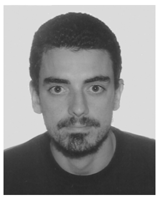
Patterns catalogue

PID_00273624

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

A catalogue of patterns that enables the quick search and reference is needed to apply these patterns to the different stages of the development life cycle.

This module contains an organized catalogue according to the stages of the software development life cycle in which the patterns can be applied (analysis, architecture, responsibility assignment and design). From each stage, those patterns that have been considered the most representative of their category and whose use is more common have been chosen.

According to the structure seen in the previous module, "Introduction to patterns", for each pattern we explain the problem it targets, the proposed solutions and the consequences (advantages and disadvantages) of its application. We also point where, in the reference book, the pattern was originally published in order to give the possibility of expanding the information available both on the pattern itself, and on other related patterns.

This module can also be used as a reference while doing the course exercises.

Objectives

The main objective of this module is to show a set of patterns and make its understanding easy through examples. It is also intended to be a reference catalogue where to find the patterns that help us improve the analysis and design of information systems to be developed. The specific objectives that the student must achieve with this module are the following:

1. To know a selection of patterns applicable to the different stages of the software development life cycle.

2. To know the different patterns, as well as their application.

1. Reference table

A reference table with a summary of all the patterns of the catalogue is included below. For each pattern its name, type, the section of the module where it is explained in detail and a summary is indicated; we also indicate some bibliographic references.

Name	Type	Reference	Section	Summary
Adapter	Design	GOF, LAR, MAR	6.8.1	It allows the use of a class with a different set of operations than it originally offers.
Layered architecture	Architecture	POSA, LAR	4.1	It organises a system making each element work at a single level of abstraction.
Historic association	Analysis	FOW	3.1	It represents the values of an association over time.
Controller	Responsibility assignment	LAR	5.1	It assigns the responsibility of handling events to a controller object.
Creator	Responsibility assignment	LAR	5.4	It assigns the responsibility to create instances to an object.
Decorator	Design	GOF, MAR	6.8.2	It adds responsibilities to an object without modifying its class.
State	Design	GOF, LAR, MAR	6.1	It allows an object to vary its behavior depending on its state.
Strategy	Design	GOF, LAR, MAR	6.8.3	It allows us to choose among a family of algorithms.
Expert	Responsibility assignment	LAR	5.2	It assigns a responsibility to the object that has the information to perform it.
Pure fabrication	Responsibility assignment	LAR	5.3	It creates a new class to assign it a domain independent responsibility.
Facade	Design	GOF, LAR, MAR	6.2	It reduces coupling between subsystems.
Dependency injection	Architecture	FOWDI	4.2	It allows us to choose and change the implementation of the services without having to change the code of the clients.
Singleton	Design	GOF, LAR, MAR	6.8.4	It ensures, for a class, that there can only be one instance.
Iterator	Design	GOF	6.3	It allows us to traverse the elements of a collection.
Factory method	Design	GOF, LAR	6.4	It creates instances of a class without knowing the exact subclass.
Template method	Design	GOF, LAR, MAR	6.5	It implements several similar algorithms reusing the common part.
MVC	Architecture	POSA	4.3	It organises interaction responsibilities with the user.
Compound object	Analysis	GOF, LAR, MAR	3.2	It allows us to handle a collection of objects and their elements in the same way.

Name	Type	Reference	Section	Summary
Null object	Design	MAR	6.8.5	It treats the null value as an object.
Observer	Design	GOF, LAR, MAR	6.6	It notifies changes on the state of an object to other objects.
Order	Design	GOF, LAR, MAR	6.7	It treats an operation invocation as an object.
Quantity	Analysis	FOW	3.3	It represents a measure with its unit.
Range	Analysis	FOW	3.3	It represents a range of values.
Proxy	Design	GOF, LAR, MAR	6.8.6	It controls access to an object.
Abstract server	Design	MAR	6.8.7	It reduces coupling between clients and servers.

2. Design principles

In order to evaluate the quality of a design, it is necessary to establish certain principles that it must comply with. In this way, we will say that a design is of quality as long as it conforms to these design principles.

The purpose of applying design patterns is to improve the quality of the system developed by taking advantage of previous experience. Therefore, when deciding on the application of patterns, we will have to do so in the light of the principles mentioned above. Therefore, in this module, a small catalogue of design principles is included.

Next, a series of widely accepted design principles that, in some cases, can help us in other stages of the development life cycle is given.

2.1. Low coupling

We have to minimise coupling.

Coupling measures the dependency between elements (such as classes, packages, etc.), typically as a result of the collaboration between elements to provide a service.

When a class has a high coupling with respect to other classes, we find the following:

- A change in the related classes can cause a change in our class.
- It is difficult to understand our class in an isolated way.
- It is difficult to reuse our class, since its use requires the presence of the related classes.

2.2. High cohesion

We have to maximise cohesion.

Cohesion measures the degree of relation between the different responsibilities of a class. The more related these responsibilities are to each other, the higher the cohesion of a class is.

A class with low cohesion has the following problems:

- It is hard to understand.
- It is difficult to reuse.
- It is hard to maintain.
- It is fragile with respect to the changes that occur in the system.

A more restrictive version of this principle is the one known as the single-responsibility principle (SRP): "A class should have responsibility over a single part of the functionality provided by the software".

Complementary bibliography

More information on the SRP principle can be found in [MAR].

2.3. Open-closed principle (OCP)

A software entity (class, module, function, etc.) should be open to extension but closed to modification.

Complementary bibliography

More information on the open-closed principle can be found in [MAR] and [MEY]. Also in [LAR] under the name of *Protected Variations*.

In design terms, this principle states that, in order to add new responsibilities to our system, we must be able to do so by adding new classes (extension), but without modifying the existing ones.

One way to comply with this principle is to create abstractions around the aspects that we anticipate that have to change so that a stable interface can be created with regard to the changes. With this solution, our system will be open to extension (adding new implementations of the defined interfaces) but closed to modification (we do not have to modify the defined interfaces).

Principle of Protected Variations

Our design must minimise the impact on the current system of changes that may occur in the future and that we can anticipate.

2.3.1. Law of Demeter (or the Principle of Least Knowledge)

Demeter's law, sometimes summarised as "Don't talk to strangers", is a heuristic that helps us comply with the open-closed principle:

An object operation should only use:

- The operations of the same object.
- The associated objects (or its attributes).
- The objects that the operation receives as a parameter.
- The objects that the operation creates.

This law helps us reduce coupling with respect to a specific class structure. In addition, it enhances encapsulation, since, in order to access the objects associated with an object O, we will have to do it via the operations of the object O, so that we make sure that it will find out about the access or manipulation.

Law of Demeter

Sometimes the name *Law of Demeter* is used as a design principle instead of the open-closed principle.

2.4. Don't repeat yourself (DRY)

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Complementary bibliography

More information on the Don't repeat yourself principle can be found in [HUNT].

