

Patterns in action

Level: Introductory

Jochen Krebs, Senior Software Engineering Specialist, IBM

15 May 2006

from The Rational Edge: For software developers unfamiliar with patterns and their various relationships, this article provides an overview of pattern usage in the context of an actual software development project.

Object-oriented software engineering (OOSE) without design patterns is like cooking without a recipe. Patterns guide us with ingredients and step-by-step instructions for assembling the solution to a recurring problem. In the same way we rely on recipes in cooking, we experience patterns as repeatable, proven solutions, and software engineering becomes more reliable and successful.

As in the culinary arts, where chopping and cutting techniques are prerequisites for mixing and flavoring dishes, there are many design patterns for all sort of challenges -- basic, intermediate, and advanced -- depending on your needs. However, food recipes often contain references to other recipes that go well with the main dish, thus enhancing the entire meal.

This article will focus on exactly these pattern relationships, combinations, and variations. It's all part of an emerging trend we might call "pattern-driven software engineering." The examples I provide are visualized in UML and would eventually be transformed into code (e.g., Java). Because patterns do not only affect the structure and dynamics of classes and objects, this article will conclude investigating the role of patterns in a service-oriented architecture (SOA).



The concept of patterns

Patterns emerge as software engineers begin to notice recurring problems. If you design software and you face a situation in which you ask yourself "Gee, I can't be the first person facing this problem!" your search for a pattern has just begun. Once you find and apply a pattern, your solution will not only benefit from the knowledge gained in the past, but this pattern might also open a door to related patterns. An individual pattern works in its described context and offers a variety of related patterns that can improve the quality of your solution even more. Eventually, one design, pattern could be a starting point for an entire pattern-driven design process.

Before we discuss the relationships among patterns, let's explore that culinary metaphor a bit and take a look at some individual patterns.

I'll describe a typical TV cooking show to help explain software patterns and their relationships. The goal of the show is to demonstrate the preparation of a specific meal. On most cooking shows, however, we find cups and bowls in front of the chef, with ingredients such as onions already prepared. That's because the expert cook doesn't need to illustrate the chopping of onions in front of the TV audience; it would be boring. Prior to the taping of the show, the chef has probably asked his subordinates for some quantity of "finely chopped onions," the same ingredients used in many recipes. What's important here is that the chef does not need to communicate the actual cutting technique, but simply asks for the well-known result: a standard cup of chopped onions.

Software engineers make use of such basic patterns, too. Some of these patterns, such as the General Responsibility Assignment Software Patterns (GRASP),¹ are so fundamental that many other patterns make use of them. Basic design patterns organize and control communication or creation, or they establish visibilities among objects. Basically, in an object-oriented system, objects communicate with each other through messages. Therefore all these messages (a.k.a. responsibilities) need to be assigned by the software engineer to build a flexible and maintainable system. Based on

that fact, object-oriented software engineers constantly ask themselves the same basic question: "Who should talk to whom?".

The problem scenario

For the remainder of this article, I will illustrate various approaches to pattern usage through the scenario of a change request to a timesheet application -- the change has to do with the timesheet approval process. Figure 1 shows a typical situation for an object-oriented designer, where a specific business rule requires identifying whether the timesheet is approved or not. The question ("Are you approved?") and the answer ("yes" or "no") are determined, but the questions remain: who should receive and who should send the message?

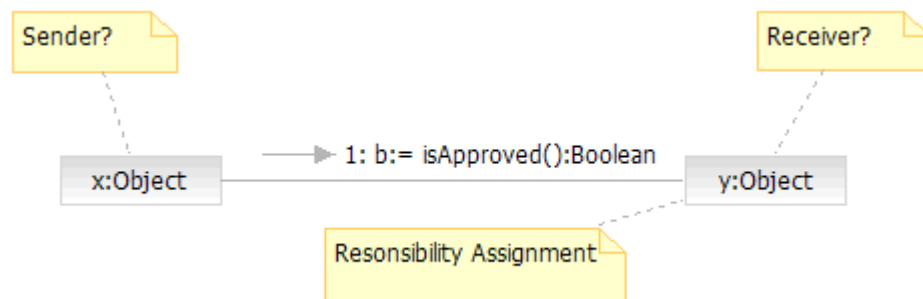


Figure 1: Responsibility assignment

Even for very basic design situations like the one described in Figure 1, we can make use of fundamental design patterns; for example, asking the GRASP patterns for help.

