Pattern Repositories for Software Engineering Education

Hans-Werner Sehring, Sebastian Bossung, Patrick Hupe,
Michael Skusa, and Joachim W. Schmidt
{*hw.sehring,sebastian.bossung,pa.hupe,skusa,j.w.schmidt*}*@tuhh.de*
Software Systems Institute (STS)
Hamburg University of Science and Technology (TUHH)

Abstract. Modern software engineering attacks its complexity problems by applying well-understood development principles. In particular, the systematic adoption of *design patterns* caused a significant improvement of software engineering and is one of the most effective remedies for what was formerly called the *software crises*. Design patterns and their utilization constitute an increasing body of knowledge in software engineering. Due to their regular structure, their orthogonal applicability and the availability of meaningful examples design patterns can serve as an excellent set of use cases for organizational memories, for software development tools and for e-learning environments.
Patterns are defined and described on two levels [1]: by real-world examples—e.g., textual or graphical content on their principles, best practices, structure diagrams, code etc.—and by conceptual models—e.g., on categories of application problems, software solutions, deployment consequences etc. This intrinsically dualistic nature of patterns makes them good candidates for conceptual content management (CCM). In this paper we report on the application of the CCM approach to a repository for teaching and training in pattern-based software design as well as for the support of the corresponding e-learning processes.

Keywords. Conceptual modeling, content management, design patterns, e-learning

1. Introduction and Motivation

The entire field of modern software engineering (SE) is a diverse and complex endeavor. Many advances had to be made to master what we used to call the software crisis. One important key to success was finally found in software patterns which introduced another level of abstraction to software design processes and decreased the complexity of designs considerably. Since their first publication in the early 90ies [1] design patterns (and the idea of software patterns in general) have been quickly adopted and today are in widespread use.

As modern software systems become ever larger and more complex most software development paradigms agree that some planning-ahead is needed in order to successfully carry out a software development project [2]. While requirements for the application under development are collected during an initial application analysis phase, the subsequent design phase aims at a coarse design of the software. In this light, design

patterns are abstractions over pieces of software with shared design requirements thus helping to exploit accumulated design experience.

Design patterns turn out to be essential pieces of knowledge for software engineers and, consequently, have to be covered by software engineering curricula. As the mere awareness of a design pattern does not enable a student of software engineering to appropriately apply it, the pragmatics of pattern applications should also be taught. This can be achieved by providing pattern definitions together with best-practice pattern applications and by enabling students to relate, publicize and discuss their personal pattern applications in such definitional contexts. Students can thus actively participate in the design process, resulting in an improved learning experience.

In previous projects we have used our Conceptual Content Management (CCM) approach to provide extensional concept definitions and to embed such CCM material into learning environments—albeit in political iconography [3] or art history applications [4]. In this paper we sketch a conceptual model for design patterns and show how previous experiences with active learning can be carried over to SE. We discuss how existing work on patterns (e.g., [1]) can be represented, communicated, and applied by CCM.

The remainder of this paper is organized as follows: We commence with an overview of requirements to SE tools and their relationships with content management in section 2. In section 3 we provide a conceptual model suitable for describing patterns in learning systems and also report on some related work. Section 4 describes pattern learning scenarios with particular emphasis on active learning in conceptual content management systems. We conclude with a summary and outlook in section 5.

2. Requirements for Software Engineering Tools

As a basis for discussion this section contains some remarks on SE processes, in particular on the pattern-based design of software, and we argue that SE—especially the aspect of conserving design knowledge—shares properties of advanced content management tasks.

2.1. Software Engineering Processes

A typical SE process consists of several phases: analysis, design, implementation and test (e.g., [2]). During these phases, numerous artifacts are created. These artifacts stem from different models and coexist for various reasons: different levels of abstraction, perspectives, revisions, or alternate representations.

Typically, artifacts of later phases are created from higher-level artifacts of earlier phases, but dependencies are not linear and difficult to trace: Diverse analysis requirements, platform constraints and modeling acts such as architectural decisions, pattern applications, etc. manifest themselves in later artifacts. While the knowledge about these manifestations is available at the point of decision-making, this information often is not part of an SE process [5]. Instead, such information has to be recorded in additional documentation which accompanies the process. Only recently efforts have begun to interconnect these artifacts at least partially [6]. The conservation and communication of development experience is an issue not systematically addressed by SE. Figure 1 shows a number of typical software engineering artifacts from one of our projects. Superimposed

