adequate tool support for networked and complex projects. We present a tool-chain for model-based requirements engineering and a visual specification of interactive systems. Through a substitution of text-based artefacts, stakeholders are able to model systems with high user interface usability and functional quality in less time.

KEYWORDS

Requirements Engineering, Visual Specification, Corporate Software Development, Design Method

1. INTRODUCTION

From the authors' experience with automotive software engineering (SE), we see that the development of corporate software systems faces very demanding challenges. Several critical ingredients contribute to development failure: the increasing importance of the user interface (UI), the separation of professions, particularly SE and usability engineering (UE), and consequently a lack of methods, tools and process models that integrate all stakeholders and their discipline-specific demands. The Standish Group identifies missing or incomplete requirements as one of the main reasons for project failure (Standish Group 2003). Communication among stakeholders is unsuccessful due to inappropriate terminologies and modelling languages (Potts et al. 1994, Zave and Jackson 1997). Text is ambiguous and leaves room for interpretation. Formal models and the respective tools are generally too complex to be understood by non IT-stakeholders. After all, stakeholders should be able to express and model their requirements in a familiar way.

In this article we introduce a model-based prototyping approach to the development of interactive corporate software systems. By employing prototypes as vehicles for both design and system specification, we are able to bridge existing gaps while making the overall design process more efficient and effective, resulting in lower development costs, but improved software quality. Simultaneously we address the different requirements of stakeholders by offering different levels of abstraction and formality. Furthermore, by following a model-based approach, time-consuming assignments such as writing documentation or usage guidelines can also be automated at the push of a button. In Section 2 we describe the various influences on corporate software development using the example of the automotive industry. We will explain how those stakeholders who are directly involved can be supported through a model-based approach. Consequently, we present the concept of our tool-chain and give an overview on some implementation details in Section 3. We will show how our approach can be applied in practice by going through an application scenario. In Section 4 we summarize our findings and indicate the direction of our future research.

2. SHORTCOMINGS OF TEXT-BASED SOFTWARE SPECIFICATION

In the automotive industry, a wide range of different corporate software systems exists. Intranet and internet sites play an important role as well. When employees are unable to perform their tasks, the usage of intranet

applications may be entirely incorrect, slow to make progress, and may finally lead to reduced acceptance and exploding support costs. Websites are important for the market success of an automotive brand and the acceptance of its products by customers. A website must create brand awareness, transport product values, enable product search, offer contact to nearby retailers and is expected to increase customer loyalty.

However, in this article we explain the challenges of corporate software engineering using the example of in-car information systems. Such systems are intended to support the driver during travelling and must be intuitive and easy to use. Being an important part of modern cars and increasingly playing a significant role in the buying decision, driver information systems need to be aligned with corporate design[1] (CD) and corporate identity[2] (CI). Hence, developers need to think in several disciplines rather than concentrating just on pure functionality. But expanding development practice is difficult when facing pressure of time and melting budgets. Especially when the UI has great weighting, methods of agile development are not sufficient to build usable systems (Constantine 2002). Furthermore, highly networked and dispersed development teams are mostly unable to employ agile methods (AM) appropriately. What is needed is adequate tool-support that supports collaborative and interdisciplinary development processes and which satisfies agile principles and practice[3].

