Pattern-Based Code Generation for Well-Defined Application Domains

Frank Heister, Jan Peter Riegel, Martin Schuetze, Stefan Schulz, Gerhard Zimmermann
Computer Science Department
University of Kaiserslautern, Germany

Abstract
Design patterns are commonly understood as describing approved solutions to problems in a certain context. Although patterns raise the abstraction level of program design, the implementation is still left to the software-designer. This paper describes our approach to generate domain specific application code using patterns. The generator is based on a subset of design patterns which contains more formalized patterns with respect to code generation. This is possible due to the restriction to a sufficiently narrow application domain. The designers work concentrates on modeling the application using an object-oriented method and binding patterns to it. The complete product code will be generated automatically from the designed class model and the patterns bound to it. The sample application domain is building simulation.

1. Introduction

Today, design patterns are a well accepted concept to support the design of large and complex applications. While the primary intention of the patterns, the communication of ‘good design’ (i.e., a proven solution for a recurring problem in a certain context), is well served with catalogs of patterns, support for patterns by existing CASE tools is still poor, making it difficult to fully exploit patterns in the later phases of software development. In our opinion, at least a subset of design patterns (we will explain this restriction below) should be integrated into existing tools and methods as ‘first class citizens’, including the possibility to do correctness checking and code generation from pattern instances. In this paper, we will describe our approach to create a domain specific pattern catalog well suited for this kind of support and to set up a generator which creates application code from the patterns used in a design.

Why are not all design patterns suitable for code generation? Based on our experience, this is primarily because many patterns are too general and are not enough formalized. Pattern descriptions are too general because patterns aim to describe a wide family of solutions for a given problem. Many general patterns like those introduced by [GHJ95] can be used in a wide variety of applications with different architectures, and it will be difficult to find a generator for these patterns which satisfies all needs of all applications. Furthermore, the interface of the patterns to other design elements like other patterns, object classes, finite state machines, etc. is defined informally, and the code templates perhaps included in the patterns are more examples than usable code fragments (which is perfectly ok in order to describe a family of solutions).

Some patterns (e.g., Metapatterns by [Pre95]) are inherently abstract and can probably never be used in conjunction with code generators, but some other could be usable if we were able to transform them into a more concrete form. Of course, this restricts the use of patterns to a smaller range of contexts. The pattern would, after it’s transformation, describe a much smaller number of solutions (hopefully larger than one).

The solution we propose to cope with the problems of generality and formalization is to set up a domain specific pattern catalog: if the application domain is sufficiently narrow, it is possible to define the aspects of a pattern that exactly, that code generation principles can be applied. In a restricted domain, we are either able to modify existing patterns, or we can look for concrete, domain specific patterns right from the start. We can then set up a pattern catalog which is, to a certain extent, formal-

---

1. With existing tools and techniques, patterns can not be used as modeling elements like objects, relations, states, and so on. Even worse, patterns have the tendency to ‘vanish’ from the design documents after they have been applied (compare [Sou95]). By integrating patterns as a modeling concept, the usage of patterns can be documented along with the other software models, and further tool support is at least possible. A first step towards this goal is the integration of patterns into UML [UML97].
ized: interfaces of the patterns in the catalog can be defined in terms of formal parameters with well defined types (like method, relation, object classes, etc.), and the code templates can be extended to perform the complete, domain specific functionality associated with a pattern which can then be transformed to application code by a generator. Please note that we do not advocate for a complete formalization of design patterns in form of a theoretical, provable framework. We just make some extensions to facilitate the use of tools. The resulting catalog, which should in every case contain a ‘system of patterns’ [BMR96], is well suited for syntactical correctness checks and code generation. It will not contain patterns to solve all software problems, but a complete set of solutions for a small, well-defined domain.

The application domain we have chosen for our approach is real time building simulation. The pattern catalog, which is briefly described in chapter 2, is structured into three sections: patterns for domain specific behavior, for simulator control, and for defining the patterns interface to varying object models and environments. The generator (called PSiGene\(^2\)), which is described in chapter 3, uses an ‘implemented’ version of the pattern catalog and features a two-phase code generator for the patterns. We will discuss related work and the results of our approach in chapter 4 and 5.