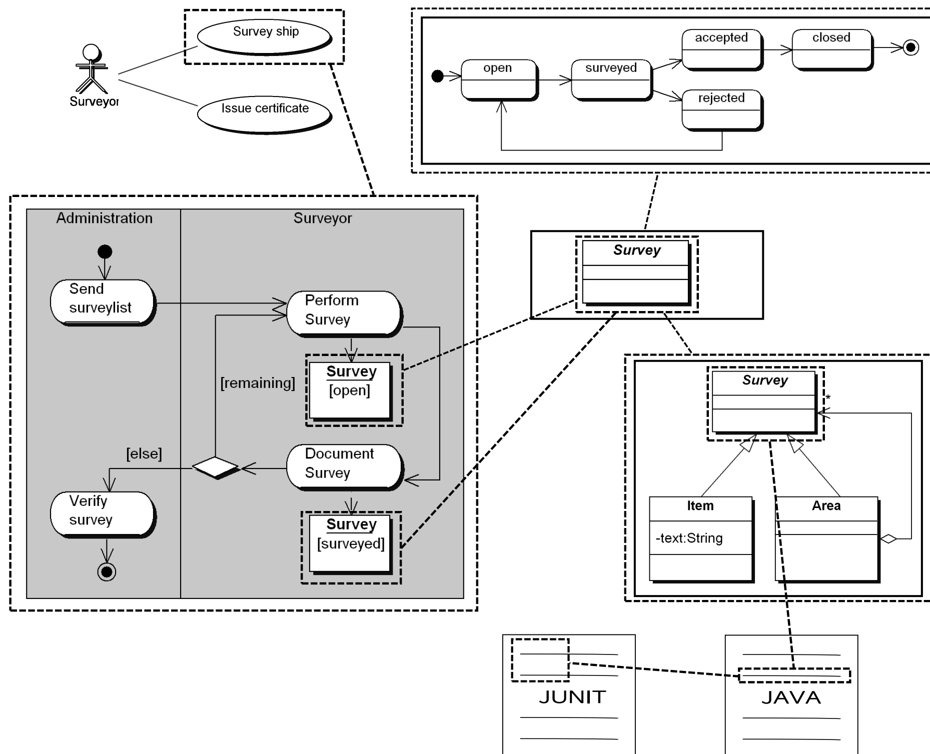

Figure 1. Examples of interrelated software engineering artifacts

onto these are the interrelationships between them (dashed lines), which are neither explicit in the artifacts themselves nor handled by the corresponding tools in a coherent and global manner.

2.2. Design Phase Activities

One of the phases of typical SE processes is that of design. This phase gains much attention because it is the point where reuse of development experience can take place at a high level.

The application of design patterns has become an important member of the set of design activities (figure 2). The reason for this is that such applications are well-understood cases of software design. Design patterns also constitute a growing body of knowledge and best-practices. They can be used to preserve and communicate design experience.

These reasons—design patterns being well-understood and being a medium to communicate development experience—makes them a germane means for teaching software development. We apply patterns in SE education as a first object of study towards a model-based treatment of design patterns. As a first step in this direction we treat specific patterns (see figure 2).

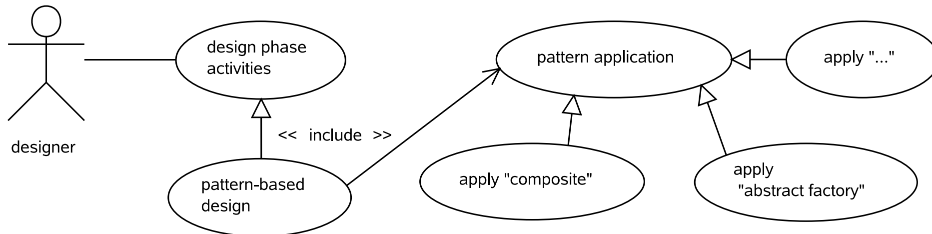

Figure 2. Design use cases

2.3. Software Engineering as a Content Management Activity

We have studied how entities are represented by means of content management with regard to SE [7]. Emphasis is put on the representation of entities that cannot fully be represented by data records alone, especially items that have subjective interpretations. We refer to the research of such content management applications as *conceptual content management* (CCM).

Based on the epistemic observation that neither content nor concepts exist in isolation, CCM is based on the conjoint description of entities by pairs of content representations and conceptual models. For pairs of these two we introduce the notion of *assets*. *Content* is presented by arbitrary multimedia documents. *Concepts* consist of the *characteristic* properties of entities, *relationships* with other assets which describe the interdependencies of entities, and *constraints* on those characteristics and relationships.

The statements of the asset language allow the definition of asset classes, the creation and manipulation of asset instances, and their retrieval. Some examples follow. For a more complete description of the asset language see [8].

The following sketches asset definitions to model an SE processes:

```
model SoftwareEngineering
class SoftwareModel {
  content xmiDocument :org.w3c.dom.Document
  concept relationship classes   :ClassDescription*
          relationship sequences :Sequence*
          constraint operationsDefined
            sequences.objActs.msgs <= classes.operations
            and ... ; matching signatures
          ⋮
}
class Sequence {
  content topNode :org.w3c.dom.Element
  concept relationship objActs :ObjectActivation*
}
class ObjectActivation ...
```

In the example, an XML document is the content of a general `SoftwareModel`, presumably an XMI representation of UML diagrams. Related instances of further asset classes are used to describe parts of the software model—classes, sequences, etc.—in

more detail. Such an asset model can be created by the structural conversion of an existing model for software in general, for example the Meta Object Facility (MOF, see [9]).