This principle states that we have to avoid, whenever possible, the duplication of information and responsibilities. If we achieve this, our system will be simpler and easier to maintain, since, when facing a change or an error, we can easily identify which the affected component is.

2.5. Liskov substitution principle (LSP)

The instances of a superclass C may be replaced with the instances of the subclasses of C.

Complementary bibliography

More information on the Liskov substitution principle can be found in [MAR].

This principle indicates that a good inheritance hierarchy must respect the behavior of the superclasses. A class or program that may use instances of a superclass S should be able to use any instance of a subclass of S without its behavior being negatively affected.

2.6. Interface-segregation principle (ISP)

Clients would not have to depend on operations they do not use.

Complementary bibliography

More information on the interface-segregation principle can be found in [MAR].

A class may offer many operations. In these cases, it is normal to find that some of the client classes only use a subset of the operations. This principle states that, in these cases, we must split the interface of the class into subsets of operations in order to avoid the coupling of the client classes involving operations that they do not use.

2.7. Dependency inversion principle (DIP)

High level modules or classes should not depend on low level ones, but rather on an abstraction.

Abstractions should not depend on the details. Details must depend on abstractions.

Complementary bibliography

More information on the dependency inversion principle can be found in [MAR].

In order to maximise the reuse of our classes, we have to avoid coupling with regard to lower level classes. That will allow us to limit the impact of a change to the level of abstraction at which it occurs. For example, if we change the database server, the domain layer should not be affected.

3. Analysis patterns

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

See also

More information on analysis patterns can be found in section "Analysis patterns" of the "Introduction to patterns" module.

3.1. Historical association

Context

A class A participates in an association of grade N.

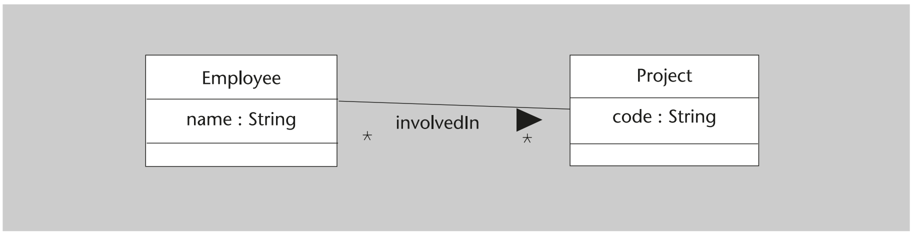
Problem

We want to be able to recover the values that the association has had in the past.

Specifically:

- We must be able to know the value of the association at a given time.
- We must be able to get the list of values of the association in a period of time.

For example, let us suppose that the system we are developing needs to store information on the projects which an employee has been involved in. We depart from the following static analysis diagram:

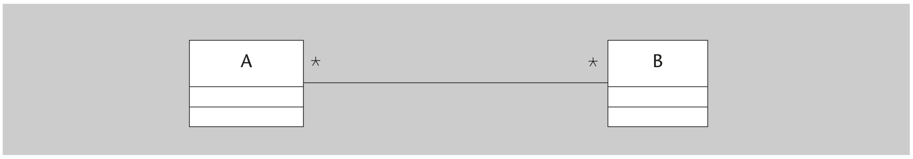


This solution, however, only serves us to know, at the present time, in which projects an employee is involved in. How can we know in which projects was each employee involved in last year? We call this information on the state of the system in the past "historical information" and it is a fairly common situation in information system development.

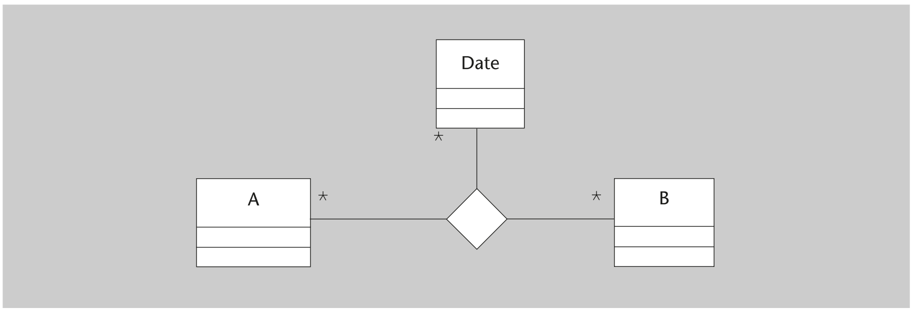
Solution

Adding a dimension to the association (converting the *n*-ary association into *n*+1-ary) that represents time.

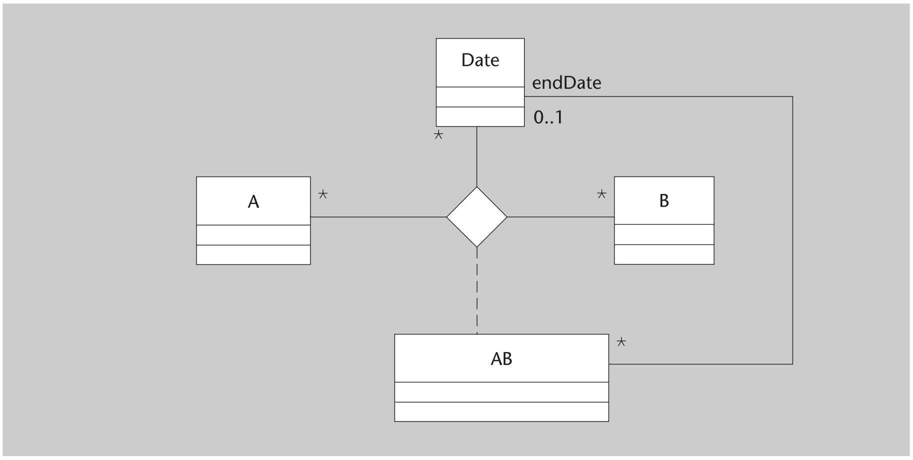
For an association as the following:



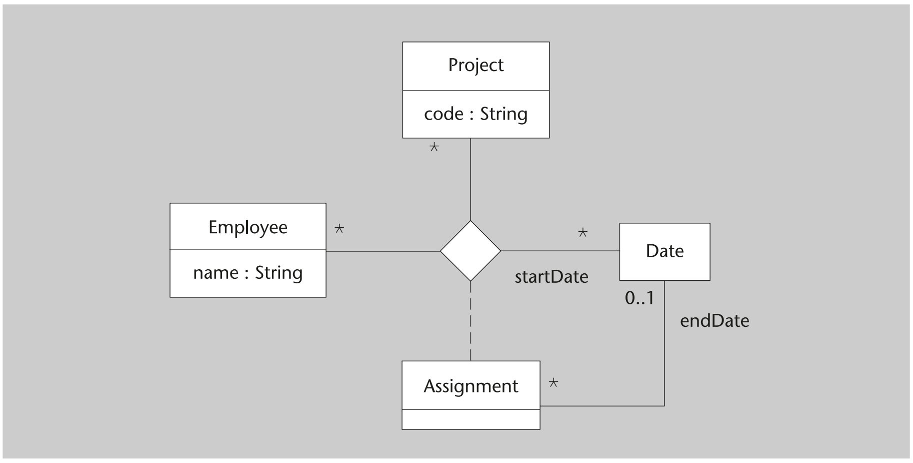
the solution would be this:



If the values that the association takes have a certain validity period, the final moment of this period will be indicated as an association of the association class:



Returning to the employee projects case, the solution would be the following:



Now our system knows, for each Employee-Project couple, the date on which this assignment took place and the date on which the assignment ended. To know what is the current assignment, we will have to look which is the most recent date or we could also consider that it is the one which does not have an end date yet.