In the automotive industry we noticed five main groups of engineering stakeholders[4] that need to be linked by capable tools. Product managers define strategic goals and benchmarks and gather requirements together with human-computer interaction (HCI) specialists. The latter usually develop a navigation concept, create appropriate dialogue sequences, and are responsible for early- and late-stage usability evaluations. Designers need to guarantee alignment with CD and CI and write a style guide together with other stakeholders. Technical experts define test settings for assessing the software product in simulation frameworks on desktop computers as well as on the embedded system in a prototype of the car. IT suppliers are the fifth group of stakeholders and are usually assigned to build the specified software system. In a survey among the stakeholders in the electrical engineering department of an automotive company, we found that managers, those with responsibility for specific functions, domain experts and HCI specialists exclusively apply standard office applications for software specification. Out of 12 stakeholders we interviewed, all use Microsoft (MS) PowerPoint, 10 employ MS Excel, 7 use MSWord, 6 paint their requirements with simple drawing tools and some more tool-experienced ones use MindManager (2) and MS Visio (1). None apply more sophisticated CASE-tools. As well as their familiarity with such applications, the most important reasons for avoiding other tools are their bad usability, the great effort needed in learning to use them, and the difficulty of understanding the related terminologies. Consequently, there is a great need for a standardized specification process. The text-based documents that are produced easily reach a complexity of several hundred pages and are therefore hard to maintain. Due to the various formats used, there is a critical danger of loss of precision and misinterpretation. But most importantly, those responsible for actually implementing the system will use completely different tools. They need to translate text-based documents into SE models and code. Hence, double work is required i.e. reworking with code what has already been textually described. At this stage, the transition from abstract system descriptions to a running system again leaves room for interpretation and, moreover, is a black-box process due to the absence of sophisticated modelling languages. The corresponding media disruption makes it difficult to cross-check the implementation with the underlying requirements at the quality gateway. Summing up, the worst thing that any company can do is attempt to write a natural language specification for a system, especially when the UI has great weighting (Horrocks 1999). A picture is worth more than thousand words when designing interactive systems. If the client is unable to assess both the look and feel of a system before forwarding the specification to a supplier, late-cycle changes will be the expensive necessity. Furthermore, early prototypes are needed for evaluation with end-users.

Prototypes are already known by UE and SE and are an accepted means of communication during the early stages of design. We wish to elevate their importance for corporate software development and propose a tool-chain for a model-based generation of prototypes at any stage of design. Hence, prototypes of different

[1] Corporate design is a part of the corporate identity and describes the overall visual appearance of an organization

[2] Corporate identity implies the overall characteristics of an organization. It helps to recognize an organization as consistent personality that acts, communicates and appears uniformly.

[3] http://www.agilemodeling.com

[4] We use the term “stakeholder” instead of “actor” to point out to the pretension to actively involve experts on client and supplier side.

fidelity should be created based on collaboratively defined models at the push of a button. Consequently, prototypes can be employed as vehicles propelling the engineering process due to the possibility of permanent testability and the chance to experience an interactive simulation whenever the underlying models change. Usually, stakeholders need to write code for building detailed and running prototypes. With a model-based approach, we reduce the complexity of coding to the extent of domain-specific languages (DSL) that perfectly fit the application and problem at hand. Furthermore, we want to offer mechanisms that allow a visual application of the DSL. Instead of writing text, stakeholders should be able to express themselves visually. By segmenting the modelling process into different parts, we can achieve a separation of concerns (according to stakeholder expertise) and problem-specific levels of abstraction. Depending on what has to be modelled, unessential information should be removed from the modelling surface.

Ultimately, we are able to avoid late and costly changes. Because a programmer can always pop-up the running specification to verify requirements, the chance for engineering success increases. Although some initial effort is necessary to define the DSL, we can reduce the overall effort on both client and supplier side. As model-based approaches allow the generation of code based on the underlying formal descriptions, suppliers can reuse parts of the system that were created during the specification process.

3. A TOOL-CHAIN FOR VISUAL SYSTEM SPECIFICATION

In order to ensure a standardized operation sequence, the individual applications of a tool-chain must offer a high degree of compatibility. Furthermore, it is advisable to keep the components of the tool-chain to a small number. In addition, the ease of use can be significantly improved through well-known control concepts and terminologies, such as those of standard Office applications (see Section 2). For the different stakeholders, the possibility to create a specification with visual methods is a substantial advantage. This means that they should be able to graphically define models that are compliant with their domain-expertise. A certain level of formality of models is necessary in order to be able to produce simulations and specifications automatically. Consequently, this formality should be hidden as far as possible to reduce engineering complexity (AM: Assume Simplicity).