SE entities are described by the aforementioned characteristic attributes and relationships. Possible types of content handles and characteristics are determined by an embedded language which also is the target language of the model compiler (currently Java). More extensive use of the asset model is achieved by employing constraints that reflect the rules of the chosen SE process. In the example this is demonstrated by a constraint `operationsDefined` which in addition to the semantics of the UML ensures that for each message sent according to a sequence diagram a method with a matching signature is defined. When this constraint is violated, the corresponding instances can either not be allowed at all, or—in case of an interactive system—the situation can be brought to the attention of the user, e.g., on a list of issues.

Since assets are especially designed to support subjective views, asset classes as well as instances can be *personalized* by means of redefinitions on an individual basis. Based on the above definition of `SoftwareModel` a user can define

```
model MySoftwareEngineering
from SoftwareEngineering import SoftwareModel
class SoftwareModel {
  concept relationship objects :ObjectDescription*
          constraint objects.type <= classes
}
```

to explicitly refer to instances and to require that all of their classes have to be part of the model. Anything that is not mentioned in the personalization remains the same as in the original definition.

For asset redefinitions there is a demand for *openness* and *dynamics*. We call a *CCM system* (*CCMS*) open if it allows users to define assets according to their current information needs. Dynamics is the ability of a system to follow redefinitions of assets at runtime without interrupting the users' work.

To account for dynamics, our approach to CCM consists of three main contributions [10]: an *asset language* for the description of entities by both content and conceptual expressions, a *modularized architecture* for evolving conceptual content management systems, and a *model compiler* which translates expressions given in the asset language into CCMSs without developer intervention.

3. Modeling Design Patterns

This section first gives a brief overview of design patterns in general and approaches to create conceptual models for them. We then show how to model design patterns in CCM.

3.1. Design Patterns

The central idea of design patterns is to capture solutions to recurring problems. In other words, experiences gained by adept programmers are put into a form that is suitable to pass on these experiences to others, thereby eliminating the need for them to relearn the same knowledge "the hard way" by rediscovering them from practice. The exact workings of such human learning mechanisms are currently under research [5]. This

capturing of short-cuts in learning experiences makes design patterns an important part of SE curricula.

Existing work on design patterns (most prominently Gamma et al, [1]) identifies four central elements of a pattern: A name, a context in which it can be applied, the solution it provides, and the consequences that arise from it. However, [1] also acknowledge that there is some subjectivity in design patterns. It is noted that what is and what is not a pattern depends on the individual user's point of view. Other work [11] also points out the common definition of a design pattern being "a solution to a problem in a context" should be extended to also include information about recurrence as well as about teaching, to provide the means to apply the solution to new situations and to notice that an opportunity to do so has arisen in the first place. This is particularly important with respect to teaching patterns, where a definition of the design patterns is not sufficient. We will describe in section 4.3 how these aspects can be handled by CCMSs.

3.2. Pattern Description Languages

The need for a pattern description language arises in several contexts which can broadly be divided into those that aim to support the task of creating software (e.g., by pattern application) and those that inspect existing software (e.g., by pattern discovery).

An early metamodel for pattern-based CASE tools in proposed in [12]. Based on this metamodel, pattern instantiation is proposed, relating available patterns to actual application artifacts. UML also offers the collaboration mechanism which can to some extend be used to model design patterns. By itself this is too weak a model for pattern-based tools, which need additional means to capture semantics, e.g., OCL [13]. Yet other tools let programmers work on different levels of abstraction allowing them to work on the source code as well as to instantiate patterns [14].

Metamodels are also employed to extract patterns from existing software. Corresponding tools identify micro-architectures by modeling classes in roles [15], or they describe design pattern applications [16]. The description is geared towards system supported application of the pattern. Taking this one step further, there are approaches to enforce the use of design patterns, e.g., [17].

All these approaches offer metamodels for design patterns with several different foci, but there is a large overlap at their cores. Commonly no clear distinction between pattern description and the general object-oriented metamodel is made. Our metamodel for patterns is loosely similar to most existing models, but aims to improve the separation of general pattern description, object-oriented metamodel and specific pattern descriptions for learning. It thus allows a fine-grained selection of standard models from which personal derivations are used to provide support for active learning scenarios.

