Supporting Evolution by Models, Components, and Patterns

Isabelle Côté and Maritta Heisel
Universität Duisburg-Essen
{Isabelle.Cote, Maritta.Heisel}@uni-duisburg-essen.de

Abstract: Evolvability is of crucial importance for long-lived software, because no software can persist over a long time period without being changed. We identify three key techniques that contribute to writing long-lived software in the first place: models, patterns, and components. We then show how to perform evolution of software that has been developed according to a model-, pattern-, and component-based process. In this situation, evolution means to transform the models constructed during development or previous evolutions in a systematic way. This can be achieved by using specific operators or rules, by replacing used patterns by other patterns, and by replacing components.

1 Introduction

As pointed out by Parnas [Par94], software ages for two reasons: first, because it is changed. These changes affect the structure of the software, and, at a certain point, further changes become infeasible. The second reason for software aging is *not* to change the software. Such software becomes outdated soon, because it does not reflect new developments and technologies.

We can conclude that software evolution is indispensable for obtaining long-lived software. However, the evolution must be performed in such a way that it does not destroy the structure of the software. In this way, the aging process of the software can be slowed down.

An evolution process that does not destroy the structure of the software first of all requires software that indeed possesses some explicit structure that can be preserved. Hence, different artifacts (not only the source code) should be available. In conclusion, to avoid the legacy problems of tomorrow [EGG$^+$09], we first need appropriate development processes that provide a good basis for future evolutions. Second, we need systematic evolution approaches that can make use of that basis.

In this paper, we first point out that the use of models, patterns, and components are suitable to provide the basis that is necessary for an evolution that does not contribute to software aging. Second, we briefly describe a specific process (called ADIT: Analysis, Design, Implementation, Testing) that makes use of these techniques. Third, we sketch how evolution can be performed for software that was developed using ADIT. There, replay of development steps as well as model transformations play a crucial role.

The rest of the paper is organized as follows: Section 2 discusses the role of models, patterns, and components for the construction of long-lived software. Section 3 introduces the development process ADIT. How the ADIT artifacts can be used to support evolution is shown in Sect. 4. Related work is discussed in Sect. 5, and we conclude in Sect. 6.

2 Models, Patterns, and Components

Models, patterns, and components are all relatively recent techniques that have turned out to be beneficial for software development. Models provide suitable abstractions, patterns support the re-use of development knowledge, and components allow one to assemble software from pre-fabricated parts. These three techniques contribute to the maturing of the discipline of software technology by introducing engineering principles that have counterparts in other engineering disciplines.

2.1 Models

The idea of model-based software development is to construct a sequence of models that are of an increasing level of detail and cover different aspects of the software development problem and its solution. The advantage of this procedure is that it supports a separation of concerns. Each model covers a certain aspect of the software to be developed, and ignores others. Hence, to obtain specific information about the software, it suffices to inspect only a subset of the available documentation.

Using models contributes to developing long-lived software, because the models constitute a detailed documentation of the software that is well suited to support evolution. Of course, the models and the code must be kept consistent. The process we describe in Sect. 4 guarantees that this is the case, because it first adjusts the models, before the code is changed.

Today, the Unified Modeling Language [For06] is commonly used to express the models set up during a model-based development process. For UML, extensive tool support is available. The ADIT process described in Sect. 3 mostly makes use of UML notations.

2.2 Patterns

Patterns are abstractions of software artifacts (or models). They abstract from the application-specific parts of the artifact and only retain its essentials. Patterns are used by instantiation, i.e., providing concrete values for the variable parts of the pattern.

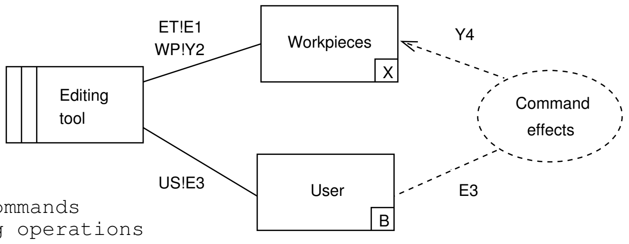
Patterns exist not only as design patterns [GHJV95] (used for fine-grained software design), but for every phase of software development, including requirements analysis [Jac01, Fow97], architectural design [SG96], implementation [Cop92], and testing [Bin00].