The Software Development Environment

The implementation of our approach does not work stand-alone, but is integrated into a large, experimental software development environment called MOOSE\(^3\). It features a set of model editors for different notations like class diagrams or pattern bindings. All models are stored in a database and can be retrieved and modified for the development of large families of applications. A set of code generators for different aspects of a software system (like generators for abstract data types, for interactive graphical user interfaces, for data exchange, and other components) interpret the models in the database and create application code. MOOSE has successfully been used for a number of applications in different domains (ECAD, CASE, Control Systems). For further readings on MOOSE we refer to [ARS97].

PSiGene is a part of this development environment and serves as a flexible building simulation component: instead of providing a fixed simulation component, we provide a generator which creates tailored versions of the simulator, according to the applications’ needs. It is important to notice that this generator requires the services of other generators of MOOSE, in particular the service of the abstract data type generator. This ADT-Generator creates a class hierarchy from a class diagram, including all method implementations for object creation and deletion, relation and attribute access methods, and methods for persistent object storage. PSiGene assumes that such a class hierarchy already exists before it starts its work.

2. The Pattern Catalog

There are two general ways of gaining a domain specific pattern catalog: the first one is based on adaption of patterns taken from books or reuse of patterns built during former modeling processes. These patterns are usually well tested because of their high degree of distribution among the pattern community. They are on a high level of review and are frequently used in existing systems.

The second way is the analysis of existing software, finding solutions for typical, recurring problems in the applications domain. This process is often referred to as ‘pattern mining’. This costs more effort than taking existing patterns, because new patterns have to be found and to be extracted from existing programs. On one hand, these new patterns should be of an adequate level of abstraction, describing a number of solutions larger than one. On the other hand, for each new pattern suitable code-templates have to be found to make generation possible. Hence, if no suitable templates exist, a pattern is only usable for modeling but code generation is not possible.

Also, a combination of both ways of gaining the catalog is conceivable.

---

2. PSiGene is an acronym for Pattern-Based Simulator Generator.
3. MOOSE is an acronym for Model-Based Object-Oriented Software Generation Environment
We primarily followed the software analysis approach. Precisely, our patterns are extractions from a simulator prototype which has been developed in our research group and from technical literature on thermodynamics. The prototype has been analyzed exactly. It is build upon a simulator kernel which offers several mechanisms such as scheduling, communication and comfortable multithreading. This kernel is treated as a library but is available on the modeling level through well-defined interfaces.

**Finding Domain Specific Patterns**

Due to the generation of the simulator class hierarchy by MOOSE, and because basic simulator functions are provided by a library, only the domain specific simulation methods (including interface methods to the generated parts and libraries) were subject to analysis. The focus of the analysis process was on behavioral patterns. During this analysis process, we primarily tried to gain information on the interface between a pattern on one side and the elements it interacts with on the other side. Examples for these elements are relations, attributes, classes, and other patterns. Furthermore, we tried to gain generic code templates implementing the behavior of the patterns. Interfaces and code templates are crucial for our approach as they are needed for tool supported modeling of applications and for code generation.

The patterns we found in the analysis can be grouped into three different categories: domain specific patterns, patterns for control and patterns for structural adaption. The first category covers patterns which deal directly with the functionality of building simulators (e.g., physical effects and simulation algorithms). For example, the pattern ThermalMass describes how a heat capacity gets influenced by heat flows. The second category deals with patterns which interface with the simulator kernel library. The last category includes all patterns that are necessary to connect the patterns of the first categories to different simulation class models. These patterns must match the peculiarities of the different models. Some of the extracted, developed and categorized patterns are shown in table 1.