3.3. A CCM Model for Pattern-based Design

Design patterns are generally presented in a semi-structured manner. Gamma et al identify four essential parts of a pattern: name, problem, solution and consequences [1]. Further substructuring of these elements is not prescribed, even though most authors try to adopt a uniform heading structure. However, there are also approaches that fully formalize the description of design patterns such as [18]. This can provide well-defined semantics for the descriptions as well as reasoning on them. However, such formalizations are

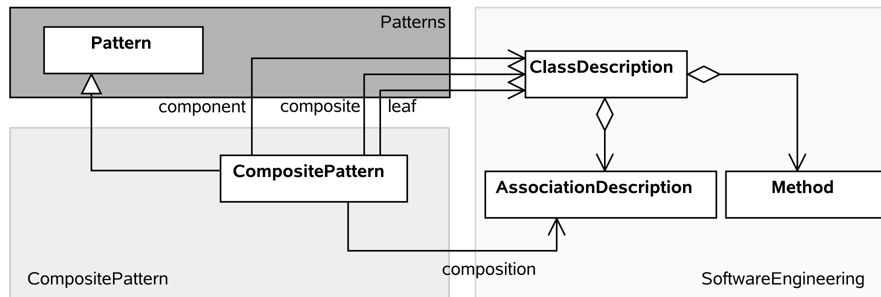

Figure 3. Classes from different models are combined to model patterns. Most of the model is omitted in this figure for conciseness.

hardly useful in teaching patterns as learners need to gain an intuitive understanding to be able to identify situations where the pattern is relevant.

At the root of our conceptual model for patterns is a basic `Pattern` class whose concept offers the four core elements of patterns: `name`—`context`, `solution` and `consequences`—which are described as content in semi-structured documents.

```
model Patterns
class Pattern {
  content name          : String
          problem       : StructuredDocument
          solution      : StructuredDocument
          consequences  : StructuredDocument
  concept relationship   collaborators :ClassDescription
          relationship   collaboratorAspects :ClassMember
}
```

Furthermore, a CCMS allows users to model the patterns to any degree of specificity they want. This can even mean that a class is created for one specific pattern alone if this is required by the learning context. We demonstrate this with the composite pattern:

```
class CompositePattern refines Pattern {
  content
    name :String := "Composite"
  concept
    relationship   component   :ClassDescription
    relationship   composite   :ClassDescription
    relationship   leaf        :ClassDescription
    relationship   composition :AssociationDescription
    constraint specialization1 composite.superClass = component
    constraint specialization2 leaf.superClass = component
    constraint aggregation composition.type = composition
      and composition.source.type = composite
      and composition.target.type = base
    ⋮
}
```

The Composite pattern is characterized by properties which are reflected in the asset class `CompositePattern`. There are classes in three roles: `component`, `composite`, and `leaf`. Both `composite` and `leaf` are subclasses of `component` as defined by the constraints `specialization1` and `specialization2`. Instances of the class which fills the `composite` role aggregate instances of the class filling the role `component` (constraint `aggregation`). For this example assume that there is a class `AssociationDescription` for UML associations with at least the three attributes used: `type` which characterizes the kind of association, as well as `source` and `target` which refer to assets describing the roles of the associated objects.

To capture pattern *applications* in a way that is meaningful to students, the pattern applications have to be put into context of the whole software system they were made in. The respective parts of the application domain of the software can be captured by using a `SoftwareEngineering` model (see section 2.2, [7]). By refining the general pattern model and using the SE model, concrete patterns can be described. Figure 3 gives an overview of the an asset model of the Composite pattern. Instances of this class are used to describe concrete applications of the pattern:

```
model CompositePatternApplication
from SoftwareEngineering import
  ClassDescription, AssociationDescription
from Patterns import CompositePattern
let graphicsPackage := lookfor Package {
  name = "de.tuhh.sts.ltood.figures"
}
let compositeApplication := create CompositePattern {
  problem := ...
  :
  composite :=
    lookfor ClassDescription {
       name = "FigureGroup"
       package = graphicsPackage
    }
  component :=
    lookfor ClassDescription {
       name = "Figure"
       package = graphicsPackage
     }
  leaf :=
    lookfor ClassDescription {
       name="Rectangle"
       package = graphicsPackage
     }
  composition :=
    lookfor Association { source=composite target=component }
}
```

An instance of `CompositePattern` describes a pattern application by providing values for all members. The content members inherited from `Pattern` are filled with

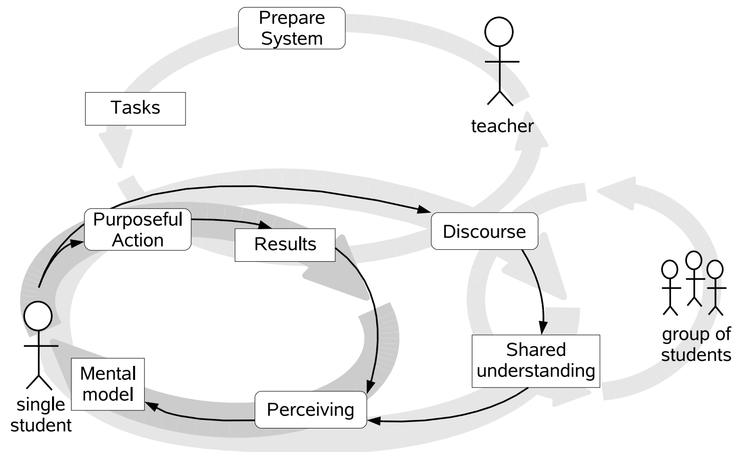

Figure 4. Learning cycle from the learner's point of view.

structured documents describing problem, solution, and consequences. These could, e.g., closely correspond to the descriptions typically found in literature on patterns. The conceptual descriptions of the pattern are also provided by retrieving (through the `lookfor` command) appropriate parts of the existing software model. These parts are then explicitly connected as an application of the composite pattern.

