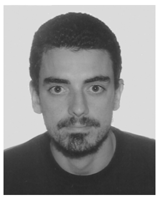
Introduction to patterns

PID_00273621

Jordi Pradel i Miquel
José Antonio Raya Martos

Universitat Oberta de Catalunya



Jordi Pradel i Miquel

Computer Engineer by the Universitat Politècnica de Catalunya (Polytechnic University of Catalonia, UPC). Founding member and Software Engineer at Agilogy. He is also a Software Engineering professor in the Languages and Computer Systems Department at the UPC and Computer Science and Multimedia Studies course instructor at the Universitat Oberta de Catalunya.

José Antonio Raya Martos

Computer Engineer by the Universitat Politècnica de Catalunya (Polytechnic University of Catalonia, UPC). Founding member and Software Engineer at Agilogy. Associate Professor in the Languages and Computer Systems Department at the UPC. Computer Science and Multimedia Studies course instructor at the Universitat Oberta de Catalunya.

The assignment and creation of this UOC Learning Resource have been coordinated by the lecturers: Elena Planas Hortal, Lola Burgueño Caballero

First edition: September 2020
© of this edition, Fundació Universitat Oberta de Catalunya (FUOC)
Av. Tibidabo, 39-43, 08035 Barcelona
Authorship: Jordi Pradel i Miquel, José Antonio Raya Martos
Production: FUOC
All rights are reserved

Contents

Introduction

Patterns are an increasingly popular tool in software development, as they allow us to apply the benefits of reuse (widely known in the case of code) to all development areas (especially analysis and design). Patterns offer proven and documented solutions that can help us refine the artifacts resulting from the different stages of the development life cycle.

This module introduces the student to the world of patterns applied to the different stages of the development life cycle. First, we will see what a pattern is, we will provide a formal description of it, we will study its structure and we will see the advantages and disadvantages of the application of patterns to software development through a small example. Then, we will make a classification of the patterns according to the stage of the development life cycle in which they are used. We will see some examples for each type of pattern.

Frameworks are also a very important tool in software development that, like patterns, help us increase software quality and, at the same time, reduce development time. In this module, we will talk about frameworks, their relationship with patterns and how they can help us apply patterns to software development.

Note that in this module the UML modeling language is used for the representation of both analysis and design diagrams.

Objectives

The main objective of this module is to introduce the concept of pattern and to understand its application throughout the life cycle of software development. Therefore, the objectives that students must have achieved by the end of this module are to:

1. Understand the concept of pattern and to know how a pattern is documented.

2. Understand the main advantages of using patterns.

3. Learn how to apply patterns throughout the development life cycle.

4. Know a classification of patterns according to the stage of the life cycle in which they can be applied.

5. Understand the concept of framework and its relationship with patterns.

1. Concept of pattern

1.1. Definition of pattern

According to the Oxford Learner's Dictionary, a pattern is, among other meanings, "the regular way in which something happens or is done". We can also find expressions such as *to be patterned after/on something*, which would mean "to be copied from something or to be very similar to it, among many others".

Gamma, Helm, Johnson and Vlissides [GOF] define a pattern as follows: "a design pattern gives name, motivates and systematically explains a general design that allows solving a recurring design problem in object-oriented systems. The pattern describes the problem, the solution, when the solution must be applied, as well as its consequences. It also gives some clues as to how to implement it and some examples. The solution is a distribution of objects and classes that solve the problem. The solution must be customized and implemented in order to solve the problem in a given context".

Thus, we could define a pattern as follows:

> A pattern is a template that we use to solve a problem in software development in order to ensure that the system developed has the same quality that other systems where the same solution has been previously applied successfully.

Abstracting the general elements of a specific solution to obtain this template is what allows us to reuse this solution created for another system in the system under development.

1.2. Advantages of using patterns

Patterns are not created on the spur of the moment, but are the result of having solved the same problem several times and having checked the consequences of applying a particular solution. This way, patterns allow us to take advantage of previous experience when solving our problem. A pattern allows us to do the following:

- Reuse solutions and take advantage of the previous experience of other people who have devoted more effort to understanding the contexts, solutions and consequences than we want or can devote.

- Benefit from the knowledge and experience of these people through a methodic approach.
- Communicate and to transmit our experience to other people (if we define new patterns).
- Establish a common vocabulary to improve communication.
- Encapsulate detailed knowledge about a type of problem and its solutions by giving it a name in order to be able to refer to them easily.
- Not have to reinvent a solution to the problem.

1.3. Disadvantages of an improper use of patterns

Having a catalogue of solutions for general problems does not exempt us from having to think about our development to understand the problems that we encounter during the process. Thus, for example, during the design stage, we must know what our objective is so we can evaluate whether applying a specific solution is better than another. It is quite common when we start working with pattern to assume that applying a pattern is always the best solution possible, which is not always the case.

Once a pattern that seems appropriate has been chosen (in other words, that it is applicable to our current context), we must ensure we have correctly understood the problem it solves and what the consequences of its application are. Reckless use of the pattern can lead us to an inadequate final solution, either because the pattern does not solve the problem we were facing or because we have not been able to evaluate the consequences of its application correctly and, therefore, the disadvantages outweigh the advantages.