Since patterns can be regarded as templates for software development models, model- and pattern-based software development approaches fit very well together. Patterns further enhance the long-livedness of software: first, since the purpose of the different patterns is known, the use of patterns supports program comprehension. Second, patterns enhance the structure of software, e.g., by decoupling different components. Thus, evolution tasks can be performed without tampering too much with the software's structure.

Problem Frames [Jac01] are patterns that can be used in requirements analysis. Since they are less known than the other kinds of patterns just mentioned, we briefly describe them in the following. The ADIT process makes use of problem frames.

2.2.1 Problem Frames

Problem frames are a means to describe software development problems. They were invented by Jackson [Jac01], who describes them as follows: *"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the* *characteristics of its domains, interfaces and requirement."* Problem frames are described by *frame diagrams*, which consist of rectangles, a dashed oval, and links between these (see frame diagram in Fig. 1). All elements of a problem frame diagram act as placeholders which must be instantiated by concrete problems. Doing so, one obtains a problem description that belongs to a specific problem class.



```
E3:  user commands
E1:  editing operations
Y2:  workpieces status
Y4:  workpieces properties
```

Figure 1: *Simple workpieces* problem frame

Plain rectangles denote *problem domains* (that already exist in the application environment), a rectangle with a double vertical stripe denotes the *machine* (i.e., the software) that shall be developed, and *requirements* are denoted with a dashed oval. The connecting lines between domains represent interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation `US!E3` means that the phenomena in the set `E3` are controlled by the domain `User`. A dashed line represents a requirements reference. It means that the domain is mentioned in the requirements description. An arrow at the end of such a dashed line indicates that the requirements constrain the problem domain. In Fig. 1, the `Workpieces` domain is constrained, because the `Editing tool` has the role to change it on behalf of user commands for achieving the required `Command effects`.

Each domain in a frame diagram has certain characteristics. In Fig. 1 the `X` indicates that the corresponding domain is a *lexical* domain. A lexical domain is used for data representations.A `B` indicates that a domain is *biddable*. A biddable domain usually represents people [Jac01].

Problem frames support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing the problem in depth. Thus, the problem frame approach is much more than a mere notation.[1] Other problem frames besides simple workpieces frame are *required behaviour*, *commanded behaviour*, *information display*, and *transformation*.

After having analyzed the problem, the task of the developer is to construct a *machine* based on the problem described via the problem frame that improves the behavior of the environment it is integrated in, according to its respective requirements.

2.3 Components

Component-based development [CD01, SGM02] tries to pick up principles from other engineering disciplines and construct software not from scratch but from pre-fabricated

[1]The diagrams used in the problem frame approach can easily be translated to UML class models, using a number of stereotypes. For a UML metamodel of problem frames, see [HHS08].

parts. Here, interface descriptions are crucial. These descriptions must suffice to decide if a given component is suitable for the purpose at hand or not. It should not be necessary (and, in some cases, it may even be impossible) to inspect the code of the component.

Using components makes it easier to construct long-lived software, because components can be replaced with new ones that e.g. provide enhanced functionality, see [HS04, LHHS07, CHS09]. Again, it becomes apparent that evolvability and long-livedness are deeply intertwined.

Component-based software development fits well with model- and pattern-based development: components can be described by models, and they can be used as instances of architectural patterns.

3 The ADIT development process

In the following, we describe the original development process ADIT that serves as basis for our evolution method in Sect. 4. ADIT is a model-driven, pattern-based development process also making use of components. In this paper we consider the analysis and design phases. To illustrate the different steps within the phases, we briefly describe what the purpose of the different steps is and how they can be realized. We also indicate in bold face the model, pattern, and component techniques that are relevant for the respective step. Should any of the three mentioned techniques not be described in a certain step, then the technique is not used in the respective step.