In the TV cooking show, the chef is using a fundamental pattern -- chopped onions -- to assemble a more complex pattern of his own, the meal itself. The level of the pattern has been elevated from a single set of techniques to a dish that comprises other fundamental techniques. The recipe has a name; for example, tomato sauce. It is the chef's responsibility to decide how many onions he uses and how he prepares them. The problem now moves to a higher level, from chopping onions to making a good tomato sauce. The chef begins applying his own pattern, the recipe, which contains other patterns (for sauteeing, chopping parsley, etc.). The experienced chef applies a pattern, in a sense, as a way to present food nicely, focusing on color, texture, and style.

Software design patterns are not different. In addition to the fundamental GRASP patterns, engineers make use of more elevated patterns, such as Gang of Four² (GoF) or architectural patterns. Now that most software engineers graduating from universities are grounded in OO principles, the software development industry has begun to raise the level of pattern adoption from the level of *problem-solving* techniques to *problem-prevention* techniques. I will use the *Design Patterns -- Resusable Objects* (from the Gang of Four) as a design pattern catalog to demonstrate the pattern relationships and use the the IBM Rational Software Architect (RSA) pattern catalog to illustrate the examples.

Let's get back to our initial scenario illustrated in Figure 1, where we plan to build a timesheet application with a focus on an approval process. The designer needs to identify whether a timesheet is approved or not. In this case, it seems almost enough to simply add an attribute called *isApproved* to the *Timesheet* object, which contains one of the boolean values, *true* or *false*. The problem with this solution is, however, that the attributes of the object can change, and depending on the content of the attribute we would need to determine the type of message that will be fired. If we want to add another option -- for example, *Submitted* -- the boolean attribute, which allows two possible values *true* or *false*, does not accommodate this design approach anymore. With the introduction of the *Submitted* state the original design (built for two values) would break and the entire business logic would require us to reevaluate our initial design. Later I will demonstrate how smooth the transition can be from a two-states to three-states design when patterns are applied. As illustrated in Figure 2, our new design approach would violate two fundamental design patterns, Expert and Polymorphism,³ and would unnecessarily couple one object with the business logic that belongs to another object. The boolean value approach would not only violate fundamental design patterns, it would also increase the maintenance burden for software engineers because the design for the *Timesheet* object could easily break and the entire object would need to be re-tested with every change.

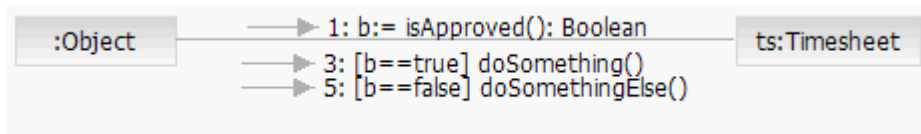


Figure 2: A UML example of violating the Expert and Polymorphism patterns

Translating the UML design from Figure 2 would generate a Java structure like the one shown in Figure 3.

```

.....
    if (b == true)
    {
        ts.doSomething();
    }
    if (b == false)
    {
        ts.doSomethingElse();
    }
.....
  
```

Figure 3: A Java example, violating Expert and Polymorphism

One solution: The State pattern

The GoF pattern catalog offers a possible solution for our design challenge. The pattern is called *State*.

First we verify that the pattern meets our needs and read the intent, application, and consequences sections of the pattern. Because the pattern says that it "Allows an object to alter its behavior when its internal state changes. The object will appear to change its class [GoF]," we go ahead and apply this pattern to our problem.

One of the benefits of applying the *State* pattern is that it can resolve the if-statement situation difficulty shown in Figure 3 by isolating the various states. The UML state-machine notation helps us depict and investigate the various states. Initially our timesheet was fairly simple and we isolated two states out of our existing structure, *Approved* and *NotApproved*.