1.4. Brief historical perspective

The origin of the term *pattern* can be found in the architecture and urbanism fields. In the late 1970s, Christopher Alexander, along with other authors, published the book *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)* [AIS]. This book contains around 250 patterns described in a specific and standardised format describing those design, planning and construction principles which, throughout their experience, they had detected remained unchanged in any situation. In this way, they managed to transmit a knowledge that, until then, could only be acquired with years of experience.

Most of Christopher Alexander's patterns use the following representation:

```
IF you find yourself in CONTEXT
```

```
  for example EXAMPLES,
  with PROBLEM,
  entailing FORCES
 THEN for some REASONS,
  apply DESIGN FORM AND/OR RULE
  to construct SOLUTION
  leading to NEW CONTEXT and OTHER PATTERNS
```

During the 1980s, many authors started working in this field looking for a way to transfer the benefits of patterns to software development. By the late eighties and the early nineties, people such as Ward Cunningham, Kent Beck, Erich Gamma and others had started to gather their experience in the form of patterns, publishing it with the objective of establishing pattern languages that functioned like the pattern languages created by Alexander.

In 1994, the first PLoP[1] conference dedicated to the discussion and improvement of the publishability of design patterns was held. Earlier that year, the Gang of Four (GOF)[2] completed the work that they had started in 1990 and published the book *Design Patterns. Elements of Reusable Object-Oriented Software* [GOF], which documented, following a standardised format, a catalogue with the twenty-three fundamental design patterns they had encountered during those years. The book quickly became a bestseller and many architects were able to identify in the catalogue the solutions they had reached themselves, establishing for the first time a common language widely adopted by the developer community.

(1)Pattern Languages of Programs http://hillside.net/plop/pastconferences.html

(2)Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides are popularly known as the Gang of Four.

1.5. Structure of a pattern

The elements of the structure of a pattern vary depending on the authors, although there is a set of minimum elements common to most of them.

The five fundamental elements that every pattern must have are the name, the context, the problem, the solution and the consequences of applying it.

The **name** allows us to refer to the pattern when we document our design or when we discuss it with other developers. It also allows us to increase the level of abstraction of the design process, since we can think of the solution to the problem as a whole.

The **context** tells us what conditions must be met for the pattern to be applied.

The **problem** tells us what the pattern solves. This problem could be a specific problem or the identification of a problematic structure (for example, a poorly flexible class structure).

The **solution** describes the elements that are part of the pattern, as well as their relationships, responsibilities and collaborations. The solution is not a specific structure, but, as explained above, it is like a template that can be applied to our problem. Some patterns present several variants of the same solution.

The **consequences** are the results and the obligations derived from the application of the pattern. It is very important that we take them into account when evaluating our design, since it allows us to really understand the cost and benefits of the application of the pattern.

1.6. Application of a pattern

How do we know when to apply a pattern? We will follow this process:

1) Identify the problem we want to solve. The first thing that must be clear is the problem we are trying to solve. We may have to break down a complex problem into different subproblems to handle them individually.

2) Present possible solutions to the problem. These solutions will be presented independently to the pattern catalogue available. The act of presenting the solutions before consulting the catalogue will allow us to have a deeper knowledge of the problem when selecting the patterns of the catalogue.

3) Identify those patterns that solve the **problem** we have detected. We will have to go to our pattern catalogue looking for patterns that could solve the problem we are facing.

4) Discard those patterns whose **context** is not consistent with the context in which we find ourselves. For the application of a pattern to be beneficial, we must make sure that it not only solves the problem that we are interested in, but that it also does so in a context compatible with ours. For example, an architectural pattern that helps us structure an application when the architecture is centralized is useless if our architecture is distributed.

5) Evaluate the **consequences** of each solution. We have to consider the consequences of the application of each of the solutions that we have preselected. With respect to pattern-based solutions, we have the advantage that the general consequences have been previously studied and documented as part of the pattern description. For the rest of the solutions, we will have to study the consequences with the risk of resulting on an incomplete study and without the help of previous experience. Finally, if a pattern has several variants, we have to evaluate the consequences of each one of them.

6) Choose and apply a solution. Once the consequences have been evaluated, we have a well-founded criterion by which to choose the most appropriate solution and, therefore, we can proceed with its application.

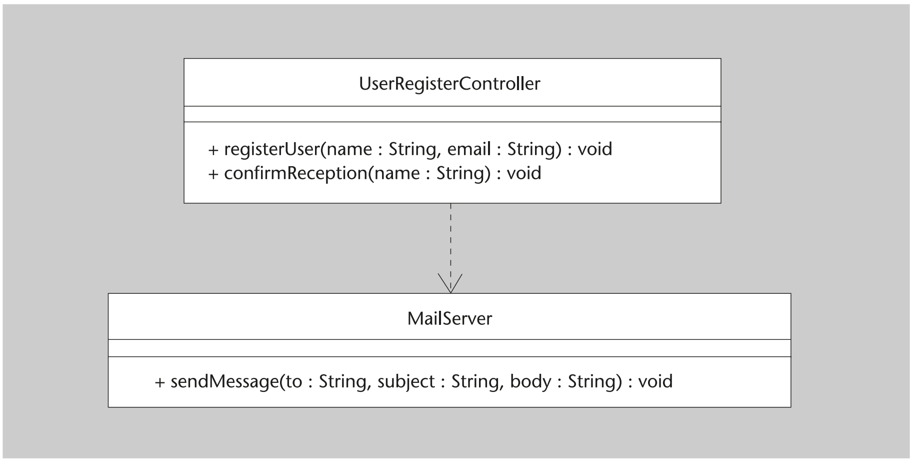
As seen, the process of applying a pattern does not substantially differ from any decision-making process when facing a problem. In the case of patterns, however, we skip studying the variants and their consequences, and we have a guide when developing a solution to the problem.