When defining new cardinalities, we must be careful. In this case, the cardinality next to Employee will indicate how many instances of employee can be associated to a same project with the same date, but it does not indicate anything with regard to other times in time (for example, an employee may be assigned more than once to a project as long as the assignment date is different).

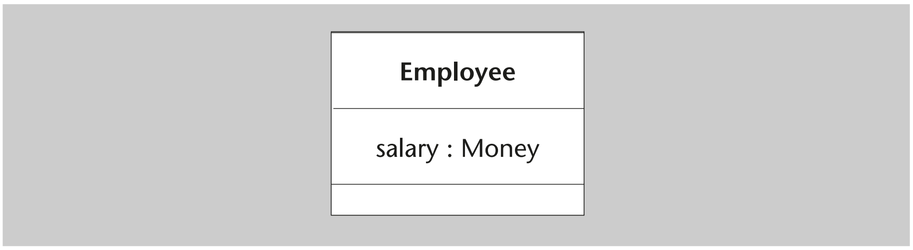
Consequences

- It enables the reflection of historical information correctly.
- The result is more complex, since the degree of the association has been increased.
- The multiplicity of the original association cannot be transferred to the historical association.
- The diagram does not indicate that time period cannot overlap.

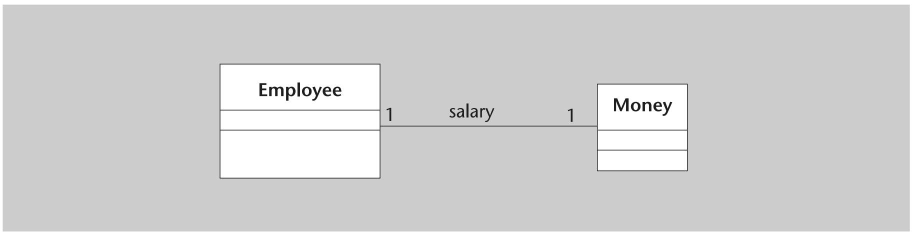
Variations

If instead of an association what we have is an attribute of which we want to know the history of values, we must make a previous transformation: we must represent the attribute as a binary association between the class and the type of data that corresponds and then apply the pattern as usual.

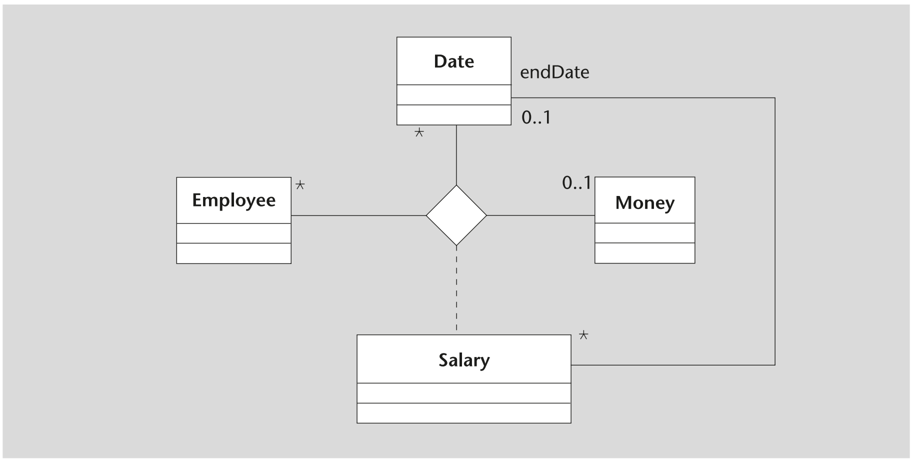
For example, we assume that the system we are developing has employees whose history salary must be known:



To apply this pattern, we must first convert the attribute into a binary association:



Once the transformation is done, we apply the pattern in the way we have seen before:



Some details about the example of pattern application:

- In this case we have assumed that the same employee cannot change salaries more than once on the same date (restriction which is reflected in the diagram by cardinality 0..1 on the Money side). If this were not the case, the cardinality on the Money side would be *.

- The diagram does not reflect, however, that an employee cannot have two salaries on the same date (that is, the time period for which each salary is defined cannot overlap).

Variations

A set of analysis patterns dedicated to more specific aspects related to the problem of time representation can be found in [FOW].

3.2. Composite Object

Context

There is an association that groups elements into collections.

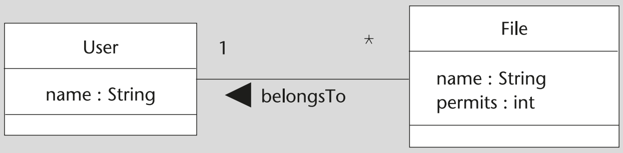
Problem

We want to handle collections of elements and elements uniformly.

For example, let us suppose the information system we are developing uses files to store information. From a specific user we know the files that belong to him or her, and also the name and permits of each file:

Analysis or design pattern?

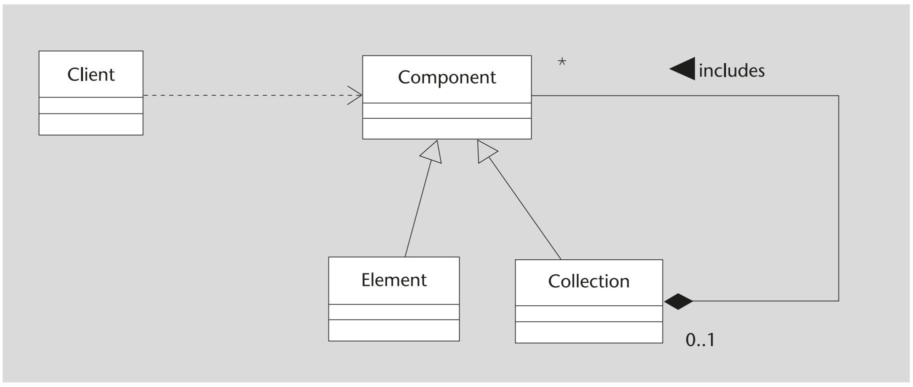
This pattern is usually considered a design pattern. We have included it as an analysis pattern because it can also be very useful at this stage of the software development life cycle.