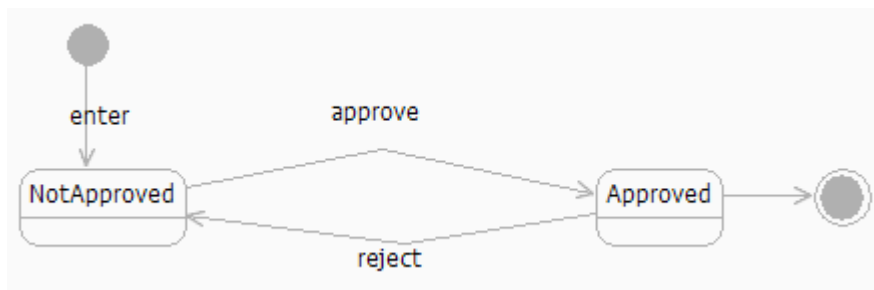


Figure 4: UML state-machine diagram for timesheet (two states)

Instead of asking the object which value is nested in an attribute (in our case *isApproved*) and make a decision based on that (which violate the principle of polymorphism) we instead tell the object what to do and simply send the message to it and let the *Timesheet* object deal with the event. What we would like to design is some way to send a message, as shown below, where *ts* is a *Timesheet* object.

```

.....
ts.approve();
.....|
  
```

Figure 5: New responsibility assignment for timesheet (Java)

After we isolate the various states, remove the if-construct from the *Timesheet* object, and assign the three responsibilities (enter, approve and reject), we then want to apply the *State* pattern to our solution. Using the RSA pattern explorer we navigate to the *State* pattern, which shows us the participating classes in the pattern.

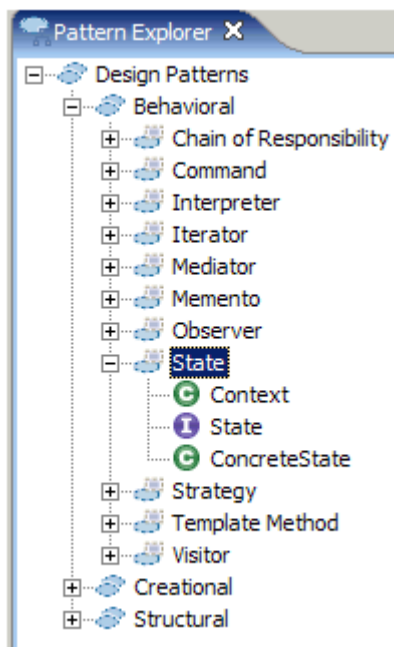


Figure 6: State pattern and participating classes within the RSA pattern explorer

In order to get an overview of the structure of the *State* pattern, the pattern explorer provides us the following layout:

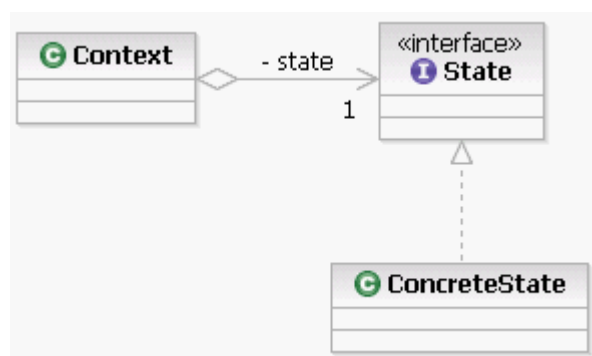


Figure 7: State pattern structure within the RSA pattern explorer

The cookie-cutter solution for the *State* pattern needs to be adjusted to accommodate our application's specific needs. After dragging the pattern from the pattern explorer directly into our workspace, we can assign the participating classes from our application-specific class model. The following diagram contains now the *Timesheet* as a context object, the Java interface *ITimesheetState* for the State and both concrete states from our timesheet application (*Approved* and *NotApproved*).

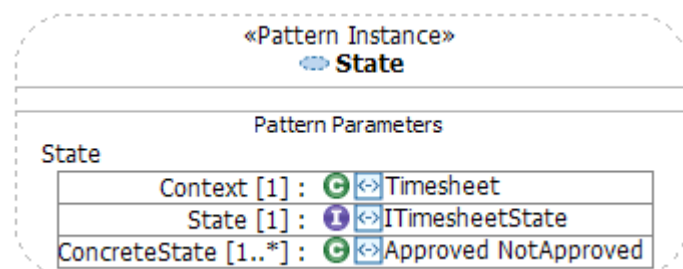


Figure 8: Applied state pattern in RSA

The dynamics of this pattern are shown in Figure 9, using Java. After the message *approve()* has been sent to the *Timesheet* object, it takes the message and delegates it to its state and provides a pointer back to itself (the *this*-parameter).

```
...
state.approve(this);
...
```