1.6.1. Example

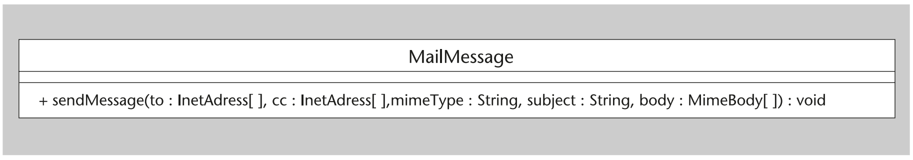
Let's suppose that, when designing our system, we find a requirement according to which, during user registration, a notification must be sent by email to the address indicated by the user. Applying the dependency inversion principle, we start designing this class by identifying the operations we need:

See also

You can see the dependency inversion principle in section 2.7 of module "Pattern catalogue".


UserRegisterController
+ registerUser(name : String, email : String) : void
+ confirmReception(name : String) : void
MailServer
+ sendMessage(to : String, subject : String, body : String) : void

We already had to solve this problem in the past in another project and we think that maybe we can reuse the solution we obtained:


MailMessage
+ sendMessage(to : InetAdress[ ], cc : InetAdress[ ],mimeType : String, subject : String, body : MimeBody[ ]) : void

We find, however, a problem: the operations that we want to use are not exactly the as those offered by the class we want to reuse.

Given this problem we can think of two alternative solutions:

a) Implement the class UserRegisterController using the operations offered by the class MailMessage, so that the controller depends directly on MailMessage.

b) Modify the class mailMessage to add the MailServer operations. In this way, the class is reused, but changing its implementation.

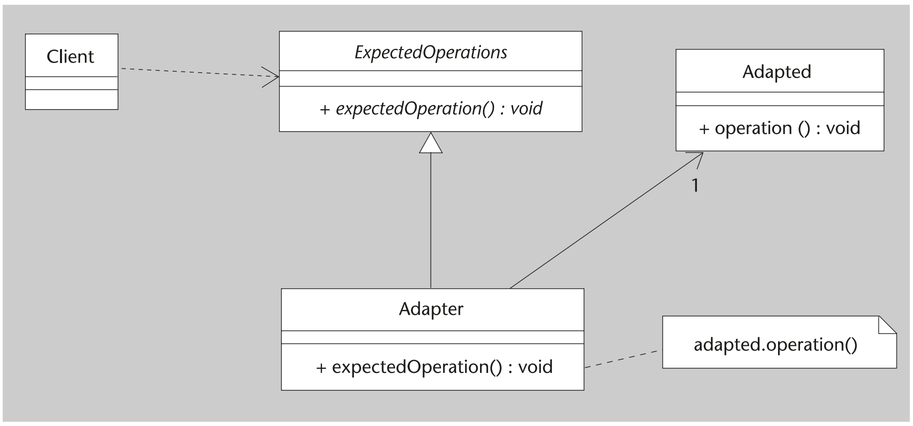
Is there any other way to solve this problem? We consulted the catalogue and found a pattern that solves our problem:

- Name: Adapter

- Context: there is a class that provides us with a functionality that we want to reuse.

- Problem: the operations offered by the class we want to reuse are not exactly the ones we want to use.

- Solution: add a class Adapter that provides the operations that the class Client expects, but which delegates the implementation to the class we want to reuse:



- Consequences:
  - It allows the Adapter to redefine part of the behavior of Adapted

  - The same Adapter can work with different subclasses of the class Adapted. The Adapter can also add functionality to the whole inheritance hierarchy at the same time.

  - We can replace the class Adapted with another implementation by developing a new Adapter.

Once we have seen that there is a pattern that seems, *a priori*, applicable to our problem, the first thing we must do is verify that the context of the pattern is suited to our situation. Do we find ourselves in a situation where "there is a class that provides us with a functionality that we want to reuse"? In this case, it is quite clear that we do.

The next step will be studying the consequences of the three solutions that we have found:

a) Modify the class UserRegisterController.

- We add a dependency to the class MailMessage so that if we want to replace it with another implementation, we will have to modify the UserRegisterController again.

- Since we are not applying a pattern, we have no experience that can guarantee this solution has no consequences we are unaware of at this time.

b) Modify the class MailMessage.

- We are losing the reuse ability of the class, since we are not sharing the code between the two systems where we used MailMessage. Therefore, from now on, we will have to maintain two different versions of the class MailMessage and the improvements we may introduce in one system will not be automatically transferred to the other.

- As in the previous case, there could be consequences that we have not considered.

c) Adapter.

- The solution is more complex, since it introduces an interface and a new class.