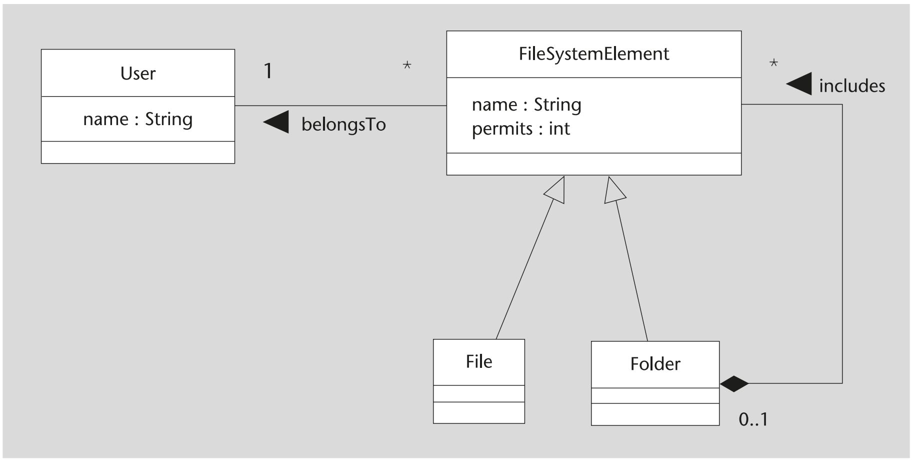
Now, however, we want to add the possibility of grouping the files into folders. On these folders we want to apply the same treatment as on the files (that is, they will also have name and permits). How can we do it?

Solution

Creating, via a generalisation, a superclass common to both the elements and collections and making the rest of the system to not know the specific subclass with which it is associated.



In our example, the result of applying the pattern would be as follows:



Consequences

- Elements and collections are of the same type, which solves the problem of uniform treatment.
- More complex solution: we have made a generalisation and, therefore, a new class has been added.

3.3. Quantity

Context

A quantity wants to be registered.

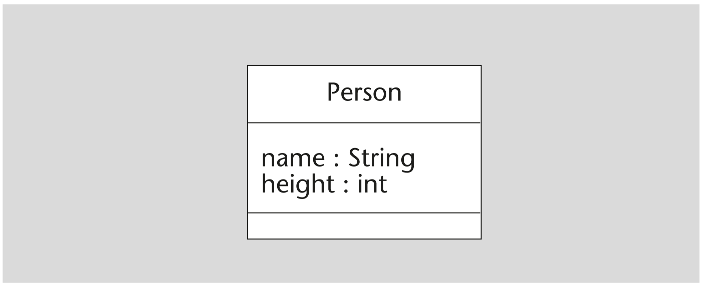
Problem

The representation of a quantity via a numerical value is a poor solution. It can lead to confusion or is not feasible when different measurement units can intervene.

Specifically, we may want:

- to make explicit the unit of measure;
- to use different measurement units;
- to be able to convert between units in a simple way.

For example, let us suppose that the information system we are developing must manage information on people, specifically height, thus we add an integer attribute that contains the person's height:



This solution, however, is not enough, since it does not specify in which unit the height is expressed. Is it in feet? In inches? A possible solution may be to indicate this unit of measure as part of the attribute name. Thus, for example, we could change the attribute name to heightInFt to indicate that the height is indicated in feet. But there are still some problems that cannot be solved:

- The unit of measure is implicit in the attribute name and, therefore, remains unclear with respect to the analysis.

- The unit of measure indicated in the attribute is always forced to be used.

- If the measure available is in another magnitude, the conversion will have to be done.

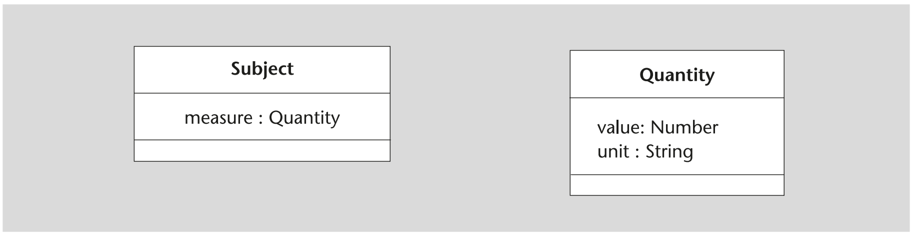
- The analysis does not reflect how this conversion can be done.

- Different measures in different units cannot be registered.

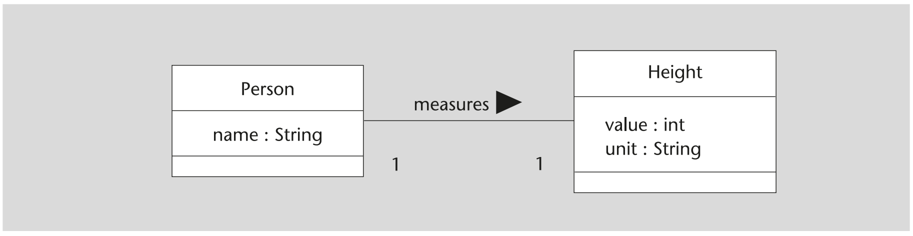
What solution can be applied in a situation like this that allows us to solve all the problems mentioned above?

Solution

Modeling the measure as a more complex type of data formed by the value and unit fields.



We can see, then, that in this way we can solve the problems we have mentioned before with regard to the analysis and, later, during the design stage, we can solve the problem of the conversions between units without major difficulties. In our case, the result would be the following:



We have to take into account that, if two people have the same height (for example, 5.6 ft), each one will have associated a different instance of Height with the same values in their attributes. In this way, if the height of one of the two people is modified, the other will not. Therefore there is cardinality 1 on the Person side.

Consequences

- We have more classes, which causes the model to gain expressiveness in exchange for losing simplicity.

- We can handle the quantity as an abstract type of data, adding the behavior we believe necessary (conversions, comparisons, etc.) increasing, in this way, the global cohesion of the system modeled.

- The unit of measure is explicit in the static analysis diagram.

- Different units of measure can be used.

Variations

Other analysis patterns documented in [FOW] (see bibliography) which we will not see in detail are related to the Quantity pattern, which enables a much more complete analysis model that provides, among other things:

- Conversion between units of the same magnitude (such as conversion ratios from centimeters to inches).
- Composite units (such as ft/s^2, to measure accelerations).

- Measures and observations: having knowledge of the moment and the mechanism by which an observation was made.

These last analysis patterns, however, are specific for domains of scientific and, especially, medical fields.

3.4. Range

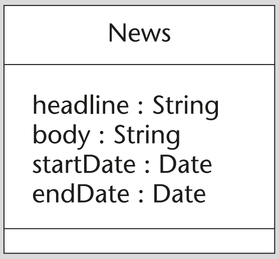
Context

We want to represent a range of values.