<table>
<thead>
<tr>
<th>domain specific behavior</th>
<th>control</th>
<th>structural adaption</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThermalMass</td>
<td>EventTriggeredComputation</td>
<td>VariableValue</td>
</tr>
<tr>
<td>ThermalJunction</td>
<td>ContinuousComputation</td>
<td>ConstantValue</td>
</tr>
<tr>
<td>ThermalExchange</td>
<td>StateMachine</td>
<td>SingleIndirection</td>
</tr>
<tr>
<td>Actuator</td>
<td></td>
<td>MethodAlias</td>
</tr>
<tr>
<td>Sensor</td>
<td></td>
<td>MethodBranchOnRelation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ComplexIndirectionOnSum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MethodComposition</td>
</tr>
</tbody>
</table>

*Table 1. Classification of the extracted patterns (excerpt)*

As an example for a domain specific pattern an extract of the pattern ThermalMass is shown in figure 1. The organization of our patterns is the same as in [GHJ95]. The sections Intent and Motivation describe the addressed problem using a suitable example from the application domain. Applicability, Participants, and Collaborations show the context and define how a pattern can be used. Structure and Implementation describe the solution. We have formalized some sections of the pattern description further in order to make code generation possible. Especially the Implementation section and the Participants section are important for our generator.

After setting up an initial pattern catalog, it has to be checked if it contains all patterns which are necessary to reassemble a complete simulator. Furthermore, it has to be tested if it is possible to generate code based on these patterns. These two steps were done by a manual test implementation of a building simulator using the patterns of our catalog.

Extracting useful patterns out of an existing application is not a straight-forward process. Rather it is an applied cyclic rollback-strategy with continuous improvements on the patterns which, in turn, can cause updates of the interfaces and, therefore, of other patterns, too. These improvements and updates have been made during the manual generation. The result was a working version of a building simulator as well as a refined set of patterns.
6.1. **Pattern ThermalMass**

**Intent**
The ThermalMass pattern computes the temperature of a mass depending on the amount of heat affecting the mass. …

**Also Known As**
Simulation thermischer Masse

**Motivation**
A volume has to act as a thermal mass to compute its temperature. …

**Applicability**
This pattern can be bound to any thermal mass. Typical this is a room or …

**Structure**

![Diagram of ThermalMass pattern]

**Participants**

**Objects**
- target
  Object to bind pattern to.

**Attributes**
- temperature
  Last computed temperature.
- volume
  Volume of the thermal mass.

**Methods**
- compute
  Does the calculation cycle for one time.

**Interfaces**
- getHeatCapacity
  Determines the storage capacity from the temperature of a mass.
- collectHeatFlows
  Determines the heat-flows from and to a thermal mass.

**Collaborations**
The computation relies on …

**Consequences**
…

**Implementation**

Smalltalk-Code-Templates

- [init]
  ```smalltalk
  self {amountOfHeat} := (self {temperature} * self {volume} * self {getHeatCapacity}).
  self {timeOfLastComputation}:
  Scheduler simSched simMillisecondClockValue.
  ```

- [compute]
  ```smalltalk
  heatCapacity := self {getHeatCapacity}.
  self {calculateTemperature}WithCapacity: heatCapacity
  self {timeOfLastComputation}:= timeNow.
  ```

- [calculateTemperature]WithCapacity: capacity
  ```smalltalk
  withHeatFlows: heatFlows
  while: passedTime self {amountOfHeat}:
  (self {amountOfHeat} + (heatFlows * passedTime)).
  self {temperature}:
  (self {amountOfHeat} / (self {volume} * capacity)).
  ```

**Related Patterns**
Thermal Exchange

---

**Pattern Formalization**

As explained above, we need to formalize some parts of the catalog entries in order to enable tool support. In particular, we need a formalization of the interface between a pattern and other elements and formal code templates suitable for code generation.

The pattern’s interface, i.e. the connection between a pattern and other participating elements, can be described with formal parameters. As shown in the *Participants* section of figure 1, the interface can be divided into several subsections like connections to object classes, relations, attributes, methods, and so on. Furthermore, formal parameters may have additional type constrains (not shown in figure 2, but implemented within tools), for example:

```
volume:attribute (type real, read_only).
```

Regarding the methods we distinguish two kinds: methods for which the complete code is generated are described in the subsection *Methods*, whereas interfaces of methods that are defined elsewhere are placed in the subsection *Interfaces*. This corresponds to template-methods and hook-methods as described in [Pre95].
We require that the code templates contain valid code fragments in an object oriented programming language (in our case Smalltalk). The templates are generic, and all parts which need to be modified for the use of a pattern are put in braces (macros, see chapter 3). An example is shown in the Implementation section of figure 1. A macro usually corresponds to a formal parameter and will be replaced during code generation.