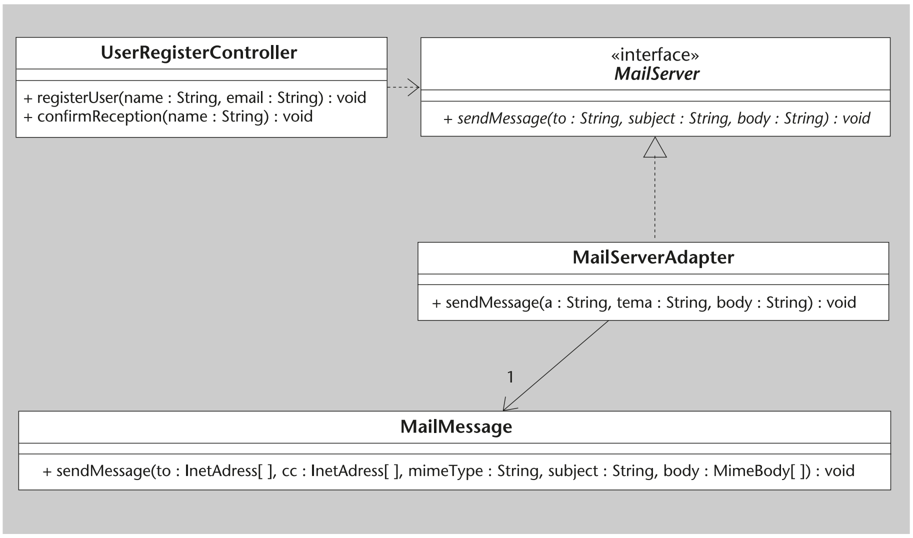
- The consequences of the pattern are already documented: we are reusing the implementation we already have without preventing the possibility of replacing it with another in the future. We can redefine the functionality of the class to make it provide us with exactly what we wanted.

Finally, we have to choose one of the solutions. Do we have to apply the pattern? It depends. To choose the most appropriate solution we must ask ourselves::

a) Is it necessary that our system is independent of the class we will reuse? Are there many classes that have the need to send mail messages through this class? If we reuse MailMessage directly by modifying UserRegisterController to use the current operations, a change in these operations (which could be caused by the other application) will affect our system and, possibly, it will have to be modified.

b) Do we have to keep the reused class independent of our system? If we add our operations to MailMessage, the maintenance of this class will be more complicated, since we will have to maintain two slightly different versions of the same class.

The solution we will choose will depend on the answers to these questions. We assume that we want to maintain our system and the class MailMessage independent from each other. In this case, we will apply the Adapter pattern with the following result:


UserRegisterController
+ registerUser(name : String, email : String) : void
+ confirmReception(name : String) : void
«interface»
MailServer
+ sendMessage(to : String, subject : String, body : String) : void
MailServerAdapter
+ sendMessage(a : String, tema : String, body : String) : void
1
MailMessage
+ sendMessage(to : InetAdress[ ], cc : InetAdress[ ], mimeType : String, subject : String, body : MimeBody[ ]) : void

The client class of the pattern is UserRegisterController, which is who will use the adapted class MailMessage. To adapt this class to the expected interface (MailServer), we have created the adapter MailServerAdapter; in this case, we have slightly modified the structure of the original pattern, since the adapter implements an interface instead of extending a class.

1.7. Influence of patterns in the life cycle

The first application of patterns in the software engineering field was made during the design stage. The patterns used documented design solutions to design problems; in fact, even today, this is the life cycle stage where the use of patterns is more common and where the greatest number of patterns is found.

Patterns are applicable to any task where generic solutions to a certain problem can be identified. Specifically, the concept can be applied to all stages of the software engineering life cycle.

Example

Remember that the original field of patterns was not even software engineering.

Some of the cases in which patterns have been successfully applied are the following:

- **Analysis models**. During the analysis stage, we face problems consisting of deciding which model best represents the concepts and constraints of the real world we are modeling. There are many situations where we face modeling problems that have been solved before and so-called *analysis patterns* can help us. Some examples include the representation of the associations about which we want to know historic information or the modeling of systems that work with accounting rules.

- **Software architecture**. As a first step in the design stage (or as a differentiated stage according to the life cycle that is followed) the overall architecture of the software to be developed must be defined. In this context certain problems will have to be solved with possible generic solutions and several useful patterns for that purpose have been documented. We will call them *architectural* patterns. For example, the layered architectural pattern offers us a way to internally organize our architecture to minimize the complexity of the resulting system.

- **Responsibility assignment**. During design (especially in its initial phases), we have to assign responsibilities to the classes that constitute the system. For example, the Creator pattern gives us a general criterion to decide which class will be responsible for creating instances of another class.

- **Software design**. Patterns are most used during the design stage. There is a wide range of documented *design patterns* that allow for the resolution of common design problems with already known common solutions. For example, as we have previously seen, the Adapter pattern provides us with a solution to the specific problem of reusing a class without using exactly the interface it provides.

- **Programming**. Once we have chosen a technology and a programming language, we will encounter problems that require solutions that cannot be generic to the design or to previous stages, but which have originated in the programming language chosen and, thus, depend on it. In this way, the so-called programming patterns (idioms) are patterns that provide solutions specific to a programming language. As an example, there are patterns that guide us in the use of Java interfaces.