A1 Problem elicitation and description
To begin with, we need *requirements* that state our needs. Requirements are expressed in natural language, for example "A guest can book available holiday offers, which then are reserved until payment is completed.", "A staff member can record when a payment is received." In this step, also *domain knowledge* is stated, which consists of *facts* and *assumptions*. An example of a fact is that each vacation home can be used by only one (group of) guests at the same time. An example of an assumption is that each guest either pays the full amount due or not at all (i.e., partial payments are not considered). We now must find an answer to the question: "Where is the problem located?". Therefore, the environment in which the software will operate must be described. A **model** which can be used to answer the question is a *context diagram* [Jac01]. Context diagrams are similar to problem diagrams but it does not take requirements references into account. A context diagram for our vacation rentals example is shown in Fig. 3. Thus, the output of this step is: the context diagram, the requirements and the domain knowledge.

A2 Problem decomposition
We answer the question: "What is the problem?" in this second step. To answer the question, it is necessary to decompose the overall problem described in A1 into small manageable subproblems. For decomposing the problem into subproblems, related sets of requirements are identified first. Second, the overall problem is decomposed by means of **decomposition operators**. These operators are applied to context diagram. Examples of such operators are *Leave out domain* (with corresponding interfaces), *Refine phenomenon*, and *Merge several domains into one domain* . After applying the decomposition operators we have our set of subproblems with the corresponding operators that were applied to obtain the different subproblems. Furthermore, the subproblems should belong to known classes of software development problems, i.e., they should *fit to* **patterns** for such known classes of problems. In our case it should be possible to fit them to *problem frames*. Fitting a problem to a problem frame is achieved by instantiating the respective frame diagram.

Instantiated frame diagrams are called *problem diagrams*. These serve as **models** for this second analysis step. Thus, the output of this step is a set of problem diagrams being instances of problem frames.

A3 Abstract software specification
In the previous step, we were able to find out what the problem is by means of problem diagrams. However, problem diagrams do not state the order in which the actions, events, or operations occur. Furthermore, we are still talking about requirements. Requirements refer to problem domains, but not to the machine, i.e., the software that should be built. Therefore, it is necessary to transform requirements into specifications (see [JZ95] for more details). We use UML sequence diagrams as **models** for our specifications. Sequence diagrams describe the interaction of the machine with its environment. Messages from the environment to the machine correspond to *operations* that must be implemented. These operations will be specified in detail in Step A5. The output of this step is a set of specifications for each subproblem.

A4 Technical infrastructure
In this step, the technical infrastructure in which the machine will be embedded is specified. For example, a web application may use the Apache web server. The notation for the **model** in this step is the same as in A1, namely a context diagram. As it describes the technical means used by the machine for communicating with its environment, we refer to it as *technical context diagram*. In this step we can make use of **components** for the first time. Usually, we rely on the APIs of those prefabricated components in order to describe the technical means, e.g., on the API of the Apache web server.

A5 Operations and data specification
The **models** set up in this step are class diagrams for the internal data structures and pre- and postconditions for the operations identified in step A3. These two elements constitute the output of this step.

A6 Software life-cycle
In the final analysis step, the overall behavior of the machine is specified. Here, behavioral descriptions such as life-cycle expressions [CAB+94] can be used as a **model**. In our case this means that in particular, the relation between the sequence diagrams associated to the different subproblems is expressed explicitly. We can relate the sequence diagram sequentially, by alternative, or in parallel.

With the software life-cycle as output we conclude the analysis phase and move on to the design phase.

D1 Software architecture
The first step of the design phase is aimed at giving a coarse-grained structure to the software. This structure can be illustrated by using structural description **models**, such as UML 2.0 composite structure diagrams. We assign a candidate architecture to each subproblem, making use of architectural **patterns** for problem frames [CHH06]. Thus, we obtain a set of sub-architectures. Some **components** within these sub-architectures are pre-fabricated or pre-existing, e.g. for the web application example we make use of a pre-existing SMTP-Client. The architectural patterns lead us to a layered architecture consisting of an application layer, an interface abstraction layer (IAL) abstracting technical phenomena to application related ones, and a hardware abstraction layer (HAL) representing hardware drivers. The overall architecture must be derived from the subproblem architectures. The crucial point of this step is to decide if two components contained in

different subproblem architectures should occur only once in the global architecture, i.e., if they should be merged. To decide this question, we make use of the information expressed in Step A6 and by applying **merging rules**. An example for such a rule is "Adapters and storage components belonging to the same physical device or data storage are merged." These rules are described in more detail in [CHH06].