Therefore, formal parameters of a pattern have to be bound to actual values in order to instantiate the pattern and use it with the generator. For that reason we have defined a language which can express pattern bindings. A typical expression written in this language looks as follows:

```
<patternname> bind: '<formalParameter1>' to: '<actualValue1>';
...
bind: '<formalParameterN>' to: '<actualValueN>'.
```

This expression has the effect that all actual values are assigned to the formal parameters in the pattern.

Moreover, the code templates may be broken into several code fragments. A template will be constructed out of these fragments during the generation process. Hereby, the construction of code templates depending on the binding of formal parameters is possible (an example is given in chapter 3).

Usually, objects participate in several patterns. Interactions between patterns are coordinated through the above described binding mechanism; for example, using the same attribute name or matching method interfaces with implementations of that method provided by another pattern.

**Modeling Example**

Figure 2a shows a brief example: in order to simulate a room thermally a pattern ThermalMass has to be bound to Room. This pattern cooperates with a second pattern, ContinuousComputation, to achieve real-time behavior. To fulfill its task, the pattern has to access several attributes and relations of the class Room (e.g., temperature and volume). The required binding can be found in figure 2b. These few expressions tell the generator what to do during the process of generation. Line 1 states that the code will be generated into the class Room, we say: ‘ThermalMass is bound to Room’. As in the abbreviated example of the pattern ThermalMass in figure 1 can be seen, the next two formal parameters, ‘temperature’ and ‘volume’, will be replaced during code generation.
ture’ and ‘volume’, have to be matched to attribute names defined in Room. In this case these attributes are carrying the same name as the formal parameters. ‘compute’ denotes the name of the method which is to be injected into the class Room. The remaining lines specify the formal parameters of the interface to other patterns.

This example explains how the mechanism of binding works. In analogy to this example all other patterns get bound to the class model in order to build a complete building simulator.

3. Pattern-Based Simulator Generator

Having a catalog of formalized, concrete patterns we can do a further step by implementing these patterns. The Participants and the Implementation parts of a pattern description contain all needed information about the code to be generated. Within PSiGene, each pattern of the catalog is modeled as a Smalltalk class. In the following, we will refer to such classes as pattern classes. Unlike other approaches (e.g., [Sou95]) these pattern classes are not implementations of the patterns functionality, but act as ‘micro generators’. Each of the pattern classes is derived from a common superclass AbstractPattern. This class contains methods supplying the needed protocol for binding patterns, for creating warning and error reports, and for generating and integrating code into a given class model. Hence, each single pattern class inherits the ability for implementing the code realizing the functionality it describes. In consequence, the community of the pattern classes describing the patterns of our catalog build the intrinsic generator.

It was found that some patterns implement variations of an abstract problem. Such variations usually differ in the algorithms used for solving the problem but coincide in their structure. Accordingly, we added abstract classes which only contain the essence needed for generating code. In other words, they define how to implement the code realizing a patterns functionality. For distinction, we call these abstract classes EssencePatterns and the concrete classes ModelPatterns. This way, we gain a hierarchy of pattern classes partially shown in figure 3.

Moreover, figure 3 shows another principle used for building large pattern classes: aggregation is used to utilize ModelPatterns which implement portions of the functionality of an aggregating pattern class. More specifically, aggregating pattern classes bind additional ModelPatterns to the class model which are invisible for the user of such patterns. This results in reuse of knowledge and code previously implemented by other pattern classes.

Applicating PSiGene requires the simulator class model already been implemented. At the same time, this class model is input to the generator. Further input is a list of bindings containing an entry for each pattern to be bound to the class model as explained in the previous chapter. Generally, participants get bound to names representing objects, methods, or relations. These are defined by the class model or will be generated by this ModelPattern or other pattern classes.
Code Generation

The process of implementing patterns into the given class model takes place in two phases: the binding phase and the generation phase.

In the binding phase, for every entry in the list of bindings the according pattern class is instantiated and its participants get assigned to the specified values. On instantiation, not only the visible pattern classes but also aggregated ones are instantiated. During this phase, no code is generated. Each of the created instances is checked for completeness, based on the formal parameters defined by each pattern. Executing the generation phase depends on the global result of the complete set of check results. Additionally, a report including the results of these completeness checks is produced for the user of PSi-Gene.