4. Pattern Learning Scenarios

The application of design patterns has become a commonly accepted design activity. Therefore, teaching the most useful patterns has become an important part of the education or training of software developers. However, applying patterns requires tacit knowledge that cannot be studied on a purely theoretical basis. One needs to learn to actively apply patterns. In this section we argue that active learning is supported well by CCM systems.

A pattern is more than a solution to a problem in a context [11]. Especially for teaching purposes improved descriptions of design patterns are needed. Using the open modeling of the CCM approach such improved descriptions can be formulated, in particular including content. By such combined descriptions, CCM allows active learning processes for design pattern application.

4.1. Active Learning Through Open Dynamic CCM

As is witnessed by many taught courses, understanding content and applying the learnt are essential parts of learning [19]. Constructionists interpret learning as the construction of knowledge and not its absorption [20]. Practical application facilitates the lasting storage of information.

Figure 4 (inspired by [21]) depicts such an active way of learning from the point of view of a student. There are two levels, which differ in closeness to the learner as well as in speed of iteration. The inner cycle constitutes active learning. It is usually carried

out by one learner alone. The outer circle describes the interaction with others, learners as well as teachers through discourse in the field of study and a shared, group-based understanding of it.

Active learning requires—as the name suggests—actions by the user in the field of study. These actions will lead to results, which the learner perceives and understands to be failures or advances. This can then be incorporated into the mental model of the field of discourse and used in the next iteration.

Typical e-learning systems support a passive way of learning: There are usually a number of ways to present lessons to learners [22]. The interaction of learner and system is frequently limited to formal testing.

To take the system support beyond this, the system has to be able to adapt to the particular needs of specific (groups of) learners. We refer to this as personalization, which happens at two levels: *content* and *structure* personalization. The former allows users to adapt the content they work with to their own needs or views of the world. This happens without interfering with the work of other users, but the system needs to provide facilities that allow the later exchange of personalized content between (groups of) users. Structure personalization means that users are not confined to the schema provided for the system, but can modify this schema to suit their needs. This aspect is very important in learning to reflect the level of the learner as well as the field of study. For more details on personalization in e-learning applications see [3].

All activities indicated in the learning circle in figure 4 are covered by personalization. It supports discourse through exchange of personalized content and conceptual models with a limited group of peer learners. Most importantly, users are enabled to take action in the system itself and learn through the results of their action. They can recombine artifacts to solve learning problems.

This allows an approach to e-learning in which learners can structurally rework or even extend the subject matter. For example, a lesson can start with a given partial model that the learners are to complete. In doing so, they apply what they previously learned. Thinning out the content is again achieved through personalization (the left-out pieces are not deleted globally but hidden in the personal view). A system setup to support this will be shown in section 4.3.

4.2. Teaching Pattern-oriented Design

This section applies the CCM approach to learning of design patterns. Particular regard is given to the active dimension of learning.

To use CCM for learning in a field of study, the generic activities of the learning cycle in figure 4 have to be backed with specific ones of the participants. Figure 5 shows some use cases from which these activities can be deduced. Here a *pattern expert* prepares case studies as examples or master solutions for a *pattern learner*, or the expert gives exercises to be solved by the learner. The person in the role of the *pattern expert* can but does not have to be the teacher at the same time.

In the previous section it has been mentioned that a teacher can hand partial solutions to learners. By means of personalization each learner can solve exercises individually. Personalization furthermore allows change of existing designs to try out modeling alternatives.

Many times, learning begins at the teacher (in the upper cycle in figure 4) who—in the case of CCM—perpares a system for the particular needs of the learners. This