Figure 2 illustrates the global software architecture of our vacation rentals example, i.e. the output of this step. Furthermore, we have a first skeleton of the interfaces connecting the different components in our architecture by taking advantage of the information of the analysis phase [HH09].

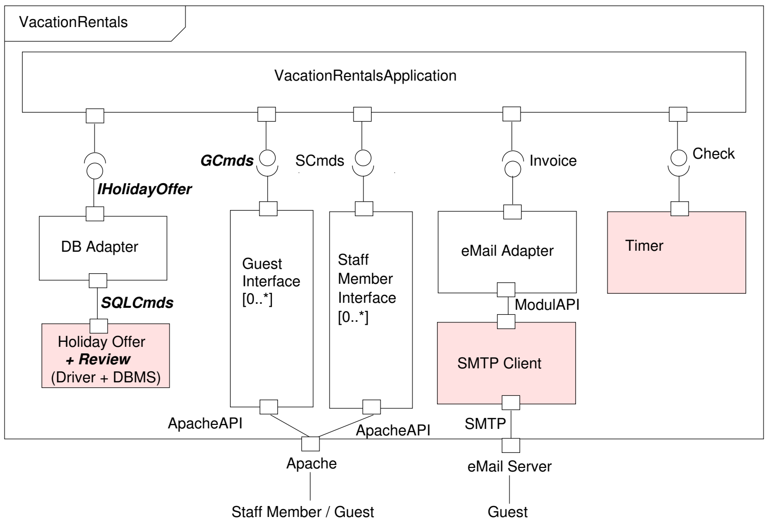


Figure 2: Global software architecture of vacation rentals

D2/ D3/ D4 Inter-component interaction/ intra-component interaction/ complete component or class behavior
Steps D2-D4 treat the fine-grained design. In these steps, the internal structure of the software is further elaborated. Several **patterns** can be applied in these steps, such as design patterns, patterns for state machines, etc.

Note that we are not treating these steps further in this paper.

4 Evolution Method

During the long lifetime of a software it is necessary to modify and update it to meet new or changed requirements and/or environment to accommodate it to the changing environment where it is deployed in. This process is called *software evolution*. The new requirements that should be met are called *evolution requirements*. An example for such an evolution requirement is "Guests can write a review, after they have left their vacation home." Modified or additional domain knowledge is referred to as aD. An example for some new domain knowledge is the additional assumption that "Guests write fair reviews".

We define a corresponding evolution step for each ADIT step. These steps address the special needs arising in software evolution by providing operators leading from one model to another and an explicit tracing between the different models. Basically, we perform a replay of the original method, adding some support for software evolution.

In the following we illustrate the steps to be performed for software evolution. As an example, we evolve the vacation rentals application introduced in Sect. 3.

EA0 Requirements relation
Not all development models will be affected by the evolution task. Therefore, we first have to identify those models that are relevant. For that purpose, we relate the evolution requirements to the original requirements. We identified several relations the original and evolution requirements may share:

- **similar**: a functionality similar to the one to be incorporated exists.
- **extending**: an existing functionality is refined or extended by the evolution task.
- **new**: the functionality is not present yet, and it is not possible to find anything that could be similar or extended.
- **replacing**: the new functionality replaces the existing one.

As a means of representation, we use a table. This table –together with other tables that are created during the process – serve for tracing purposes. An example of such a representation is shown in Tab. 1.

The entry "recordPayment" in the first row refers to the requirement involving the staff member which has been introduced in Sec. 3. The entry "writeReview" in the first column refers to the above mentioned evolution requirement. The table is read as follows: "writeReview" is similar to "recordPayment".

	...	recordPayment	...
writeReview		similar	
⋮		⋮	

Table 1: Excerpt of relation between requirements and evolution requirements for vacation rentals

The requirements sharing a relation with the evolution requirements are collected to form the set of relevant requirements (*rel_set*). This set then constitutes the focus for our further investigation.