With the simulation, the usability of a system, as well as logical errors and inconsistencies, can be detected and evaluated during early stages of design (AM: Rapid Feedback). Due to the usage of models, the impact of changes to the formal descriptions can be immediately assessed at the push of a button (AM: Embrace Change). It is therefore possible to provide formal specifications to the development partners (i.e. suppliers) at any time. The exchange can take place by either providing the generated code or forwarding the system descriptions in a mark-up language (e.g. XML). The interactive simulations must be executable on several platforms in order to be able to receive sound test results e.g. from their assessment in driving simulators or on the car's embedded system. When experiencing the software under realistic conditions, aspects such as beam of light, driver diversion or the haptics of the real control elements can be examined. Furthermore, the expandability of the tool-chain plays an important role. In the case of new functions or the bonding of other in- and output-devices (e.g. language control), it must be possible to integrate and/or model these accordingly.

In the following, the tool-chain is described in detail using the example of the development of a driver information system. We separate the development process into the three dimensions of design, content and behaviour (see Figure 1), resulting in three components of the tool-chain (Bock 2007). Suitable applications will be integrated, and their contribution reunited, with the help of a code generator and a software framework (AM: Multiple Models, Create Multiple Models in Parallel). The dimension content is responsible for the processing and actual representation of the information objects. The content that needs to be integrated into the screens of the system is usually defined by technical experts. With the dimension design we cover the creation and editing of UI screens. Accordingly, UI designers are mainly responsible for styling the system's look and feel. Operation sequences and menu structures are assigned to the dimension behaviour. The corresponding logics are developed by technical experts (i.e. programmers). Ergonomists (i.e. usability specialists) are involved in all areas of development. Figure 1 shows the conceptual composition of the tool-chain and the corresponding assignment of stakeholders. Programmers are also responsible for providing UI components (e.g. widgets, controls) that enable the designer to create the interaction layer.

The domain-specific language (DSL) is defined up-front by all stakeholders jointly, as interdisciplinary knowledge of the application domain is necessary (AM: Model with others). The modelling language is based on a meta-modelling language. We therefore use a meta-case tool[5], which helps to provide models of the DSL (see Figure 1). When using an existing modelling language (e.g. UML) the syntax and semantics must be modified (e.g. UML profiles) in order to be able to illustrate the domain. The advantage of a DSL is that it is developed for a particular domain. As stakeholders are sufficiently familiar with the domain of the DSL, they do not have to learn new modelling or programming languages. During disjoint modelling of the separated concerns, the predefined DSL guarantees later compatibility of combinability of models. Nevertheless, collaborative sessions help to discuss and extend the models as well as the DSL. Hence, ongoing synchronization is not limited to code generation