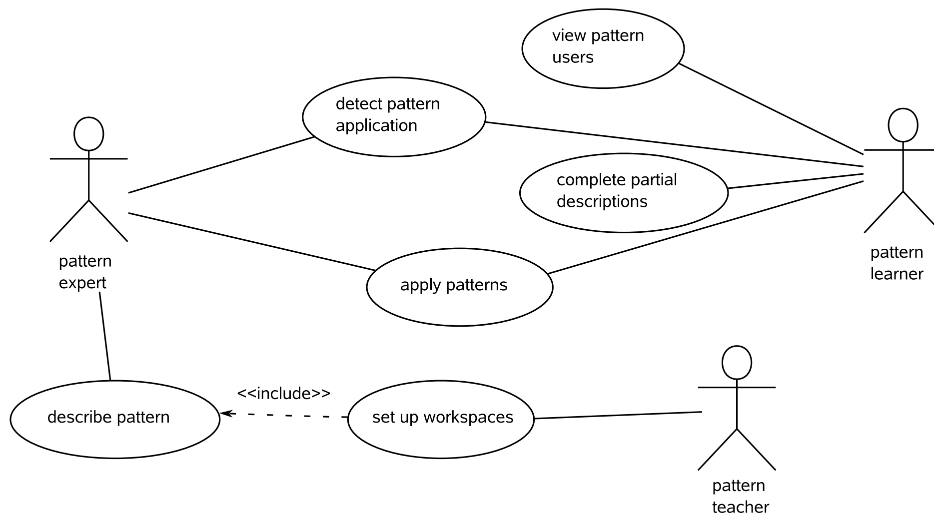

Figure 5. Learning use cases

involves the setup of the general surroundings of the field of study as well as the formulation of particular tasks. When teaching design patterns, the general surroundings are largely described by software engineering in general. The teacher would thus preload the system with a general software engineering model (section 2.3) which forms a mostly constant basis for the work on design patterns. There is thus little personalization to be expected on the software engineering model, but such personalization is of course possible.

Teaching design patterns requires rich descriptions [11]. In section 3.3 asset models for typical design pattern descriptions have been shown. Based on the software engineering model, the teacher can import models which describe patterns. Since these are the field of study, heavy personalization is expected here. The openness of the model allows the teacher to extend these models for teaching purposes. Personalizations made by the teacher can lead to concrete tasks for students, for instance where some parts of a model are deleted in the personalization and the task is to fill the created gaps.

Another possible task created through personalization is a scenario in which the teacher provides a description of a concrete software and it is up to the students to detect the use of patterns by creating personal instances of the appropriate pattern applications. Here only class diagrams are given, the learners detect pattern applications by capturing the classes' roles in (personal) `Pattern` asset instances. More advanced students can be asked to refactor the provided system description through the application of additional patterns where possible.

In both cases it is important to stress that while learners work alone or in small groups they can consult with the other learners of their course or with the teacher as both work in the same system. The system setup that is required to achieve this will be shown in the next section.

System-based learning in general needs support from the system outside the direct field of study: a model of the learners, their advances in the field of study, and a matching with appropriate materials come to mind. Though the above extensions made to the pattern model by the teacher do not automatically lead to a complete model of e-learning

they can be combined with general learning models [23] to form a complete environment of learning.

4.3. *A CCMS for Teaching Patterns*

In order to achieve dynamics, CCMSs are generated from asset models (section 2.3) by a model compiler [24]. On model changes—a personalization step in the cases considered here—a CCMS is dynamically modified to account for the changed model. Each modification preserves existing asset instances and maintains back references to the public model that has been personalized.

Dynamics of CCMSs is further enabled by an architecture that supports evolution [8]. A CCMS consists of a set of cooperating *components*, each hosting assets of one particular model. Components in turn are implemented by *modules* which offer a certain functionality. The components' functionality is provided by modules working in concert.

Two particular kinds of modules used below are *client modules* and *mediation modules*. Client modules access third-party software such that the services are accessible to a CCMS. A typical example is a client module to map asset definitions to a database for asset persistence.

Mediation modules delegate requests to their two base modules according to a definable strategy which implements a particular behavior. For instance, in a personalization scenario existing asset definitions are stored alongside their personalized variants using two different client modules. A mediation module provides retrieval of assets from both modules, creation of personal assets in the personal client module, and personalization of public assets by copying an asset from a public client module to the personal one and modifying the copy.

Through the generative approach a CCMS for the management of pattern descriptions can be generated from the models presented in section 3.3. General requirements for such a CCMS can be deduced from [25]. The ability to serve as a tutoring system is based on the consideration of both content and conceptual models in assets and on the modeling openness.

Hosted content describing software systems can serve as an extensional definition of a design pattern by giving an abstract definition and by showing several applications, counter examples, etc. Conceptual models point out the design patterns that are visible in content, for example, in a complete class diagram used as a case study. The dynamic nature of CCMSs permits active learning processes as discussed in the previous sections. For example, students can improve given designs by introducing patterns. In doing they understand how the respective pattern is applied in practice. Furthermore, personalization allows to change definitions of patterns. This way students can experience which properties patterns have and why they are required.

A design pattern CCMS can be set up as a compound system which includes the CCMS generated for an SE model like the one sketched in section 2.3 as a component. Figure 6 shows an example of such a CCMS generated as a tutoring system. This component can be used independently in SE processes, while pattern description components manage the accompanying information on pattern applications.