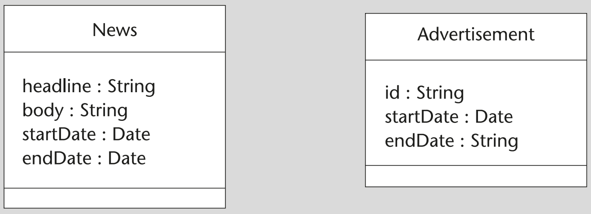
Problem

Our analysis should reflect the semantics of a range, such as knowing if two ranges overlap or if a value is within a range.

For example, let us suppose that the information system we are developing should let us publish news to the Internet. This news will have a validity period. One solution could be to add these dates to the News class:



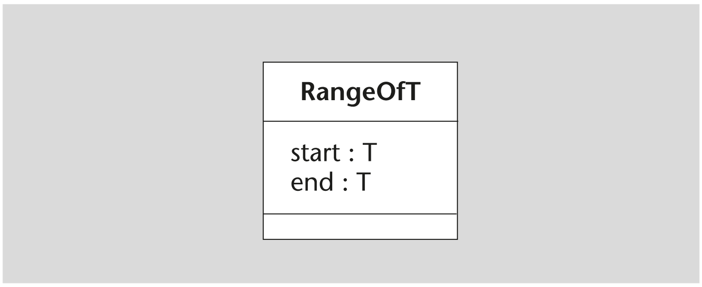
The problem with this solution is that we are adding to the class News all the responsibilities associated with managing the date range (comparing them, managing them, etc.). In addition, if we have more classes with date ranges, the solution is complicated, since the information about how date ranges works is scattered:



Solution

Creating a class that represents a range of values which will be responsible for containing the semantics of the range.

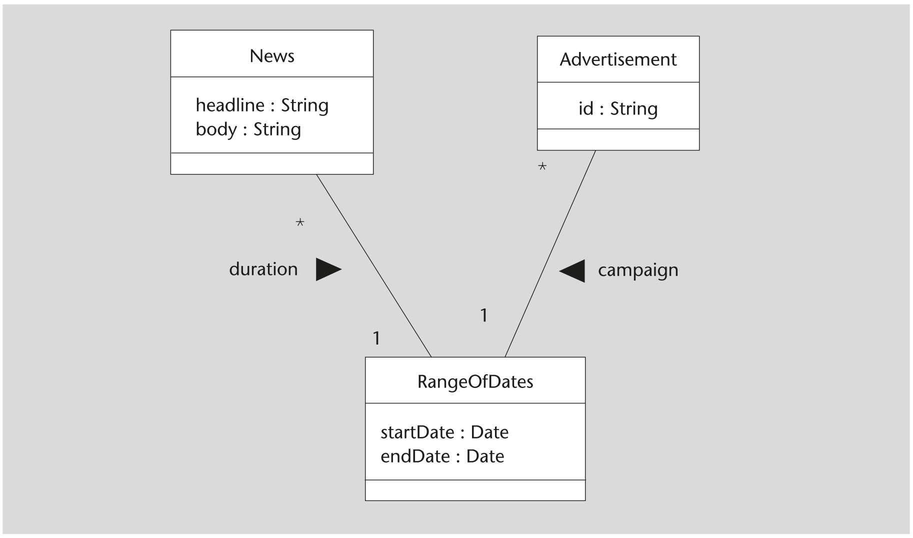
This class will have a start value and an end value. We will consider that an instance of the range contains all the values between the start and the end value and that two instances overlap if they contain any common value:



Example

Optionally, we could also use a parameterised class to represent the range.

In the case of news and advertisements, the solution would be the following:



Consequences

- It can only be applied with types that support comparison (that is, that have a default order defined), although a more sophisticated version could be created in which the sorting criteria could be indicated.

- How do we handle open ranges (for example, greater than six)? We can use a special value (the null value, or the maximum representable, etc.) as an indicator of the lack of limit and encapsulate this fact within the class so that clients are not affected.

- In addition to the operations that indicate that a value belongs to a range, we can add operations that compare ranges, that tell us whether they overlap, etc.

Variations

Instead of having a start attribute and an end attribute, we can have a start attribute and a length.

4. Architectural patterns

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

See also

More information on architectural patterns can be found in section "Architectural patterns" of the "Introduction to patterns" module.

4.1. Layered Architecture (Layers)

Context

We want to structure our system in such a way that each component works at a certain level of abstraction.

Problem

- Our system must work at different levels of abstraction.

- High-level components cannot directly use services with a much lower level of abstraction because we want to minimize the complexity of the resulting system. For example, we do not want graphical interface classes to directly access the database.

- We want to design an architecture that prevents changes in parts of the code to spread throughout the system, reducing coupling between parts.

- We want to prevent the application logic and its user interface from being excessively coupled to be able to reuse, if necessary, the same logic with a different interface.

- We want to prevent the business logic and the technical services of the application from being excessively coupled to be able to easily reuse, distribute or replace a technical service by a different implementation.

For example, the system we are developing manages the salary information of the employees of a company. The system will work at different levels of abstraction:

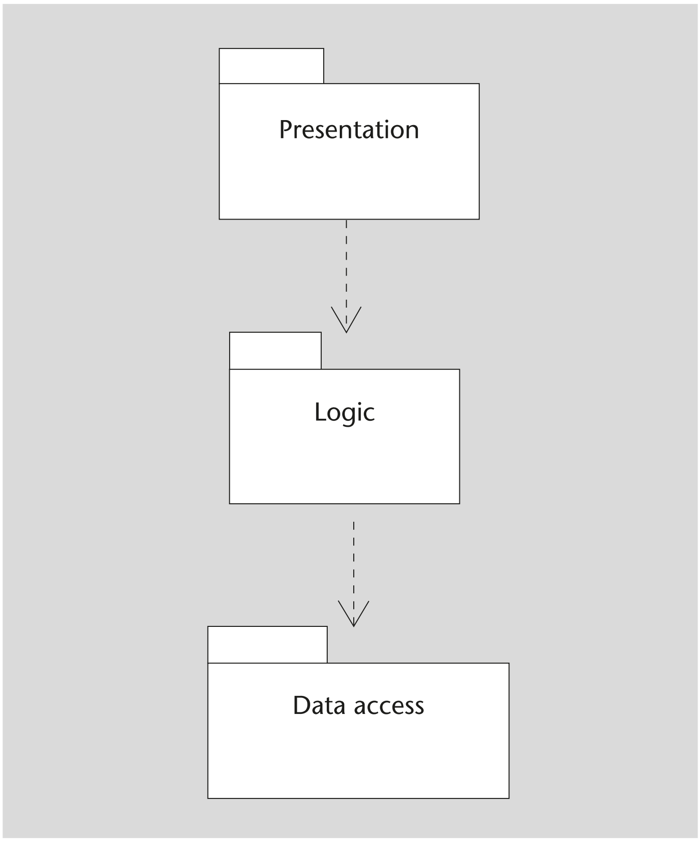
- Graphical interface elements (buttons, menus, etc.).
- Input/output operations (writing to disk, network messages, etc.).
- Queries and updates in a database.
- Software classes that represent domain concepts (employees, salary, etc.).