EA1 Adjust problem elicitation and description
We use the output of A1, i.e., context diagram as input for the first step of the evolution method. We revise this context diagram by incorporating the new/changed requirements and domain knowledge to it. To support the engineers, we defined evolution operators, similar to the original operators, to ease modifying the context diagram. The **operators** relevant for this step are (for more details refer to [CHW07]):

add new domain – A new domain has to be added to the context diagram.

modify existing domain – A domain contained in the context diagram has to be modified, e.g. by splitting or merging.

add new phenomenon – A new phenomenon is added to an interface of the context diagram.

modify existing phenomenon – An existing phenomenon has to be modified in the context diagram, e.g. by renaming.

In some cases, it may also occur that neither domains nor shared phenomena are newly introduced. Then, no changes to the context diagram are necessary at this place, but the new requirements/domain knowledge may require changes in later steps.

The resulting context diagram now represents the new overall problem situation (cf. Fig. 3). The modifications are highlighted through bold-italic font, e.g., a phenomenon *writeReview* has been added to the interface between the domains guest and vacation rentals (operator add new phenomenon). Furthermore, add new domain and add new phenomenon have been applied to introduce the domain *Review* with the corresponding phenomenon *writingReview*.

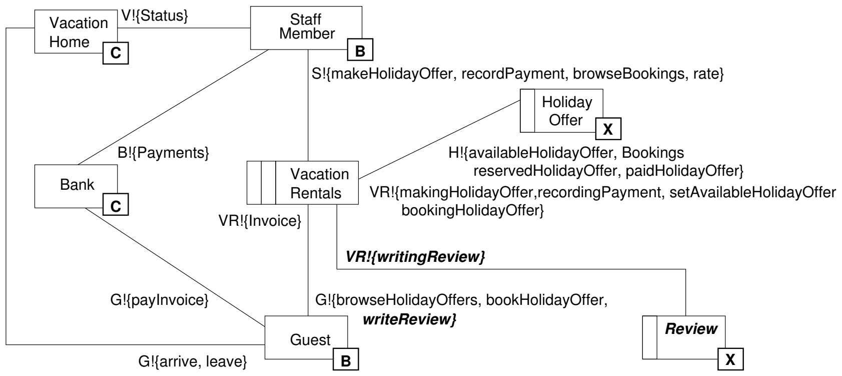

Vacation Home
C
V!{Status}
Staff Member
B
S!{makeHolidayOffer, recordPayment, browseBookings, rate}
Holiday Offer
X
H!{availableHolidayOffer, Bookings reservedHolidayOffer, paidHolidayOffer}
B!{Payments}
Bank
C
Vacation Rentals
VR!{makingHolidayOffer,recordingPayment, setAvailableHolidayOffer bookingHolidayOffer}
VR!{Invoice}
VR!{writingReview}
G!{payInvoice}
G!{browseHolidayOffers, bookHolidayOffer, writeReview}
Guest
B
Review
X
G!{arrive, leave}

Figure 3: Context diagram with revisions

EA2 Adjust problem decomposition

In this step we take the output of A2, i.e., the resulting set of problem diagrams, as well as the operators that were applied to the context diagram for the decomposition as input. Furthermore, we use both the context diagram created in EA1, as well as the rel_set of EA0 as additional input. As we modified the context diagram by applying some operators, this has impacts on the problem decomposition, as well. Therefore, it is necessary to investigate the existing subproblems. We again define evolution operators to guide the engineer.

Examples of evolution **operators** for this step are:

integrate eR in existing problem diagram – New domains and associated shared phenomena may be added to an existing problem diagram.

eR cannot be added to existing subproblem - create new one – The eR is assigned to a given subproblem, but the resulting subproblem then gets too complex. Hence, it is necessary to split the subproblem into smaller subproblems.

More details on these operators can be found in [CHW07].

In addition it is necessary to check whether the resulting problem diagrams still fit to problem frames or, for new subproblems, to find an appropriate frame to be instantiated, e.g. via **operators** such as *fit to a problem frame* or *choose different problem frame*.

Figure 4 shows the problem diagram for the evolution requirement “writeReview” (operator *eR cannot be added to existing subproblem - create new one*). As we know that it is similar to “recordPayment” we check whether it is possible to apply the same problem frame. In this case, this is the problem frame *simple workpieces* (cf. Fig. 1). It is obvious that our problem diagram is an instance of that problem frame (operator *fit to a problem frame*).