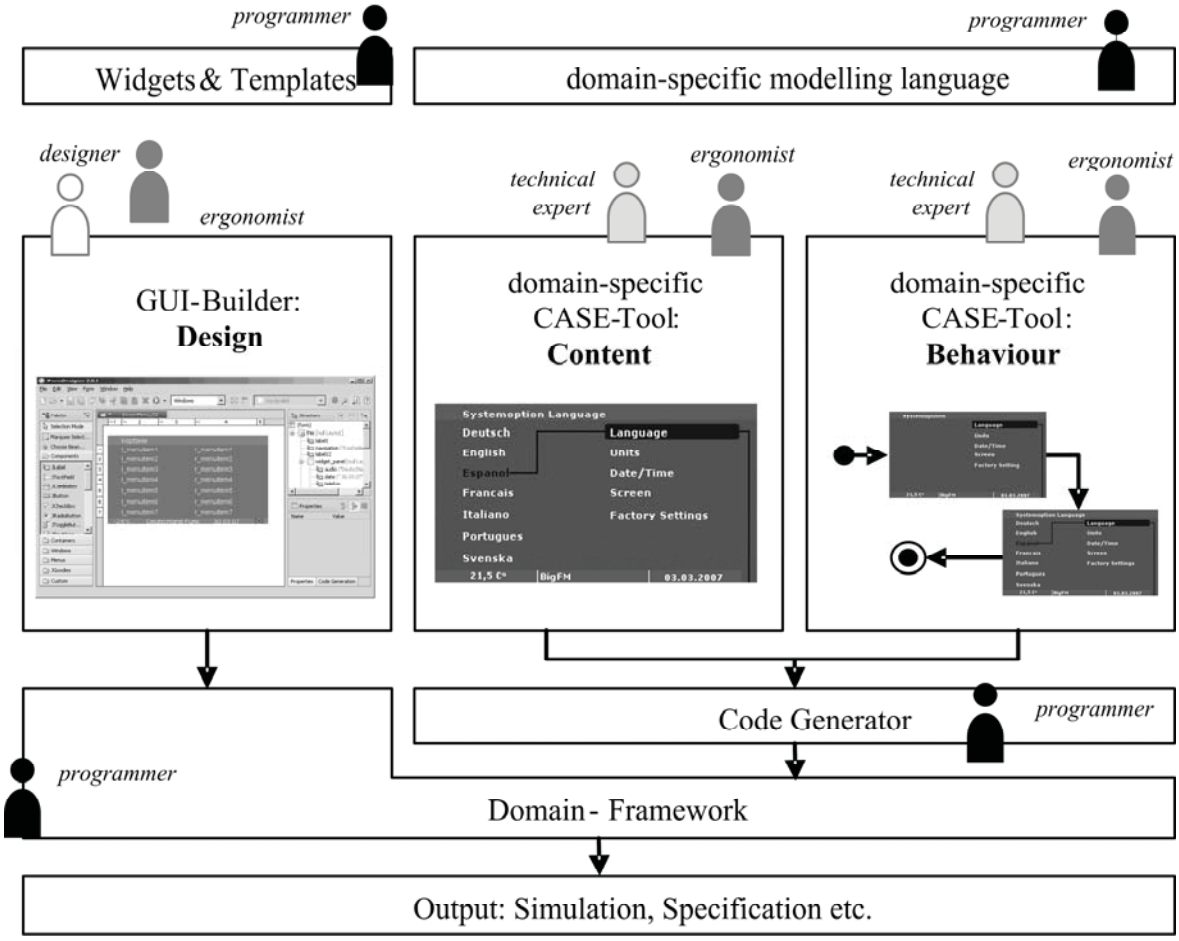

Figure 1. Structure of the tool-chain (Memmel et al. 2007

For the creation of the DSL, the concepts of the domain are analyzed and tangible objects are identified and classified. In the following example, these are a set of screen classes (e.g. information screens, menu screens and/or submenu screens) as well as some input elements. For these objects we define unambiguous terms, provide meaningful symbols and describe their relations. The transitions are described by the linkage of static screens and possible interaction into other system states. Certain dialogue sequences can be permitted or excluded by rules defined in the underlying formal model. These constraints prevent situations in which inconsistent control sequences are modelled. An example of a modelling rule is the restriction that a

[5] MetaEdit+, MetaCase (http://www.metacase.com)

special input element can be used only once per screen. This also helps new users to design a system without being afraid of creating inconsistencies and conflicts by accident.

For the dimension design we use the GUI-Builder called JFormDesigner, whereas for the dimensions content and behaviour we apply the methods of model-based software development (Speliers 2004). The reason is that, we didn't find an adequate meta-modelling language that offers enough possibilities to describe user interfaces with a sufficient modelling language. GUI-Builders still provide the designers with more potential to implement creative ideas than does a formal modelling language.

The GUI-Builder offers the designer the possibility to create and edit UIs without programming knowledge. The JFormDesigner (see Figure 2) itself offers an easy and intuitive UI. The developers can create dialogue components for an UI by simple drag and drop operations and interaction by direct manipulation. Dialogue components can be created and placed on the screen as easily as with a simple painting program. Individually created surface components (Widgets) can be integrated and the tool can deal with colours, transparency and fonts for designing attractive UIs. Unused functions and symbols can be faded out. In our example, the organization of the screens and templates, as well as the graphical creation of individual components (e.g. volume controls), is done with the GUI-Builder. JFormDesigner produces both Java code and XML for the description of the UI. Hence, the layouts of the screens can be easily integrated into the simulation. Other XML-based GUI Builders can also be integrated into the tool-chain.