In the generation phase, every instance built in the binding phase is asked to generate code for the class it is bound to. Every pattern class does not only know how to implement its portion of code, but (also knowing its concrete target classes) it carries out actions for optimizing the generated code, controlled by a generation method within each pattern. For example, loops over relations of cardinality n get eliminated if the bound relation only is of cardinality 1.

![Figure 4. Simple example for optimized code generation, (a) building blocks and resulting code template for relation (b) of cardinality n and (c) of cardinality 1](image)

The code realizing a pattern’s functionality resides in each pattern class in form of code fragments. Such code fragments are used as building blocks for constructing a code template which is optimized for the models structure. A template contains macros which are replaced right before implementing the code. The macros inside the building blocks are placed in braces. These macros can either be replaced by other building blocks or formal parameters. An example for optimized and blockwise replacement is shown in figure 4: the resulting code template for generation of a bound pattern ComplexIndirectionOnSum gets constructed out of four building blocks. Depending on the relation’s cardinality, either a loop is generated (fig. 4b) or not (fig. 4c). A generation method for the above template looks as follows:
generateCode
| macroText target relation |

① target := Smalltalk at: (self elementAtMacro: 'target').
② relation := self elementAtMacro: 'relation'.

“Construct Code-Template”
macroText := self body.
③ (target perform: relation) isKindOf: RelationN “check for cardinality n”
  ifTrue: [
    macroText replaceMacro: ‘loopOrAdd’ with: (self loop).
    macroText replaceMacro: ‘connections’ with: (self connection).
    macroText replaceMacro: ‘add’ with: (self add).
    macroText replaceMacro: ‘object’ with: ‘anObject’. ]
  ifFalse: [
    macroText replaceMacro: ‘loopOrAdd’ with: (self add).
    macroText replaceMacro: ‘object’ with: (self connection). ].

“Inject Code into target with previous replaced macros.”
④ macroText replaceMacrosUsing: self macrosAndElements.
⑤ self putMethod: (self elementAtMacro: ‘computation’) withSource: macroText to: target.

A simplified example for pattern binding and code generation is shown in figure 5: the class *Room* needs a new method *totalQ*. It is meant to compute the total heat flow of all *Radiators* connected via the relation *radiators*. The pattern ComplexIndirectionOnSum serves as solution to this problem. The relation *radiators* is of cardinality n, so the adequate template (fig. 4b) will be generated and used for further processing. Using this code template and the binding of ComplexIndirectionOnSum to *Room* as target, a method will be injected into *Room* like described in figure 5. This injected method is the result of the above presented method *generateCode*: according to the actual bindings, target at ① resolves to the class *Room* and relation at ② to #radiators. At ③, the relation *radiators* is checked for its cardinality and according to the result the corresponding code template will be build. The replacement of all macros left in the template happens at ④. Here, *self macrosAndElements* returns the list of associations between participants and model elements. Every macro will be replaced by its assigned value. Finally, at ⑤ the constructed method will be injected into the target class *Room* using the assigned name *totalQ*.

Of course, this example is very simple. More complex patterns claim more than just macro replacement. For example, the StateMachine pattern has to generate new classes as states are defined with the binding.

Moreover, the presented example shows the two levels of design being used: the macro level and the pattern level. The macro level is fine grained doing low level code generation. It builds and implements optimized code out of the class model and the bindings both given as input to PSI-Gene. In contrast, the pattern level is coarse grained and takes place in the design phase of software development. The user knows the problem a pattern solves and the structure of the class model. Binding a complex pattern to the class model results in adding big portions of code. In other words, the user binds solutions to the class model.

4. Related Works

Our approach is related to the area of application generators and pattern-based design strategies. It shares with the first (e.g., UI builder, simulation systems) that it is optimized for one domain. In divergence to most application generators, we do not use domain specific modeling languages. The users of our technique are meant to be experts in object-oriented design and not in simulation methods.