Figure 9: Message delegation from the context to the State object

After the message *approve(this)* has been sent, the state which at runtime is located in the *State* object will handle the event (which is truly polymorphic). For example the *NotApproved* state would implement the *approve(ITimesheetState state)* message like this:

```
public void approve(ITimesheetState state)
{
    ITimesheetState newState = new ApprovedState();
    state.setState(newState);
}
```

Figure 10: Concrete state method implementation -- Not Approved

To support the polymorphic approach, we need to assign the approve responsibility also to the *Approved* state, even though we will not do anything in this particular situation.

```
public void approve(ITimesheetState state)
{
    // do nothing
}
```

Figure 11: Concrete state method implementation -- Approved

Now that the *State* pattern[GoF] has been applied, let's see what happens in our one-pattern design if a requirements change occurs: for example, stakeholders need to be able to submit their timesheet after the time has been entered and request approval. The following state machine diagram shows two new states, *Entered* and *Submitted*, which replaced the previous state *NotApproved* to accommodate this requirement change.

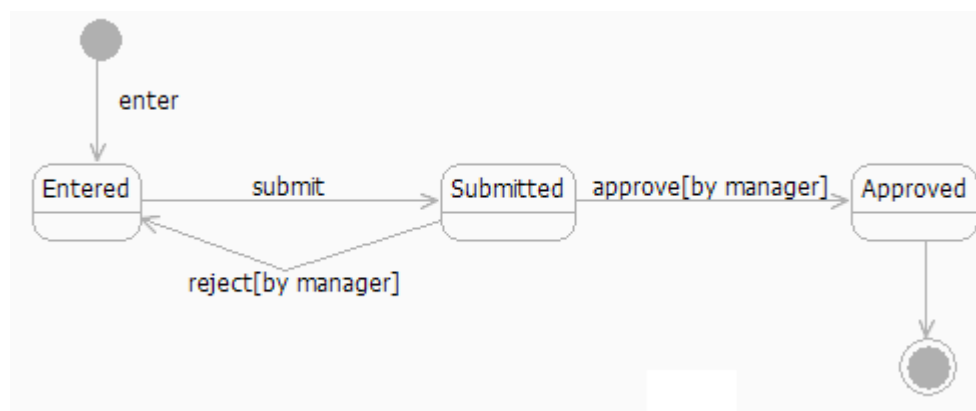


Figure 12: UML state machine for timesheet (more states)

The UML Design Class diagram in Figure 13 depicts the changes caused by the new requirement to the class model. Even though new state classes and messages have been introduced and one state has been removed, the changes are still very manageable. The most important point to be made is that the *Timesheet* has not been changed at all. It still keeps passing all the messages it receives to its actual state. That is a tremendous improvement to our if-else construct from Figure 3, because the area of concern from a testing perspective has shifted away from the *Timesheet* object to its states.

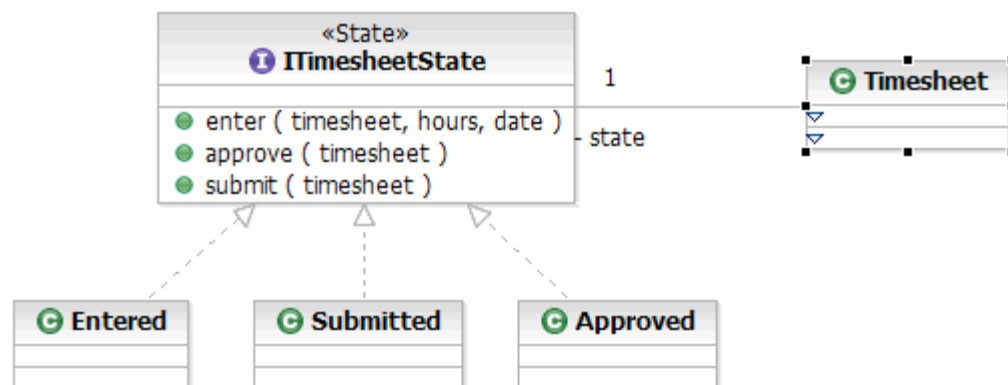


Figure 13: Partial UML design class diagram (timesheet and new states)

Pattern-driven development

In the previous section I illustrated a design problem, applied a common solution (the *State* pattern) to it, and pointed out the advantages of the design using the pattern (maintainability and flexibility). In a pattern-driven solution, a designer will not only apply a pattern when a problem occurs, but will drive the entire design by using patterns. This approach is slightly different, because it assumes that the designer works actively with design pattern catalogs and uses the relationships between those patterns. The patterns within a catalog are usually grouped according to a chosen template. The GoF pattern template, for example, has *Name and Classification*, *Intent*, *Also Known As*, *Motivation*, *Applicability*, *Structure*, *Participants*, *Collaborations*, *Consequences*, *Implementation*, *Sample Code*, *Known Uses* and, last but not least, *Related Patterns*.