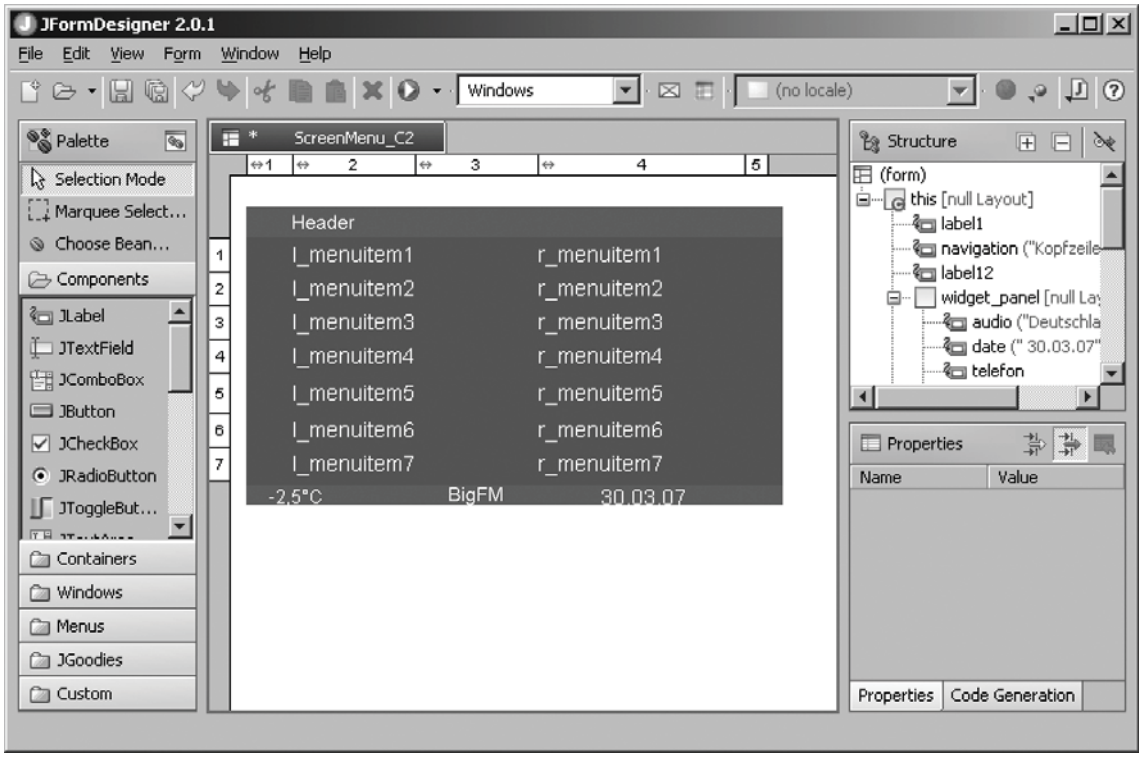

Figure 2. Design component in the GUI-Builder

Figure 3 shows models created with the DSL. In contrast to model notations such as State Chart diagrams, we use visual objects from the domain. Hence, these objects support communication among stakeholders during system development. In the modelling editor, objects are placed by drag and drop, as with the GUI-Builder. This is followed by the editing of the objects' parameters, such as e.g. the menu entries of a screen. The objects can now be connected by means of different control elements (typically representing hardware buttons). Consequently, a navigation path of the system is built. With the help of the DSL, unreasonable connections can be prevented by the domain rules of the modelling language.

The user can reduce the complexity of the models by dividing them into several smaller units (see Figure 3, 1-2). In addition, different abstraction levels can be implemented with the modelling tool. The user can decide whether he would like to see or edit further details of the logic and behaviour concerning a single

screen (see Figure 3, 2-3). Thus, the view is adapted to suit the problem thanks to the respective focus. Hence, with the modelling language that has been created, the application domain can be visualized and accessed in the best possible way.

The transformation of the models into the different output formats (simulation, formal specification, documentation, etc.) is done with a code generator that was written in-house. The generator has two responsibilities: (i) the interpretation of the formal language and (ii) concretizing the models on the basis of the formal language by (iia) a transformation with associated rules or (iib) a macro expansion with templates (Sendall & Kozaczynski 2003). The adoption of generative methods increases the code quality compared to manual programming and reduces the error rate. In our example, the generator of the meta-case tool is used for the transformation of the models