In the example of figure 6 an SE component is shown by the *SE Community's Client Module* that accesses some third-party software, here a database management system storing the SE asset definitions.

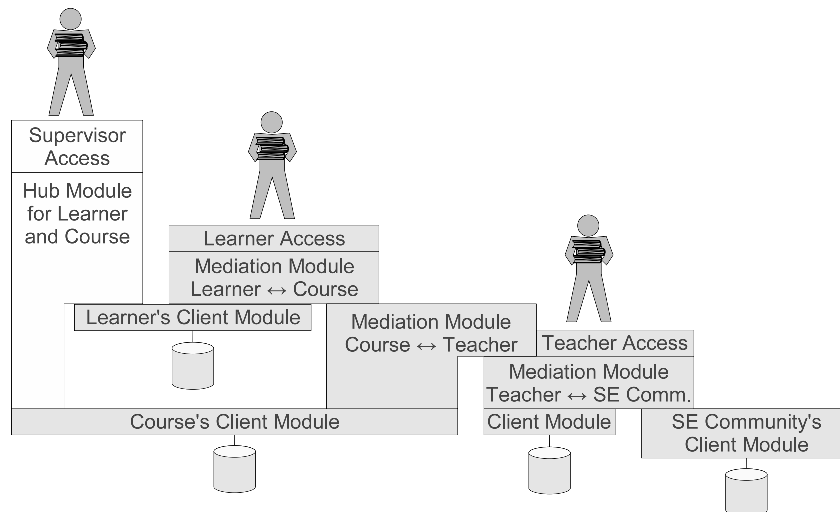

Figure 6. A Sample CCMS for E-learning Patterns

Based on the asset collections found in the SE component a teacher can prepare course material for an SE course using a teacher's component by interaction with the *Teacher Access* module. Here a model like the `Patterns` model from section 3.3 is employed. Since the `Patterns` model imports definitions from the `SoftwareEngineering` model, the components of a teacher and the SE community are connected through a *Mediation Module Teacher* ↔ *SE Community*. This module makes the general SE definitions available to the teacher's component and merges them with the personal asset definitions managed by the client module. Thus it allows to interrelate assets as discussed in section 3.3.

CCM components for courses are set up in a similar fashion as those for teachers. Albeit the purpose differs: for courses the dynamics is used to tailor the materials which a teacher prepared to the needs of a certain course. Still, assets are interrelated through the *Mediation Module Course* ↔ *Teacher* such that a teacher can navigate from an asset presenting some design to a course to the more general asset from which it has been derived.

Participants of a course are equipped with a learner's component accessible via the module *Learner Access*. Such a component is an environment for active learning. In addition to the possibility to access the learning materials prepared for their course, learners are equipped with a repository of their own. This way students can create their own models and create personalized instances from given materials. This use of openness allows learners, for instance, to solve problems and experiment with their solutions.

When students finally deliver the results of their work they simply *publish* their modified asset models. Taking the personalization path backwards the modified models are finally presented to the teacher who can then review them. Constraints are checked on every publication stage so that a student's work will be rejected if it conflicts with the general SE or pattern definitions.

Students usually do not only deliver a final solution. On their way to creating it they also will discuss intermediate results with teachers and fellow students. The discussion

among students can take place in the course component. Students can publish intermediate results to this component—at least if they meet all formal constraints—making them visible to students of the same course. These students can then, for example, annotate the proposed asset definitions.

Teachers, in addition to providing learning materials, take the role of a supervisor while students work on their tasks. This role is supported by an additional component accessible via *Supervisor Access*. This component does not maintain any additional content. Instead it allows a supervisor to take a look at both the materials presented to a course and the personalizations made by all participating students. This way a supervisor can advise students by monitoring their progress while given models can be inspected if required through the access to the course component.

5. Summary and Outlook

We have shown that it is beneficial for teaching purposes to model design patterns dualistically. This helps students to understand patterns by rich medial representation of software design as well as a conceptual models pointing out their particularities. Furthermore schema personalization enables active learning as it allows learners to model the subject under study in the most appropriate way. Similar benefits arise from instance personalization where students are asked to complete partial content.

In future work it will be interesting to extend our conceptual model of design patterns to reach further into SE. It can then not only be used for teaching but will also be applicable to software creation proper. In the area of learning systems improving the reactions of the system to its users seems promising. The goal is to make learning systems react smartly to student's modeling decisions. A promising approach is to try to detect common misconceptions with patterns. This way learners receive more directed feedback and do not need to consult their supervisor or fellow students in oder to understand the basics of patterns.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1994.

[2] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2000.

[3] Hans-Werner Sehring, Sebastian Bossung, and Joachim W. Schmidt. Active Learning By Personalization - Lessons Learnt from Research in Conceptual Content Management. In *Proceedings of the 1st International Conference on Web Information Systems and Technologies*, pages 496–503. INSTICC Press Miami, May 2005.