The *Related* patterns section within the pattern template contains important clues to other patterns that might be applicable in the context of the object. For example, according to the pattern catalog, the *State* pattern is often related to *Flyweight* and *Singleton*[GoF]. This information must trigger a new set of questions to the software engineer -- for example, "Are there any parts of my solution which could also benefit from the use of the *Flyweight* or the *Singleton* pattern?" -- and cause the engineer to examine the existing approach.

Our solution using the *State* pattern currently has one drawback. If our timesheet system will handle, say, 5,000 timesheets in the approved state, we would also carry 5,000 instances of the *Approved* state. Also, with every state change, we would create a new instance of a new state and the Java garbage collector would need to collect the old state objects. This might not be very critical for our timesheet application, but in other scenarios this could be very expensive in terms of resources. Figure 14 shows just a small number of timesheets and associated states that would multiply thousands of times in our application, as it is thus far designed.

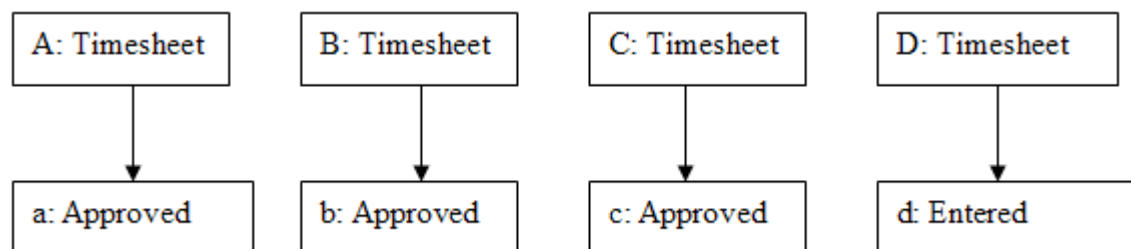


Figure 14: Object model prior to Flyweight pattern

In our timesheet example, the states "approved, submitted, and entered" are good candidates for *Flyweight* objects because there is no need to add any additional attributes to one of the states to distinguish these instances. With the *Flyweight* pattern, we are actually able to improve the *State* pattern solution even further, as shown in Figure 15.

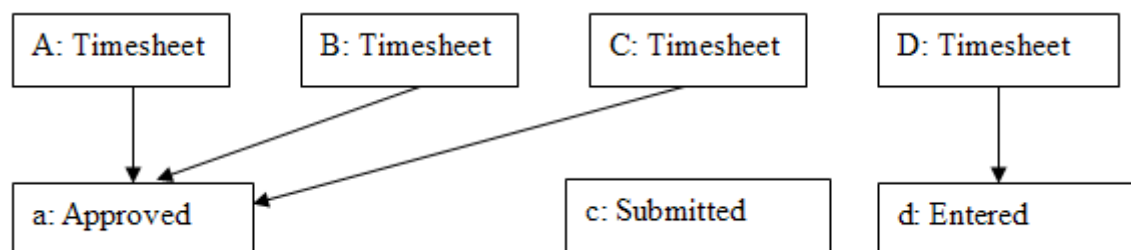


Figure 15: Object model after applying the Flyweight pattern

Our timesheet application now contains only three different state instances at any given moment, which increases maintainability and performance.

But... applying the *Flyweight* [GoF] pattern raises a new challenge for our designer. Instead of creating a new instance of a particular state object or carrying the flyweight objects around as parameters, we would like to achieve visibility to this one instance of state in that particular situation. The *Singleton* [GoF] pattern serves exactly this purpose. We could either implement each state as a *Singleton*, or create a factory of states which creates and manages the states. The latter method would increase maintainability even more through separation of concerns.