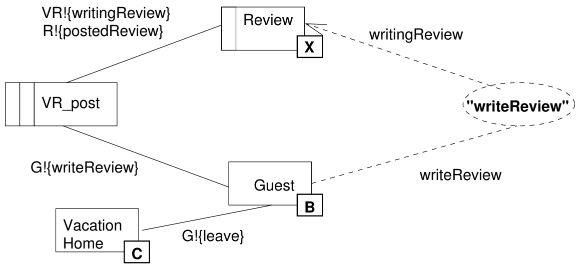


Figure 4: Problem diagram for “writingReview”

EA3 Adjust abstract software specification
The sequence diagrams of A3 serve as input for this step, together with the results of EA2. Whenever an existing sequence diagram has to be investigated, the following cases may occur:

- Operator *integrate eR in existing problem diagram* was applied.

  Operator name: use existing sequence diagram

  The existing diagram has to be modified in such a way that it can handle the additional behavior by adding messages, domains etc. to the corresponding sequence diagram. The changes reflect the modifications made in EA2. It may also be necessary to add new sequence diagrams, as well.

- The parameters of existing messages must be adapted.

  Operator name: modify parameter

  Existing parameters must be either modified (rename), extended (add) or replaced. Otherwise, the sequence diagram remains unchanged.

- Operator *eR cannot be added to existing subproblem - create new one* was applied.

  Operator name: no operator

  The original ADIT step must be applied.

EA4 Adjust technical software specification
The original technical context diagram is adapted to meet the new or changed situation. The procedure is the same as in A4.

EA5 Adjust operations and data specification
The actions performed in EA3 trigger the modification. For example, a message on the sequence diagram as been modified. These changes are then also made in the corresponding operations in this step. For newly introduced operations, the original method is applied.

EA6 Adjust software life-cycle
New relations need to be incorporated. Existing relations need to be adapted, accordingly. However, the behavior that has not been altered by the evolution task must be preserved.

ED1 Adjust software architecture
We use the revised subproblems of Step EA2 as input. For our example we were able to apply the same architectural style for “writeReview” as we did for “recordPayment”. Therefore, we can apply the same merging rules, as well. Figure 2 illustrates the resulting global architecture. The changes are indicated in italics and bold face.

The interfaces are adapted according to the changes made in the analysis phase based on **operators** such as *modify interface*, *add new component*, or *add new interface*. For example, we introduced a new message “writeReview()” in step EA3 (as consequence of operator *eR cannot be added to existing subproblem - create new one*). By doing a replay of the merging rules, we know, that it is necessary to extend the existing interfaces *IHolidayOffer* and *GCmds* (operator *modify interface*). As writing a review is performed after guests have taken the holiday offer the component *Review* can be merged with the existing component *Holiday Offer* (operator *modify component*) resulting in a modification of the corresponding interface *SQLCmds* (operator *modify interface*), as well.

5 Related Work

This work takes up ideas from modern software engineering approaches and processes, such as the Rational Unified Process (RUP) [JBR99], Model-Driven Architecture (MDA) [MSUW04], and Service-Oriented Architecture (SOA) [Erl05]. All these approaches are model-driven, which means that in principle, an evolution process for them can be defined in a similar way as for the ADIT process.

The work of O’Cinnéide and Nixon [ON99] aims at applying design patterns to existing legacy code in a highly automated way. They target code refactorings. Their approach is based on a semi-formal description of the transformations themselves, needed in order to make the changes in the code happen. They describe precisely the transformation itself and under which pre- and postconditions it can successfully be applied.

Our approach describes and applies model transformation in a rather informal way. The same is true for the transformation approaches cited above. Czarnecki and Helsen give an overview of more formal model transformation techniques [CH03].

Researchers have used versions and histories to analyze different aspects considering software evolution. Ducasse et al. [DGF04] propose a history meta-model named HISMO. A history is defined as a sequence of versions. The approach is based on transformations aimed at extracting history properties out of structural relationships. Our approach does not consider histories andhow to analyze them. In contrast, we introduce a method for manipulating an existing software and its corresponding documentation in a systematic way to perform software evolution.