2. Types of patterns

There are many possible taxonomies of software engineering patterns, each one according to different criteria. We will classify the patterns depending on the life cycle stage in which they are used, since it is one of the most accepted and useful classifications.

This will not be considered, however, a strict criterion, since the boundary between one stage and another is not always clear. Thus, for example, some patterns that we will classify as architectural are considered by many authors design patterns.

We will distinguish the following types of patterns:

- Analysis patterns
- Architectural patterns
- Responsibility assignment patterns
- Design patterns
- Programming patterns (idioms)

In this course, we will emphasize the responsibility assignment and design patterns, but we will also see some analysis and architectural patterns.

2.1. Analysis patterns

Analysis patterns are those patterns that document solutions applicable during the realization of the static analysis diagram to solve the problems that arise: they provide us with proven ways of representing general real world concepts in a static analysis diagram.

We can see this more clearly with an example: let's suppose we are developing a management system for a weather station. In the static analysis diagram, we have a Thermometer class that must store the information on the measured temperature. We can think of using an attribute of type integer to represents this data, but we would lack very important information to interpret the value of this attribute: on what scale is this temperature represented? What does a temperature of 100 mean? Are they Celsius, Fahrenheit or Kelvin degrees? So then, we see that the integer attribute is clearly insufficient.

The Quantity analysis pattern [FOW] allows us to solve this kind of problem: we need to add a Temperature class with the responsibility of knowing both the degrees and scale they are in.

2.2. Architectural patterns

Architectural patterns are those that are applied in the software architecture definition and, therefore, solve problems that will affect the whole system design.

For example, in the initial definition of our system architecture, we encounter the problem that its complexity forces us to work at different abstraction levels. We want the presentation components not to interact directly with the technical services at a low level such as the database.

The layered architectural pattern solves this problem by proposing that each component is assigned to a layer with a certain level of abstraction: presentation, domain and technical services, in such a way that each layer depends only on the following one. Thus, the presentation components will work on their own level of abstraction and will delegate any other task to the domain layer.

2.3. Responsibility assignment patterns

Responsibility assignment patterns are used during the design stage. But this type of pattern solves problems during a very specific task of object-oriented design: responsibility assignment to software classes.

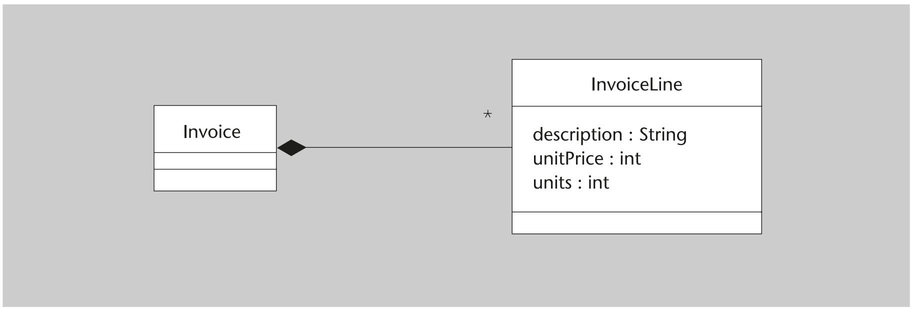
This type of design patterns clearly differs from general design patterns in that the problems they solve are not specific to a particular situation or system, but they are problems common to any responsibility assignment. These patterns, in fact, consist of guidelines to help us make responsibility assignment following the most basic design principles.

All responsibility assignment patterns share a very similar structure. They answer the question: "Which class must have the responsibility R for the design to follow solid design principles?". When we start the design, based on the conceptual model of the analysis, we create a class model of the design. At this time, we start assigning responsibilities that will collaborate to implement the system behavior.

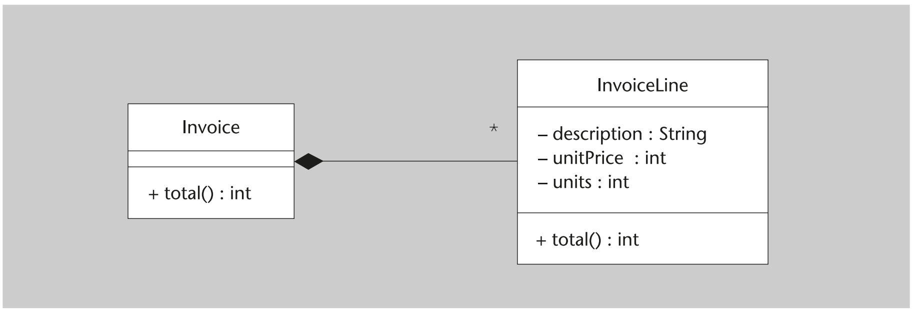
Example

Larman, author of the GRASP [LAR] pattern family, which is the most important example of responsibility assignment patterns, documents design principles in the form of patterns. Thus, for example, for Larman it makes sense to about of the High cohesion and Low coupling patterns.