Solution

Organising the large-scale logic structure of the system into separate layers with different responsibilities in such a way that lower layers are general low-level services and higher layers are more application specific.

Collaboration and coupling takes place from the highest to the lowest layers. A layer of level N will only be coupled to the layer of level N-1. In this way, each class will have a more coherent and simpler vision of the system.

A very typical application of this pattern is the three-tier architecture:



- Presentation: this level of abstraction is the one of the graphical user interface elements (screens, buttons, etc.) and its responsibilities are collecting user inputs, calling the logic layer and displaying the results to users.

- Logic: this level of abstraction is the one of the concepts of the system and its responsibilities are those of implementing business rules. Logic layer classes see the system as a set of entity classes (salary, employee, etc.) that contain the information that manages the system and a series of use cases or system operations that determine its functionality.

- Data access: this level of abstraction is the one of the platform (operating system, files, etc.) and its responsibilities are isolating as far as possible the logic layer from the platform elements.

In our example, to calculate a salary, the different layers will intervene as follows:

1) Presentation layer: we will only worry about knowing that a certain button has been pressed and that, as a result, a certain screen must be displayed with some data that we will receive from the domain layer.

2) Logic layer: we will receive a call to an operation of a class that must know how the calculation is made from its parameters. We do not have to worry about which buttons have been pressed or what screens are displayed, nor on what physical files is the information we have to use.

3) Data access layer: the vision that these classes have of the system consists of a database, a series of auxiliary files and, perhaps, a remote server to which we send data through the Network. They don't know anything about salarys, or employees, nor graphical user interfaces and they only have to respond to read/write requests on disk, queries in the database, etc.

Consequences

- It reduces the system complexity, since each component works at a single level of abstraction clearly defined by the layer that contains it.

- It enables the exchange of an entire layer for another in a system with minimal impact.

- It reduces efficiency due to the potential increase in indirectness levels between objects when executing a task. In addition, it may be the case that a layer performs tasks that the upper layers do not need.

- It facilitates reusing entire layers thanks to the definition of clear interfaces between them. This reuse may therefore be of a greater volume of classes than the reuse of individual components.

4.1.1. Variants

The number and function of the layers can vary from one system to another and, in fact, these are two of the variables to consider when applying the pattern. For example, we can use two layers as in client/server systems.

4.2. Dependency injection

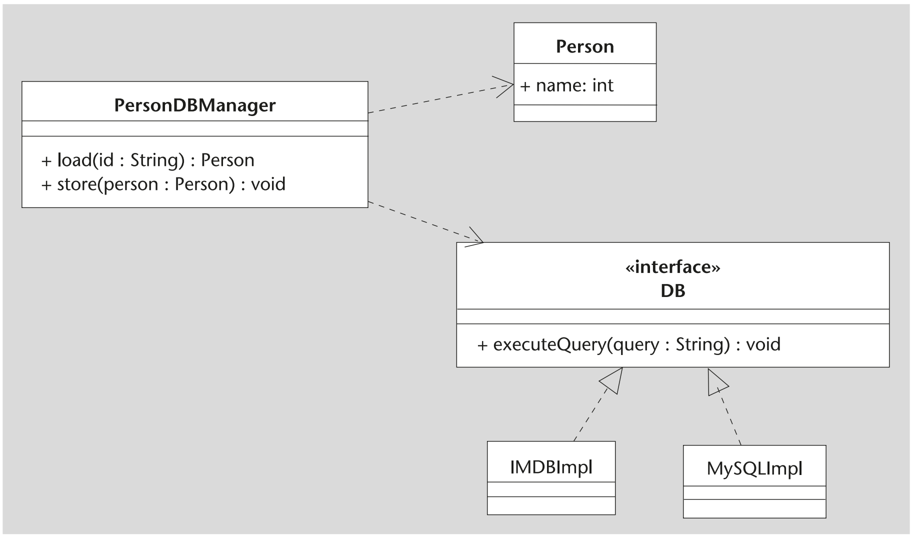
Context

In our system, we have defined a series of services (such as sending emails or a persistence service) for which we have (or may have in the future) several implementations.

Problem

We want to choose and change the implementation of each service without having to modify the code of our classes, which will not depend on any specific implementation, but on the service interface.

Let us suppose we are developing a system that must access a database. Some of the classes will need to use a database access component to perform their functions:


Person
+ name: int
PersonDBManager
+ load(id : String) : Person
+ store(person : Person) : void
«interface»
DB
+ executeQuery(query : String) : void
IMDBImpl
MySQLImpl

In this case, we have two implementations of the same interface. It could be the case that, in the development environment, we work with a database in memory to accelerate execution (IMDBImpl) and, in the production environment, with a database server (MySQLImpl). In this case, we need a mechanism that allows us to transparently choose the implementation that we will use in the class PersonDBManager.

Solution

Offering operations to be able to inject its implementation into the user classes of a service. This injection will be performed by a third class that will typically be part of a specialised framework.

This class or framework must allow us to transparently configure which implementation we want to use for our classes (for example, via a configuration file).

In our case, this would be the result of applying dependency injection to the class PersonDBManager:

```
public void setDB(DB implementation) {
  this.db = implementation;
}
public void load(String id) {
  db.executeQuery(SELECT...");
}
```

Consequences

- The dependencies are explicit, as they appear as operations or attributes of the class Client. For example, setDB, in the class PersonDBManager makes the dependency of PersonDBManager towards DB explicit.

- The dependencies are mixed with the operations or attributes of the class. In the previous example, it is not clear whether DB is a property of the class PersonDB or a dependency.

- Reuse is facilitated by not relying on any specific dependency injection implementation (if we want to distribute our classes, we will have to not distribute the dependency injection service).

- It is easier to test our class in isolation, since we do not need to modify the dependency resolution mechanism, but simply configure the instance on which we will execute the unit testing with the implementations better suited to do the test. For example, to test an instance of PersonDBManager we only have to provide a DB implementation by calling the setDB operation.

Complementary bibliography

More information on software classes unit testing can be found in [BECK].

- It is easy to identify each service, since all have a name (this name will be determined by the name of the operation we use to inject the service implementation).

- We must guarantee that every instance is correctly configured when it is used.

Variations

- Injection by constructor: dependencies are parameters of the constructor of our object. This ensures that it is impossible to have an instance that is not configured correctly, since it is configured when it is created. This solution is problematic in programming languages in which parameters have no name if we have more than one of the same type, since there is no way to differentiate the two services.

- Injecton by interface: an interface that our class must implement is created. The interface defines the operations that will be used to inject the dependencies.

Implementations

Some Java frameworks that facilitate the application of this pattern are PicoContainer (www.picocontainer.org) and Spring Framework (www.springframework.org).

4.3. Model-view-controller (MVC)

Context

We are developing a system with a graphical user interface.

Problem

We want to decouple the graphical interface of our system from the rest of the system.