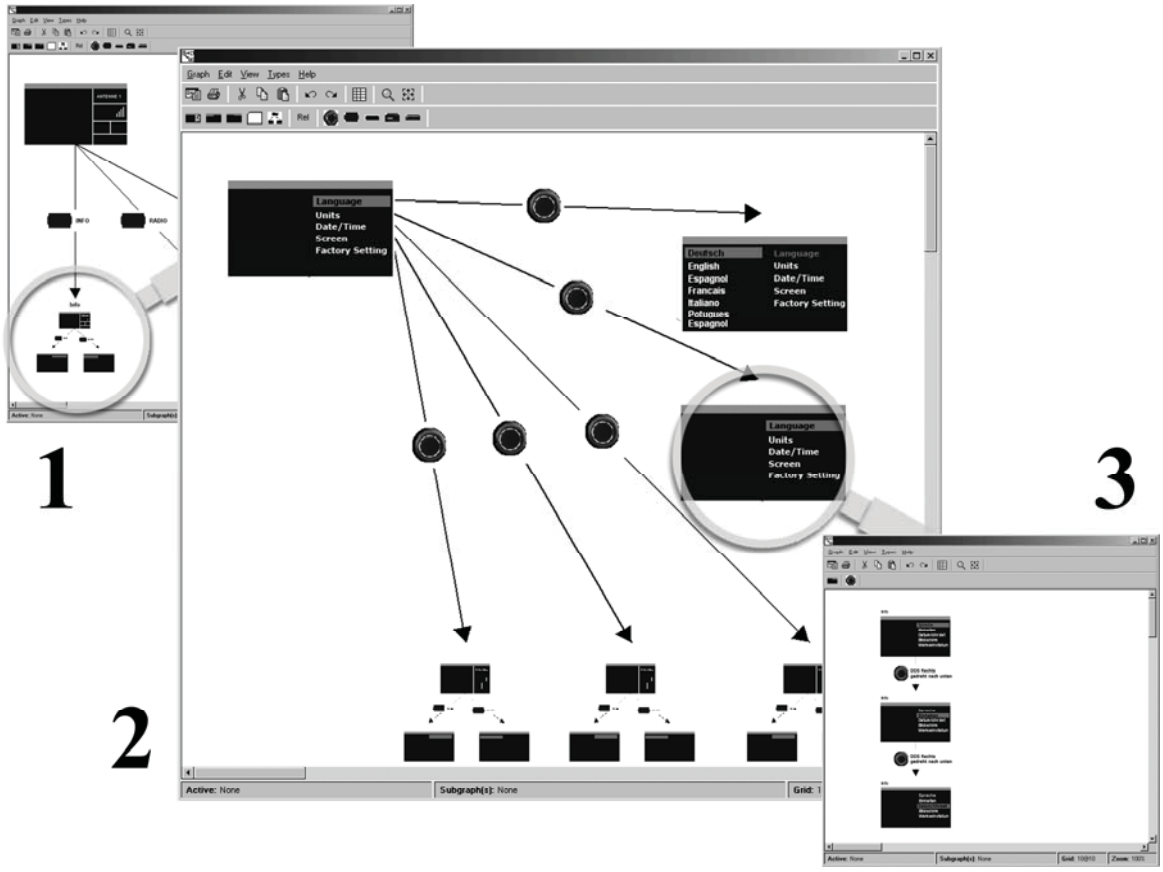

Figure 3. Models of the domain-specific language in different abstraction levels

The transformation rules and/or templates must be provided once for each respective output format. The templates are formulated with the proprietary script language of the meta-case tool. By means of this script language, the information about the models can be accessed, such as e.g. the contents of the screens. This information will be used in order to produce source code (classes, configuration etc.) for the simulation. Thus e.g. one class for every screen is generated. The class also contains the navigation path from one screen to the next. It is advisable to generate only the essential components of the simulation, because the programming of the templates for the code generator is obviously less easy than programming with an object-oriented programming language. Functions that do not change from model to model are therefore provided in the domain framework (see Figure 4). Our domain framework, which is implemented in Java, needs several components for an interactive simulation. For each screen type of the domain-specific modelling language, a reference implementation has to be put in place (see Figure 4, ScreenMenu and ScreenSubMenu). These reference implementations form the basis classes for the code generator screen classes that are produced (see Figure 4, Screen1, Screen2 and Screen3) and offer functionalities for the screen types (e.g. special menu functions). For each further screen type of the DSL, a Java class has to be implemented. The generated classes can be integrated into the domain framework with a configuration file (see Figure 4, Config file) that

is also generated by the code generator. On the one hand the configuration file connects the screens with the domain framework, and on the other hand it assigns a design to each screen. Thus the design component is connected with the two dimensions content and behaviour.