With code generation we share the basic principles of powerful code generators like GenVoca [BST94]: subsystem building blocks, standardized interfaces and additional parameters. Our subsystem building blocks are domain specific patterns. They include a description of their interfaces in the *Participants* and *Collaborations* section of each pattern. Parametrization takes place by binding the pat-
terns to a class model. The GenVoca generators concentrate on program transformation, enhancing a target language with powerful program constructs. This is contrary to our approach where the generators’ functionality is encapsulated in pattern classes.

There are some approaches working with code generation from patterns. For example, Budinsky et al. [BFV96] use design patterns from the “gang of four” (Gamma et al. [GHJ95]) to instantiate code templates, but their pattern implementation is not as interwoven with the overall system design as our approach, so that the resulting code fragments have to be manually adapted to the software system. By restricting the domain in which applications will be modeled, we are able to capture domain specific knowledge (i.e. special algorithms or structures) into patterns. There are various pattern languages for different domains (e.g., [Smi95]). Restricted domains allow patterns to be concrete enough that automatic code generation is possible.

An approach using design patterns for the same purpose as we do is the Demeter Method [Lie96]. There, the class design and the algorithms are kept separate. Design patterns are used for the implementation of the algorithms. These usually work on several of the defined classes, and only those are mentioned which are important for implementing the algorithm. Instead of using code generation, mapping of classes and relations used in design patterns to the class structure is performed at runtime.

Figure 5. Simplified pattern binding and code generation example
5. Conclusion

We have developed a domain specific, pattern-based modeling and software generation approach. The goal is to support the modeling of building simulators by using domain-specific simulation patterns. These patterns have a formal interface and can be bound to a class model describing the simulator’s structure. With these pattern bindings together with the class model, complete product code for a building simulator can be generated.

By generating complete product code, the bindings of our patterns and the design decisions they document do not get lost during the implementation phase (compare [Sou95]). Additionally, comments in the generated code show which pattern was responsible for the inspected code fragment (making manual adaptions, if needed, more simple). But normally the code does not have to be inspected or altered at all.

In respect to design patterns, we have to admit that the distinction between the patterns themselves and the design templates serving one possible solution of a patterns problem is not yet completely given. Focusing to the concept at first point, we simply took one template for each pattern. By now, different algorithms solving the same problem are realized as patterns with similar names. A further improvement might be letting the user choose one of several design templates solving an applied design problem.

In addition, interaction between patterns happens on a quite low level (i.e., close to the implementation level) which may cause some problems with respect to finding the matching names of attributes and methods.

Many different building simulators can be modeled using our method. This flexibility is due to the fact, that on one hand, the underlying class model can be developed (to a certain degree) independent of the patterns, and on the other hand, different patterns providing a similar functionality can be applied. Especially adaptions of existing simulators or adaptions to given class models are simple. Adaptions can be: including new effects to be simulated, changing the accuracy of a simulated effect, changing the structure of the building, using a different simulation method (real-time, accelerated real-time, or time-warp) or changing the computation intervals. While our pattern catalog is domain specific, we believe that the overall methodology can be applied to other domains as well (using different patterns): we are currently extending our approach to the domain of reactive systems and experimenting with patterns for graphical, interactive user interfaces. The transition to new domains is relatively straight-forward (provided that one finds the required patterns!) as long as the environment (the support by MOOSE, the model the patterns are bound to, etc.) does not change. Other domains may require significant reimplementation efforts, but the main principle should be applicable, anyway.

We are now implementing a graphical pattern binding editor to be able to describe pattern bindings more easily. This editor is able to perform checks if all patterns are bound correctly to the class model and can give hints on how to model certain structures (e.g., if one pattern is bound to the model it can propose to bind other patterns also). Furthermore, we are looking for more simulation patterns capturing new physical effects and for different algorithms describing the same effect (to be able to gain more flexibility in modeling a simulator, e.g., trading speed for accuracy). So far, our experiences show that automatic code generation from specialized patterns is possible and that they aid in the development of domain specific applications.
References


[GHJ95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns, Addison-Wesley, 1995


[Pre95] Pree, W., Design Patterns for Object-Oriented Software Development, ACM Press, Addison-Wesley, 1995

[Smi95] Schmidt, D. C., Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software, Communications of the ACM Special issue on Object-Oriented Experiences, Vol 38, No 10, 1995