In a layered architecture, we want to define the architecture of the presentation layer.

When designing our application graphical user interface classes, we must keep in mind that the interaction with the user is a very complex task and if we do not make a good responsibilities separation, maintaining this part of the application can easily become very hard. On the other hand, the user interface of a system is probably the part that will change the most.

Let us take as an example a form to edit user data that allows us to modify the name. Once the name has been changed, the user must press the Accept button.

Solution

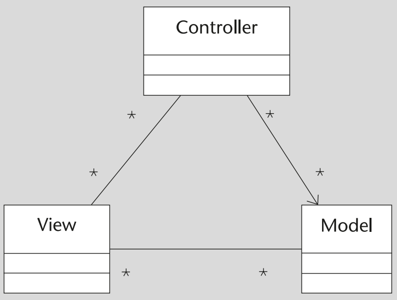
Dividing the system into three types of components: models, views and controllers.

Models encapsulate the state of the system. Its level of abstraction is the domain of the problem. It is where the so-called *business logic*[1] is implemented. They also notify the views of changes in the system status.

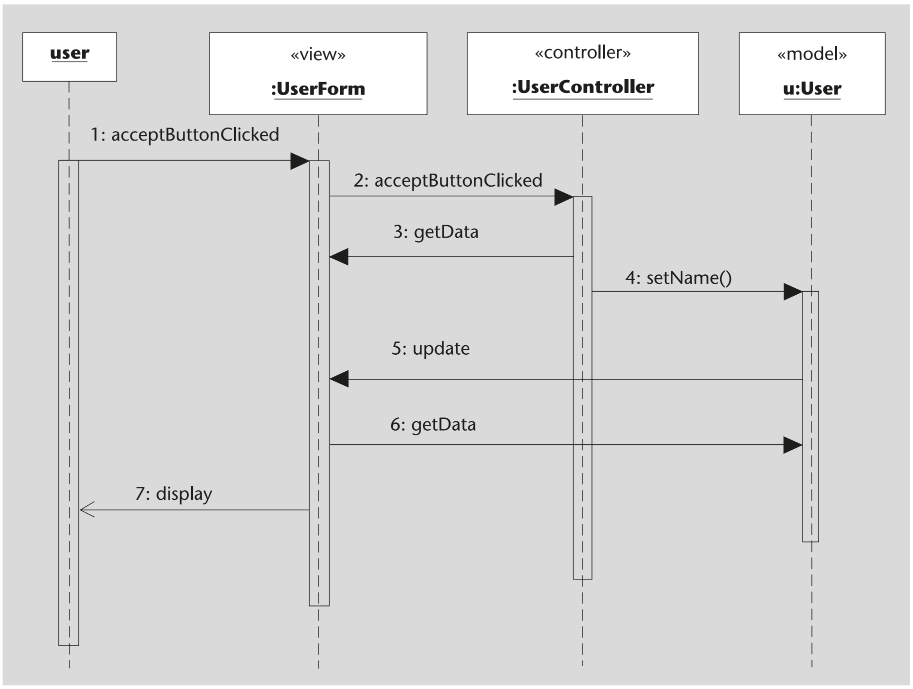
(1)System behavior in terms of the domain.

Views display data to users and collect the interactions (for example, a button on a screen has been pressed) to send them to the controllers. They query the model for the state of the system.

Controllers establish the correspondence between user actions and system events. They also decide which views are displayed to users:

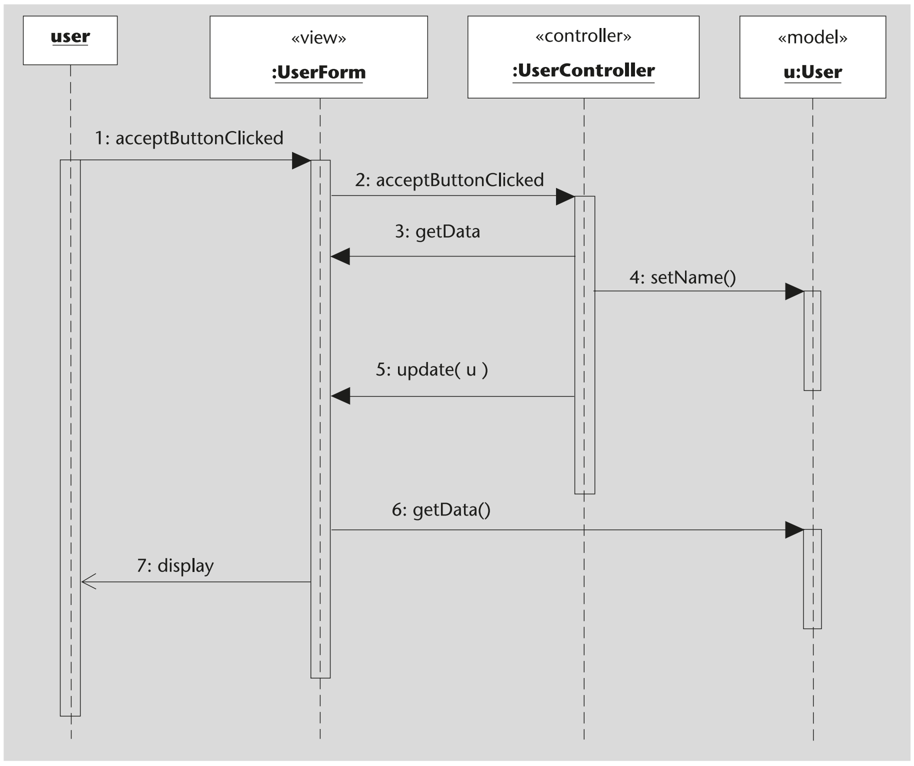


In our example, when the user presses the Accept button, the view will call the controller; this will translate the event into a system operation on the model (setName). As a result of its state change, the user will notify the views that must update:



In a layered architecture, the model will consist of the set of classes of the domain layer. However, since the domain layer cannot be coupled to the presentation layer, the models will not directly update the views. One way of solving this problem is to make the controllers, which know the model and are able to communicate with the views, responsible for notifying the views. In this way, the controllers, after modifying the model, notify the views so that they update.

In a layered architecture, therefore, our example would work as follows:



Consequences

- The cohesion of the classes is higher because all their responsibilities are related.

- We can support more than one display technology (for example, web interfaces and “rich client” interfaces).

- Developers can specialize in a specific domain, whether it is the application domain or the graphical interface development domain.

- We can update the display logic independently of the business logic (or update the business logic without having to modify the display logic).

- If we want the views to be automatically updated, every time there is a change in the model we will have to establish a mechanism to notify these changes. That can easily create a dependency of the logic layer in the presentation layer, which is undesirable if we are using a layered architecture.

5. Responsibility assignment patterns

Responsibility assignment patterns are a special type of design pattern that are not applied to solve a particular design problem, but are applied, in general, to split responsibilities between the different classes in the class diagram. Thus, its application is usually prior to the application of the rest of the design patterns.