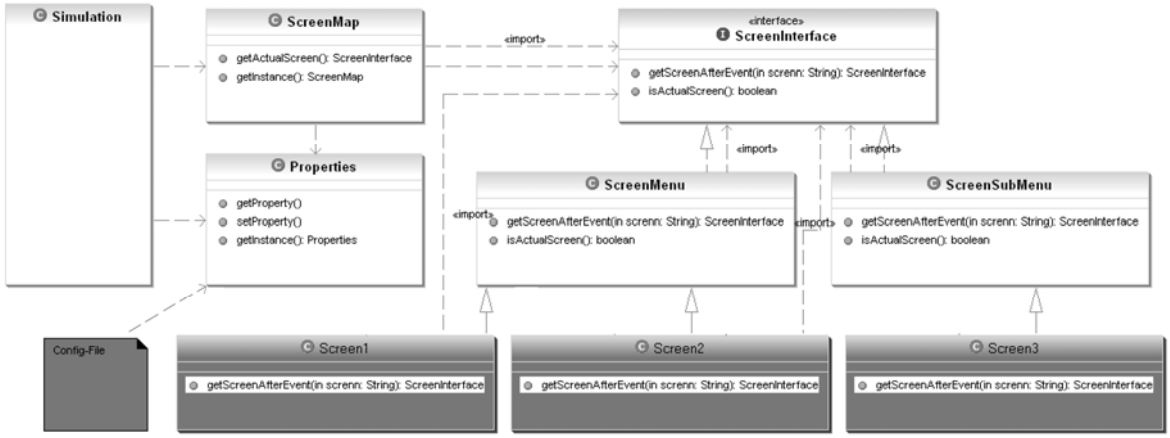

Figure 4. Simplified representation of the domain framework with UML.

Finally, a simulation of the model is generated at the push of a button (see Figure 5). This prototype can be interactively experienced, e.g. with stakeholders and end-users, in order to evaluate the simulation. For providing traceability, the actual system states of the simulation are highlighted in parallel to the running simulation in the modelling environment (see Figure 5, Screens 1-3).

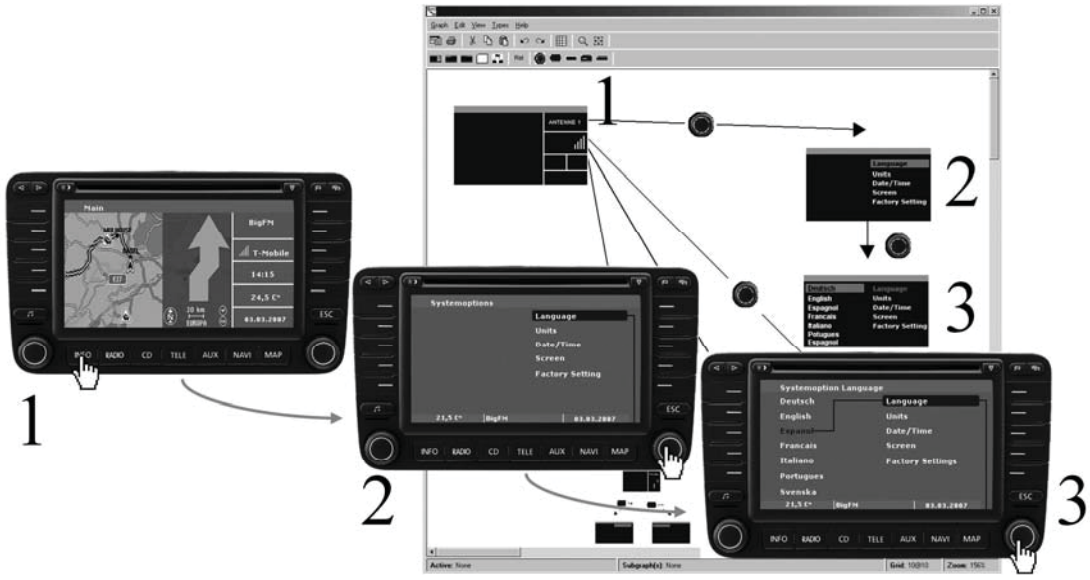

Figure 5. Interactive simulation on a developer PC.