Let's suppose we have the following partial model:



Which class will be responsible for calculating the total of an invoice? According to the Expert responsibility assignment pattern, it will be the class that has the necessary information to make this calculation. In our example, the class Invoice has the information of the invoice lines that form an invoice and will be the expert chosen. To calculate the total of the invoice, however, this expert will have to add the total of each line. Which class will have the responsibility to calculate the total line? Again, according to the Expert pattern, the class that has the necessary information is InvoiceLine. Therefore, our design model is as follows:



2.4. Design patterns

Design patterns are those that are applied to solve specific design problems that will not affect the whole system architecture.

We do not include responsibility assignment patterns in this group although they are also used in the design stage.

The example of the Adapter design pattern application we have previously seen is a good example of application of a pattern of this group.

See also

You can also see the Adapter design pattern example in section 1.6.1 of this module.

2.5. Programming patterns (idioms)

A programming pattern (idiom) is a pattern that describes how certain tasks can be implemented using a specific programming language.

Therefore, programming patterns (idioms) solve problems that are either specific to a programming language or are generic, but have a known good solution that is specific to a programming language.

For example, during analysis and design, we have used in our models an enumeration called Colors that can take the values Red, Blue and Green. When we implement the system in Java (version 1.4), we encounter the problem that it does not have support for enumerations. The programming pattern "Enumerations in Java" allows us to solve this problem by creating a class Color with a protected constructor that will have three attributes, one for each possible value:

```
public class Color {
 public static final Color RED = new Color();
 public static final Color GREEN = new Color();
 public static final Color BLUE = new Color();
 protected Color() {}
}
```

Now we can use this class as follows:

```
if(color == Color.RED)
...
```

In the course, we will focus on the highest level patterns and, therefore, we will not see programming patterns.

3. Frameworks

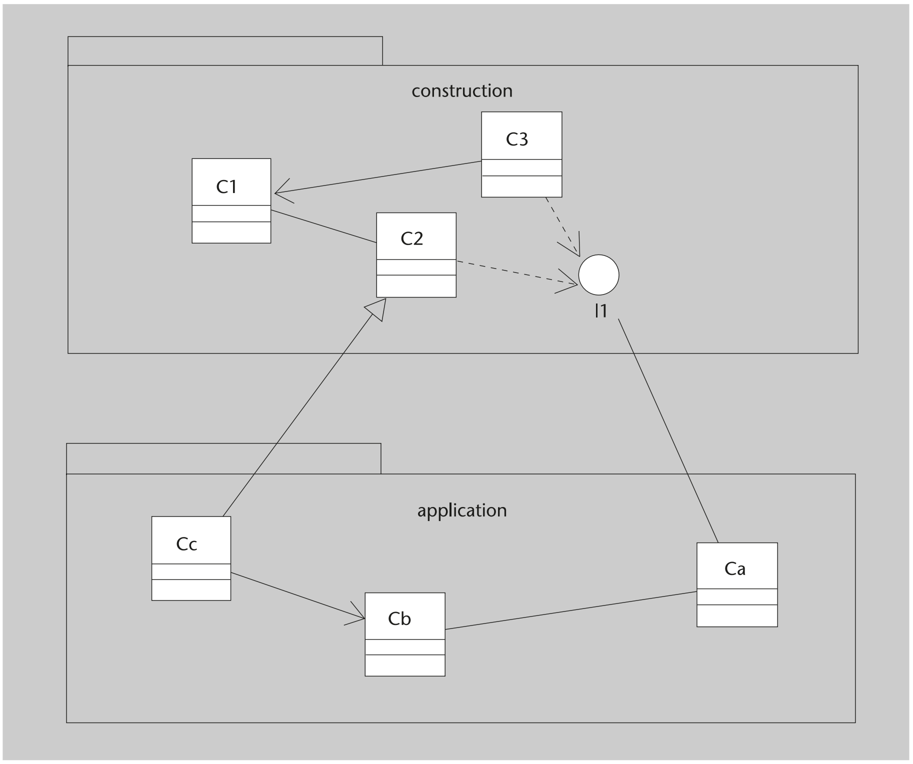
A framework is a set of predefined classes that we must specialize and/or instantiate to implement an application or subsystem saving time and errors. The framework offers us a generic functionality already implemented and tested that allows us to focus on the specificity of our application.

We must not mistake a framework with a class library. A framework is a special type of library that defines the design of the application that will use it.

A framework shares the execution control with the classes that use it following the "Hollywood principle":[3] the framework will implement the generic behavior of the system and will call our classes at those points where a specific behavior must be defined.

(3)"Don't call us. We'll call you".

Since a framework has to call our classes, it will define the interface that it expects the classes will have. Thus, to use a framework we will have to implement the specific behavior of our application respecting the interfaces and collaborations defined in it.



In this case, for example, our application uses a framework through inheritance (from class Cc towards C2) and the implementation of an interface (Ca implements I1) used by the framework (by C2 and C3). In this way, the framework will take control of the execution by calling the Ca and Cc classes of our application when needed.

3.1. Advantages and disadvantages of using frameworks

When we use a framework, we reuse a design, saving time and effort. On the other hand, since the control is shared between the application and the framework, a framework can contain much more functionality than a traditional library.

Another advantage of using frameworks is that they give us a partial implementation of the design we want to reuse.

The main disadvantage of the use of a framework is the need for an initial learning process. To reuse the design of the framework, first we must completely understand it.