ROSE is the name of a tool created by Zeller et al. [ZWDZ05], which makes use of data mining techniques to extract recurring patterns and rules that allow to offer advice for new evolution tasks. The tool can propose locations where a change might occur based on the current change, help to avoid incomplete changes, and detect coupling that would not be found by performing program analysis. It does, however, not provide help on what to do once a potential location has been identified. With our method we intend to provide help to systematically modify source code after the corresponding part has been located.

Detecting logical coupling to identify dependencies among modules etc. has been the research topic of Gall et al.[GHJ98]. Those dependencies can be used to estimate the effort needed to carry out maintenance tasks, to name an example. Descriptions in change reports are used to verify the detected couplings. The technique is not designed to change the functionality of a given software.

Others investigate software evolution at run-time [Piz02]. Evolving a software system at run-time puts even more demands on the method used than ordinary software systems. Our approach does not take run-time evolution into account.

Sillito et al. [SMDV06] conducted a survey to capture the main questions programmers ask when confronted with an evolution task. These fit well to our method as they can be used to refine the understanding of the software at hand especially in later phases.

We agree with Mens and D'Hondt [MD00] that it is necessary to treat and support evolution throughout all development phases. They extend the UML meta-model by their so-called evolution contracts for that reason. The aim is to automatically detect conflicts that may arise when evolving the same UML model in parallel. This mechanism can very well be integrated into our method to enhance the detection of inconsistencies and conflicts. Our approach, however, goes beyond this detection process. It strives towards an integral method for software evolution guiding the software engineer in actually performing an evolution task throughout all development phases.

The field of software evolution cannot be examined in isolation as it has contact points with other disciplines of software engineering. An example is the work of Demeyer et al. [DDN02]. They provide a pattern system for object-oriented reengineering tasks. Since software evolution usually involves some reengineering and also refactoring [Fow00] efforts, it is only natural that known and successful techniques are applied in software evolution, as well. Software evolution, however, goes beyond restructuring source code. Its main goal is to change the functionality of a given software.

Furthermore, software evolution can profit from the research done in the fields of feature location e.g. [ESW06, KQ05], re-documentation, e.g. [CY07, WTM+95], and agile development processes such as extreme programming [Bec99].

Finally, we have to mention another paper of ours on pattern-based software evolution for component-based systems [CHS09]. In this paper, we give architectural evolution patterns that can be used to evolve component architectures.

6 Conclusions

In this paper, we have pointed out that using models, patterns, and components is a promising approach to develop long-lived software. However, long-lived software needs to undergo evolution. Therefore, we have shown how software that was developed according to a model-/pattern-/component-based process can be evolved in a systematic way. We have applied our method successfully on several software systems amongst them open source software such as *Doxygen* [Hee09].

In the future, we intend to elaborate on the model transformations that are needed to perform the evolution steps. We will formalize the model transformations and also provide further tool support for the evolution process. Furthermore, we plan on conducting further evolution tasks on other open source projects to validate the method as well as the tool.

References

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Bin00] Robert Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 2000.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994. (out of print).

[CD01] John Cheesman and John Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proc. of the 2nd OOPSLA Workshop on Generative Techniques in the Context of MDA*, 2003.

[CHH06] C. Choppy, D. Hatebur, and M. Heisel. Component composition through architectural patterns for problem frames. In *Proc. XIII Asia Pacific Software Engineering Conf.*, pages 27–34. IEEE Computer Society, 2006.

[CHS09] Isabelle Côté, Maritta Heisel, and Jeanine Souquières. On the Evolution of Component-based Software. In *4th IFIP TC2 Central and Eastern European Conf. on Software Engineering Techniques CEE-SET 2009*. Springer-Verlag, 2009. to appear.

[CHW07] Isabelle Côté, Maritta Heisel, and Ina Wentzlaff. Pattern-based Exploration of Design Alternatives for the Evolution of Software Architectures. *Int. Journal of Cooperative Information Systems, World Scientific Publishing Company*, Special Issue of the Best Papers of the ECSA'07, December 2007.