4. SUMMARY AND CONCLUSION

Our experience reveals that model-based approaches provide promising concepts for overcoming today's urgent problems in development processes for corporate software systems (Bock 2007). Thus, visual domain-specific languages explicitly capture experts' knowledge and experience, since this is used for identifying and modelling essential domain concepts. With the help of languages created in this way, new team members can become acquainted with company-specific standards more easily and can thus be integrated in

interdisciplinary development teams significantly more easily. Moreover, domain-specific modelling, together with the tool-chain architecture presented, enables clients to create tailor-made tool support that offers all developers the appropriate level of abstraction for their individual development tasks. Furthermore, problems have to be solved only once at a high level of abstraction and not – as before – once in the implementation level and a second time for documentation. Specifications i.e. requirements engineering can therefore be established as the central backbone of model-based development processes, with significant potential for clients and their collaboration with suppliers. If formal, electronic – and thus machine readable – specifications can be exchanged between all stakeholders, the specification problem as well as the communication problem in traditional development processes can be overcome. One opportunity to evaluate the improvements of the process with the tool-chain is the comparison of the process with and without the tool-chain. Therefore we intend to review the two processes with the DATech[6] testing manual for Usability-Engineering processes.

In principle, the concepts presented here can be adopted for any other domain besides the UI design of automotive embedded systems. For instance, a visual domain-specific language could be created for specifying the structure of corporate websites or the information flow in a business process. Despite this flexibility and the potential benefits, experience from a pilot project shows that current meta-CASE-tools can be improved. In particular, developers would expect interaction patterns from standard office applications e.g. auto layout, grids, object inspectors and tree views for object hierarchies. Additionally, if these tools provided better graphical capabilities, the GUI builder would not have to be integrated via the domain framework, and layout could also be specified with a domain-specific language. This would be another important step in reducing complexity in usually heterogeneous IT landscapes.

Overall, meta-modelling offers promising beginnings for a unification of engineering disciplines. As demonstrated with the tool chain presented here, this modelling approach is consistent with agile principles and practice. This offers an opportunity for bringing RE, SE and UE closer together, a convergence that is badly needed for coping with the technical and organizational complexity of interdisciplinary and networked development processes for software systems.

REFERENCES

Bock, C., 2007, Model-Driven HMI Development: Can Meta-CASE Tools do the Job? I*n Proceedings of the 40th Annual Hawaii International Conference*, Waikoloa, HI, USA.

Constantine, Larry L., 2002, Process Agility and Software Usability: Toward Lightweight Usage-Centered Design. *Information Age*, Vol. 8(2)

Horrocks, I., 1999. Constructing the user interface with statecharts. Addison-Wesley, Harlow.

Luoma, J. et al, 2004, Defining Domain-Specific Modeling Languages: Collected Experiences. Jyväskylä, Finnland.

Memmel, T. et al, 2007, Model-driven prototyping for corporate software specification, In P*roceedings of the Engineering Interactive Systems* (EIS) 2007. Salamanca, Spain, 2007.

Potts, C. et al., 1994, Inquiry Based Requirements Analysis, *IEEE Software*, Volume 11, Issue 2, pp. 21-32.

Sendall, S., Kozaczynski W., 2003, Model Transformation – the Heart and Soul of Model-Driven Software Development. IC Tech Report, 2003.

Speliers, J-P., 2004, Making model-based code generation work. *Embedded Systems Europe*, August / September 2004

Standish Group, 2003. Chaos Reports 2003: http://www.standishgroup.com

Zave, P. and Jackson, M., 1997, Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, Volume 6, Issue 1, pp. 1-30.

[6] DATech - Deutsche Akkreditierungsstelle Technik GmbH / German Accreditation Body for Technology