3.2. Relationship between frameworks and patterns

The main difference between a pattern and a framework is that the framework is a software element, while the pattern is an element of knowledge about the software. Therefore, the application of a pattern will start from an idea, but not from an implementation; in fact, the pattern will have to be implemented from scratch adapting it to the specific case. In contrast, the application of a framework starts from a set of already implemented classes, the functionality of which we will extend without modifying them.

From the pattern point of view, a framework can implement a set of patterns for us. That is, the design we reuse can be based on identified patterns just as if we had done it. In this case, understanding the patterns which a framework is based on will help us understand its functioning allowing us to effectively use it.

3.3. Use of frameworks

The use of frameworks affects our software development process. Since we are reusing a previous design, we have to adjust our system to the design we want to reuse.

Although that may seem a limitation, the ability to present our design in terms of a design design that was previously applied successfully by the authors of the framework will help us ensure the quality of the design.

3.4. An example framework: Hibernate

When we develop a system that stores its data in a relational database, it is common to encounter the problem of synchronizing the data of the objects we have in memory with the data from the relational database. This problem is that many software applications cannot take full advantage of the object-oriented technology, as they have to work directly with the relational data model.

Hibernate is a framework for Java objects to relational databases persistence that allows us to add, in a transparent manner, the ability of persistence to the classes we design. This structure offers us a series of mechanisms by which we can work with persistent objects in the database from an object-oriented point of view with the support, therefore, for concepts such as associations, inheritance, polymorphism, composition and use of collections. Hibernate also defines a proprietary query language called *Hibernate query language*, that allows for the creation of queries with SQL capability with some extensions to support the use of objects.

To develop an application with Hibernate, it is essential to have at one's disposal the relational model and the design of the classes. Starting from this point, we will define the correspondence between Java classes and tables in the relational model using an XML-based language. Then, all that needs to be configured is how it will connect to the database so that Hibernate transparently deals with synchronizing the object-oriented data model in memory and the relational model of the database server.

3.4.1. Advantages

- It allows us to work on relational databases using object-oriented capabilities. This feature causes a reduction in development complexity with the consequent increase in productivity, reliability and maintainability.

- It includes numerous performance improvements compared to other solutions such as loading data on request (which avoids loading data that finally will not be consulted), transparent optimization of SQL queries (thanks to the knowledge of the different RDBMS[4] that the framework has) or the use of a cache memory.

(4) Relational Database Management Systems.

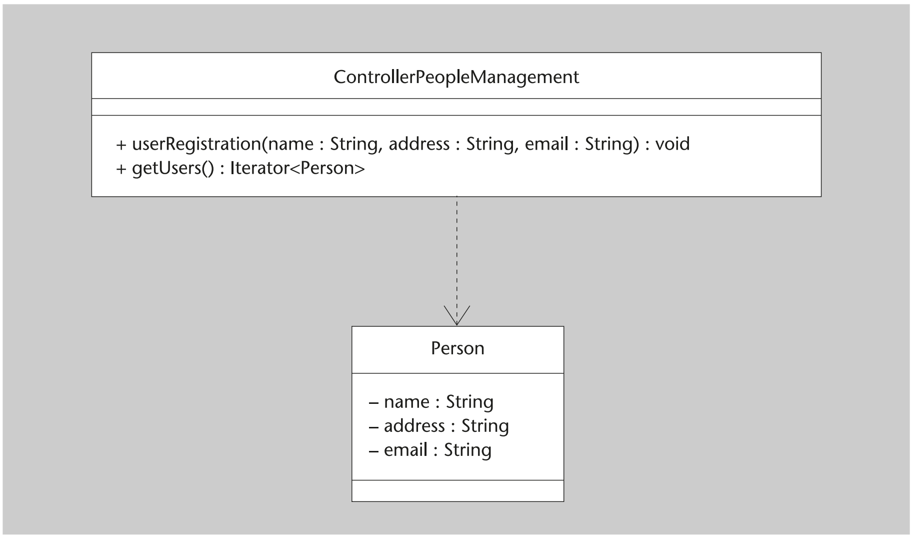
- It allows us to change the SQL dialect (and, hence, the database manufacturer) without having to modify the code.

3.4.2. Disadvantages

The main disadvantage of Hibernate is the need for an initial learning process both with respect to its programming model and to the specific interface it offers.

3.4.3. Example of use

We want to design a people data management application with the following domain class model:


ControllerPeopleManagement
+ userRegistration(name : String, address : String, email : String) : void
+ getUsers() : Iterator<Person>
Person
– name : String
– address : String
– email : String

Our relational model is very simple:

```
create table Person (
  name varchar (32),
  address varchar (80) not null,
  email varchar(80) not null,
  primary key (name)
)
```

To adapt our design to Hibernate, we must define the correspondence between classes and tables using the following XML document:

```
<hibernate-mapping>
   <class name="domain.Person" table="Person">
   <id name="name" type="string" >
     <generator class="assigned"/>
   </id>
<property name="address"/>
<property name="email"/>
</class>
</hibernate-mapping>
```