[Cop92] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[CY07] Feng Chen and Hongji Yang. Model Oriented Evolutionary Redocumentation. In *COMPSAC '07: Proc. of the 31st Annual Int. Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*, pages 543–548, Washington, DC, USA, 2007. IEEE Computer Society.

[DDN02] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DGF04] Stéphane Ducasse, Tudor Gîrba, and Jean-Marie Favre. Modeling Software Evolution by Treating History as a First Class Entity. In *Proc. on Software Evolution Through Transformation (SETra 2004)*, pages 75–86, Amsterdam, 2004. Elsevier.

[EGG$^+$09] G. Engels, M. Goedicke, U. Goltz, A. Rausch, and R. Reussner. Design for Future – Legacy-Probleme von morgen vermeidbar? *Informatik-Spektrum*, 2009.

[Erl05] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[ESW06] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. In *Journal of Systems and Software*, 2006.

[For06] UML Revision Task Force. *OMG Unified Modeling Language: Superstructure*, April 2006. http://www.omg.org/docs/ptc/06-04-02.pdf.

[Fow97] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.

[Fow00] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of Logical Coupling Based on Product Release History. In *ICSM '98: Proc. of the Int. Conf. on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[Hee09] D. van Heesch. Doxygen - A Source Code Documentation Generator Tool, 2009. `http://www.stack.nl/~dimitri/doxygen`.

[HH09] D. Hatebur and M. Heisel. Deriving Software Architectures from Problem Descriptions. In Jürgen Münch and Peter Liggesmeyer, editors, *Workshop Modellgetriebene Softwarearchitektur – Evolution, Integration und Migration (MSEIM), Software Engineering 2009*, LNI P-150, pages 383–392, Bonn, 2009. Bonner Köllen Verlag.

[HHS08] Denis Hatebur, Maritta Heisel, and Holger Schmidt. A Formal Metamodel for Problem Frames. In *Proc. of the Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, volume 5301, pages 68–82. Springer Berlin / Heidelberg, 2008.

[HS04] Maritta Heisel and Jeanine Souquières. Adding Features to Component-Based Systems. In M.D. Ryan, J.-J. Ch. Meyer, and H.-D. Ehrich, editors, *Objects, Agents and Features*, LNCS 2975, pages 137–153. Springer, 2004.

[Jac01] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JZ95] M. Jackson and P. Zave. Deriving Specifications from Requirements: an Example. In *Proc. 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[KQ05] Rainer Koschke and Jochen Quante. On dynamic feature location. In *ASE '05: Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 86–95, New York, NY, USA, 2005. ACM.

[LHHS07] Arnaud Lanoix, Denis Hatebur, Maritta Heisel, and Jeanine Souquières. Enhancing Dependability of Component-Based Systems. In N. Abdennadher and F. Kordon, editors, *Reliable Software Technologies – Ada Europe 2007*, LNCS 4498, pages 41–54. Springer, 2007.

[MD00] Tom Mens and Theo D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering Journal*, 7(1):39–59, February 2000.

[MSUW04] S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled*. Addison-Wesley Professional, 2004.

[ON99] M. O'Cinnéide and P. Nixon. A Methodology for the Automated Introduction of Design Patterns. In *ICSM '99: Proc. of the IEEE Int. Conf. on Software Maintenance*, page 463, Washington, DC, USA, 1999. IEEE Computer Society.

[Par94] D. L. Parnas. Software aging. In *ICSE '94: Proc. of the 16th Int. Conf. on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Comp. Soc. Press.

[Piz02] M. Pizka. STA – A Conceptual Model for System Evolution. In *Int. Conf. on Software Maintenance*, pages 462 – 469, Montreal, Canada, October 2002. IEEE CS Press.

[SG96] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software*. Pearson Education, 2002. Second edition.

[SMDV06] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proc. of the 14th ACM SIGSOFT Int. Symposium. on Foundation of Software Engineering*, pages 23–34, New York, NY, USA, 2006. ACM.

[WTM$^+$95] K. Wong, S. R. Tilley, H. A. Muller, M. D. Storey, and T. A. Corbi. Structural redocumentation: A case study. *IEEE Software*, 12:46–54, 1995.

[ZWDZ05] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.