In pattern-driven development, the destination may be the origin for new patterns. For example, the *State* pattern often harmonizes with the *Flyweight* and *Singleton* patterns. However, *Flyweight* and *Singleton* have further associations to even more patterns, and so on. For simplicity, I have limited my illustration of these pattern relationships to the GoF catalog; the *Also Known As* section of the pattern template opens the door to other catalogs as well.

Pattern-driven development in Service-Oriented Architectures

In service-oriented architectures (SOA), there are services, providers, and consumers, just as there are responsibilities, receivers, and senders in object-oriented architectures. The difference is that the application-centric design approach, (e.g. the timesheet application) is elevated to the orchestration of services aligning the IT system with, for example, the business processes. So far, our design for the timesheet application has been driven by technology and a requirements change. To include the timesheet application in a SOA, services must be exposed so that human and non-human interfaces can be consumed.

In SOA design, the isolated view of the timesheet application would be replaced by a view of how the services of the timesheet application harmonize with other services from other applications. Depending on business requirements, the timesheet application could be seen in light of an organizational payroll process or in combination with a cost-breakdown-structure for product management. Exposing or modifying existing services or creating new services becomes a critical task for the enterprise and application architect. A more flexible and maintainable system built with patterns is therefore a critical element for a successful SOA.

Earlier in this article I showed how patterns promote the application of additional related patterns, even across catalogs. We also saw that patterns exist in various forms, from helping with design decision's on an object level to patterns on an application level (i.e., assigning responsibilities vs. GoF patterns). The pattern-driven approach decisions helps in providing flexible and maintainable services in an SOA, and the SOA itself can drive and stimulate the pattern-driven approach. Business modeling patterns as well as architectural/network patterns motivated through an SOA can drive pattern-driven development top-down, whereas the pattern-driven application design prepares a successful SOA bottom-up.

Conclusions

Finding a matching pattern for a problem not only presents a solution to a problem, but also means in many cases the beginning of a new search and further evaluation of related patterns. This type of search/discovery/exploration activity should be familiar to you if you've ever used an Internet search engine to explore a topic you only vaguely understand. You often start with terms you're not sure about, but as you see more accepted terms and areas of knowledge unfold in your result set(s), you gain insight into the "patterns" of thinking and solutions that exist. Soon, you are able to enhance your own queries, eventually expanding your original lines of thought.

When properly documented and cataloged, patterns provide a common roadmap, encourage engineers to investigate the problem space, and, more importantly, allow us to apply a set of proven solutions rather than only one particular solution. I have used a basic enhancement request to illustrate the impact of change to a pattern-driven solution for a system in maintenance mode. Following an iterative-incremental process model, projects face very similar situations during construction, and we can easily map the benefits of pattern-driven development to incremental improvements in the project.

The Rational Software Architect (RSA) provides capabilities to support a pattern-driven engineering process by starting with common design patterns (for example GoF), or by creating its own pattern catalog. Publishing a pattern catalog with RSA and sharing the library of patterns on an enterprise level increases adoption rate which results in more reliable and flexible IT design. The ability to react to and adapt to organizational change is a fundamental strategy for a SOA.

References

Erich Gamma, Ralph Johnson, Richard Helm, John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley: 1995.

Craig Larman, *Applying UML and Patterns -- An Introduction to Object-Oriented Analysis and Design and Iterative Development* 3rd Edition, Prentice Hall: 2004.

Christopher Alexander, *A Timeless Way of Building*, Oxford University Press: 1979.

James W. Cooper, *Java Design Patterns -- A Tutorial*, Addison-Wesley: 2001.

Alpert, Brown, Woolf, *The Design Patterns -- Smalltalk Companion*, Addison-Wesley: 1998.

Notes

¹ See Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition), Prentice Hall: 2002.

² See Erich Gamma, Ralph Johnson, Richard Helm, John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional: 1995. These four authors are commonly known as "The Gang of Four," and thus the patterns described in this book are referred to by the same short-hand term.

³ According to Larman, op. cit.

About the author



Jochen (Joe) Krebs (<http://www.jochenkrebs.com>) is a Senior IT Specialist for the Rational Software Brand within the IBM Software Group. He is responsible for successful enablement of Rational products and services for clients in the financial sector. Prior to joining IBM Rational he worked as an Instructor and Senior Consultant with a focus on project management, requirements management, software engineering processes and object-oriented technologies using Smalltalk and Java. He holds his MSc in Computing for Commerce and Industry at the Open University.