Next, the controller behavior must be modified so that it uses the framework:

```
public class ControllerPeopleManagement {
 public void registerPerson(String name, String
 address, String email) {
     Session session = HibernateUtil.currentSession();
     Transaction tx = session.beginTransaction();
     Person p = new Person();
     p.setAddress(address);
     p.setEmail(email);
     p.setName(name);
     session.save(p);
     tx.commit();
     HibernateUtil.closeSession();
}
public Iterator<Person> queryPeople() {
 List<Person> result = new LinkedList<Person>;
 Session session = HibernateUtil.currentSession();
 Transaction tx = session.beginTransaction();
 Query query = session.createQuery("from Person");
 for (Iterator it = query.iterate(); it.hasNext();) {
     Person p = (Person) it.next();
     result.add(p);
 }
 return result.iterator();
}
```

As it can be seen, the class Person will not need to be modified, since Hibernate takes care of the persistence in a transparent way by calling the operations of Person needed. Regarding the controller, although it has a specific part of code for persistence, the proportion of this part with respect to the whole code is very small.

As seen, using Hibernate is much less intrusive than using, for example, SQL statements within the code. On the other hand, it allows us to work at a higher level of abstraction and avoids the need to be aware of freeing up resources once used, obtaining connections, etc.

3.4.4. Hibernate and patterns

It should be noted that Hibernate uses design patterns in its construction, such as the Proxy pattern. In addition, the Hibernate community of this framework has defined a whole catalogue of programming patterns specific to Hibernate.

Thus, it is clear that the relationship between a framework and patterns is twofold: on the one hand, they help the framework to be better designed and, on the other hand, the use of the framework generates a new catalogue with its own patterns.

Summary

Patterns have become a very important tool in software development because they allow us to benefit from reuse in contexts other than code reuse.

We have studied the general concept of pattern, how a pattern is documented and how this tool can be applied to the different stages of the development life cycle. We have seen the advantages and risks derived from pattern misuse.

We have also seen a pattern classification according to the stage of the development life cycle in which they are applied. This classification will be useful to quickly locate and relate to each other the different patterns of the catalogue that will be collected in the corresponding module.

Finally, we have seen the concept of framework and its relationship with the analysis and object-oriented design with patterns, as well as a practical example of use of the Hibernate framework.

Self-evaluation

1. List the advantages of using patterns.

2. Name two reasons why the use of a pattern can be considered inappropriate.

3. Is the use of patterns exclusive to software engineering?

4. What are the fundamental elements of the framework of a pattern?

5. How does pattern use contribute to the decision-making process during, for example, software design?

6. What is the difference between an analysis pattern and a design pattern?

7. Explain what the so-called “Hollywood principle” consists of.

8. Name an advantage and a disadvantage of using frameworks.

Answer key

1. Reusing previous solutions, benefiting from the knowledge of others, communicating and transmitting the experience, establishing a common vocabulary, encapsulating knowledge and not having to reinvent a solution to a problem.

2. Because it does not solve the problem we faced or because we have not properly evaluated the consequences of its use and the disadvantages outweigh the advantages.

3. No; in fact, the term *pattern* comes from the architecture and urbanism world.

4. Name, context, problem, solution and application consequences.

5. The process is more or less the same, but we can avoid studying the consequences of applying the pattern and we have a guide when developing a solution to the problem.

6. The fundamental difference lies in the stage of the development life cycle where they are used.

7. This principle makes reference to the fact that the system execution control is shared by a framework and the application that uses it. It consists of the fact that the framework is the one who calls our classes at those points where it is necessary to define a specific behavior.

8. An advantage is that they give us a partial implementation of the design that we want to reuse. A disadvantage is that, to reuse this design, first we have to fully understand it, so an initial learning period is necessary.

Glossary

adapter A design pattern that allows us to reuse a previously existing class using a set of operations different than the one that the class provides.

class library Set of reusable and interrelated classes.

conceptual model syn. **static analysis diagram**

context Element of the pattern description that indicates what conditions must be met for the pattern to be applicable.

expert Responsibility assignment pattern that proposes to assign a responsibility to the class that has the necessary information to carry it out.

framework Set of classes that we must specialize and/or instantiate to implement an application or subsystem.

GRASP Main responsibility assignment pattern family documented by Craig Larman.

Hibernate Persistence framework of Java objects in relational databases.

idiom Specific pattern for a particular technology, typically a programming language.

layered architecture Architectural pattern that proposes the organization of our architecture in a set of layers that work at different abstraction levels.

life cycle Set of software development stages in a previously established order.

pattern Template that we use to solve a problem during software development.

pattern application consequences Element of the pattern description that indicates the results and compromises derived from the application of the pattern.

programming pattern syn. **idiom**

quantity Analysis pattern that allows for the modeling of a quantity indicating the scale on which it is measured.

responsibility assignment Design task in which it is decided which class has to be in charge of each responsibility within an object-oriented system.

static analysis diagram Class diagram representing the real world concepts that our system knows and manipulates.
syn. **conceptual model**

Bibliography

[AIS] Alexander, C. and others (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press.

[GOF] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts: Addison Wesley Professional.

[FOW] Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Massachusetts: Addison Wesley Professional.

[LAR] Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. New Jersey: Prentice Hall.

[POSA] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. (1996). *Pattern-Oriented Software Architecture: A System Of Patterns*. West Sussex, England: John Wiley & Sons Ltd.