[4] Sebastian Bossung, Hans-Werner Sehring, Patrick Hupe, and Joachim W. Schmidt. Open and Dynamic Schema Evolution in Content-intensive Web Applications. In *Proceedings of the Second International Conference on Web Information Systems and Technologies*, pages 109–116. INSTICC, INSTICC Press, 2006.

[5] Yann-Gaël Guéhéneuc, Stefan Monnier, and Giuliano Antoniol. Evaluating the Use of Design Patterns during Program Comprehension – Experimental Setting. In *Proceedings of the 1st International Workshop in Design Pattern Theory and Practice*. IEEE Computer Society Press, 2005.

[6] Iris Reinhartz-Berger. Conceptual Modeling of Structure and Behavior with UML – The Top Level Object-Oriented Framework (TLOOF) Approach. In *Proceedings of the International Conference on Conceptual Modeling - ER 2005*, volume 3716 of *LNCS*, pages 1–15, 2005.

[7] Sebastian Bossung, Hans-Werner Sehring, Michael Skusa, and Joachim W. Schmidt. Conceptual Content Management for Software Engineering Processes. In *Advances in Databases and Information Systems: 9th East European Conference, ADBIS 2005*, volume 3631 of *Lecture Notes in Computer Science*, page 309. Springer-Verlag, 2005.

[8] Hans-Werner Sehring and Joachim W. Schmidt. Beyond Databases: An Asset Language for Conceptual Content Management. In *Proc. 8th East European Conference on Advances in Databases and Information Systems*, volume 3255 of *LNCS*, pages 99–112. Springer-Verlag, 2004.

[9] Object Management Group. *Meta Object Facility (MOF) Specification*, 1.4.1 edition, July 2005.

[10] Joachim W. Schmidt and Hans-Werner Sehring. Conceptual Content Modeling and Management: The Rationale of an Asset Language. In *Proceedings of the International Andrei Ershov Memorial Conference, Perspectives of System Informatics, PSI '03*, volume 2890 of *LNCS*, pages 469–493. Springer-Verlag, 2003.

[11] John Vlissides. *Pattern Hatching: Design Patterns Applied*. The Software Pattern Series. Addison Wesley Longman, 1998.

[12] Bernd-Uwe Pagel and Mario Winter. Towards Pattern-Based Tools. In *Proceedings of the European Conference on Pattern Languages of Programming and Computing '96*, 1996.

[13] Gerson Sunye, Alain Le Guennec, and Jean-Marc Jézéquel. Design Patterns Application in UML. In *ECOOP*, pages 44–62, 2000.

[14] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool Support for Object-oriented Patterns. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP '97 - Object-Oriented Programming: 11th EuropeanConference*, volume 1241 of *Lecture Notes in Computer Science*, page 472. Springer, 1997.

[15] Yann-Gaël Guéhéneuc, Houari A. Sahraoui, and Farouk Zaidi. Fingerprinting Design Patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 172–181. IEEE Computer Society, 2004.

[16] Daniel Lucrédio, Alexandre Alvaro, Eduardo Santana de Almeida, and Antonio Francisco do Prado. MVCASE Tool – Working with Design Patterns. In *Proceedings of the Third Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2003)*, 2003.

[17] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In *ASE*, pages 166–173. IEEE Computer Society, 2001.

[18] Tommi Mikkonen. Formalizing Design Patterns. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society.

[19] W.J. Clancy. A Tutorial on Situated Learning. In *Proceedings of the International Conference on Computers and Education*, 1995.

[20] Ulrik Schroeder. Meta-Learning Functionality in eLearning Systems. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, and Medicine on the Internet*, 2002.

[21] Heidrun Allert, Christoph Richter, and Wolfgang Nejdl. Lifelong Learning and Second-order Learning Objects. *British Journal of Educational Technology*, 35(6):701–715, 2004.

[22] Sissel Guttomsen Schär and Helmut Krueger. Learning Technologies with Multimedia. *IEEE Multimedia*, 7(3):40–51, 2000.

[23] M. Derntl and R. Motschnig-Pitrik. Conceptual Modeling of Reusable Learning Scenarios for Person-Centered e-Learning. In *Proceedings of International Workshop for Interactive Computer-Aided Learning (ICL'03)*. Kassel University Press, 2003.

[24] Hans-Werner Sehring, Sebastian Bossung, and Joachim W. Schmidt. Content is Capricious: A Case for Dynamic System Generation. In Yannis Manolopoulos, Jaroslav Pokorný, and Timos Sellis, editors, *Proceedings of Advances on Databases and Information Systems: 10th East European Conference, ADBIS 2006*, volume 4152 of *LNCS*, pages 430–445. Springer-Verlag, 2006.

[25] G. Meszaros and J. Doble. Metapatterns: A Pattern Language for Pattern Writing. In *Thrid Pattern Languages of Programming Conference*, Monticello, USA, 1996.