See also

More information on GRASP patterns can be found in [LAR] or in section "Responsibility assignment patterns" of the "Introduction to patterns" module.

To document these patterns, we will slightly modify the structure of the explanation, since they all solve the same problem: how to assign responsibilities to our classes so that our design respects the principles of quality design.

See also

Check the design principles catalogue in section "Design principles" of this module.

5.1. Controller

Context

Some events occur and we must decide who will be responsible for managing them.

Example

This pattern should not be confused with the Model-view-controller pattern seen in section "Model-view-controller (MVC)" of this module.

In a layered architecture, when the presentation layer detects that an event that must be notified to the logic layer has occurred (for example, the user wants to register a customer), we need to determine which class of the logic layer will receive this event to deal with.

Solution

Creating a class that we will call *Controller* and assigning the responsibility of managing the event to it.

There are different types of controller:

- Facade controller: it represents the global system or a subsystem. It contains as many operations as events may occur in the system or subsystem.

- Use case controller: it represents the use case in which the event occurs. It contains as many operations as events may occur throughout the execution of the use case.

- Session controller: it represents an actor's session. A session is an instance of a conversation between the actor and the system and it can have any duration. Events corresponding to all use cases in which the actor can intervene may occur.

- Transaction controller: it represents a single system event. The event parameters will be attributes of the class. It only contains an operation that we will call *execute*.

Example

In a web application, for example, the presentation layer will be in charge of interacting with users via their browser. The browser will interact with the system by sending HTTP requests that the system will respond to using the HTML that the user will see. The presentation layer will be responsible of translating HTTP requests to system events in terms of the logic layer. But then, who will be responsible in the logic layer of managing these events?

Let us suppose that in a use case to buy products from an ecommerce we have a query request of a product category:

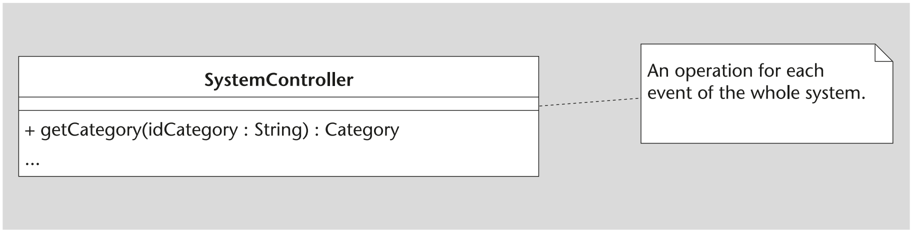
```
GET http://www.emusic.com/getCategory?idCategory=321
HTTP/1.1
```

The presentation layer will translate it into a system event that corresponds to an operation of the logic layer with the following signature:

```
public Category getCategory(String idCategory)
```

But which object of the logic layer will be responsible for handling this request? According to the Controller pattern, it will be a controller object that we can define at different levels:

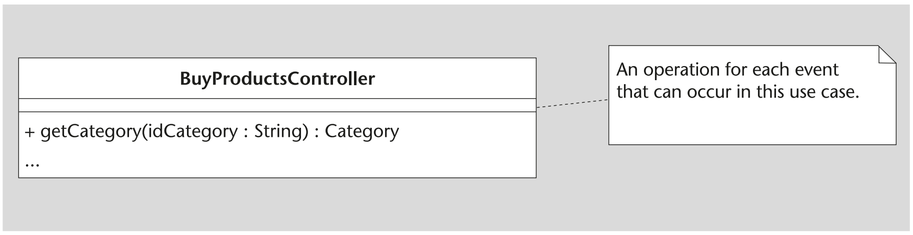
- Facade controller:


SystemController
+ getCategory(idCategory : String) : Category
...
An operation for each event of the whole system.

This type of controller has the advantage of being a simple and elegant solution. When we use it, we must choose a name that represents the whole system that will also provide an abstraction of the logic layer for other layers. In case of doubt, SystemController is always indicative enough.

The facade controller will be adecuate when there are not many different events or when we want to release the layer that is using them from switching and choosing the most appropriate controller.

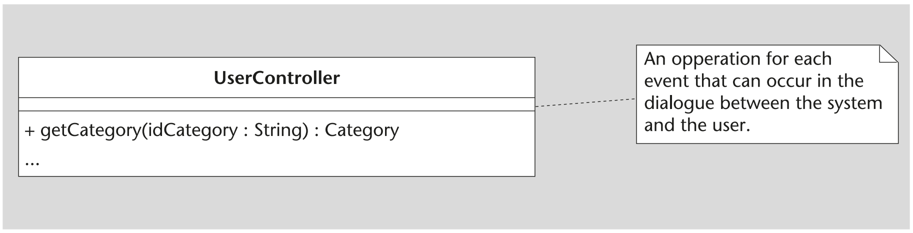
- Use case controller:


BuyProductsController
+ getCategory(idCategory : String) : Category
...
An operation for each event that can occur in this use case.

This strategy is interesting when there are many different events, since it allows us to group them. In these cases, the facade controller becomes too complicated and leads us to designs with low cohesion and/or high coupling.

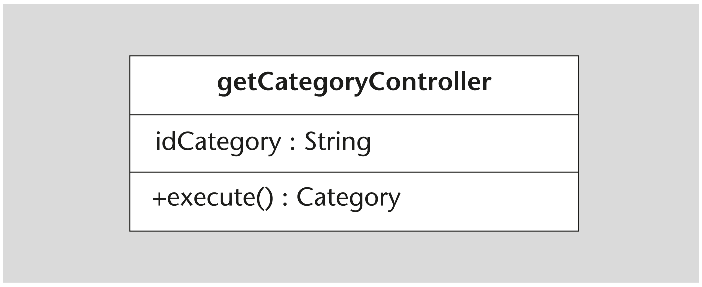
On the other hand, it has the advantage of allowing the use case state to be controlled so that it can manage and reason about which events are valid at each moment of the use case. Finally, the analysis based on the stereotypes of boundary classes, control and entity, translates into this type of controller in a very direct way facilitating the design task.

- Session controller:


UserController
+ getCategory(idCategory : String) : Category
...
An opperation for each event that can occur in the dialogue between the system and the user.

These controllers share much of the advantages of use case controllers. Normally, the number of controllers following this strategy is lower and, therefore, it may be adequate in systems with an medium number of events.

- Transaction controller:



Finally, transaction controllers have the advantage of allowing the handling each event as an object. In addition, they allow us to easily extend the system by adding new classes for the new events that we may want to add.

The four options proposed by the controller pattern have their advantages and disadvantages. It is necessary to choose a modality that does not generate an excessive number of simple controllers, but that does not create overloaded and difficult to understand and manage controllers.

Corollary

An important conclusion of this design pattern is that we must avoid at all costs that the presentation layer and its components (such as button, window, text area for desktop applications or *servlets* and JSP for web applications, etc.) are responsible for managing system events. That is, system events should be managed by the logic layer.

5.2. Expert

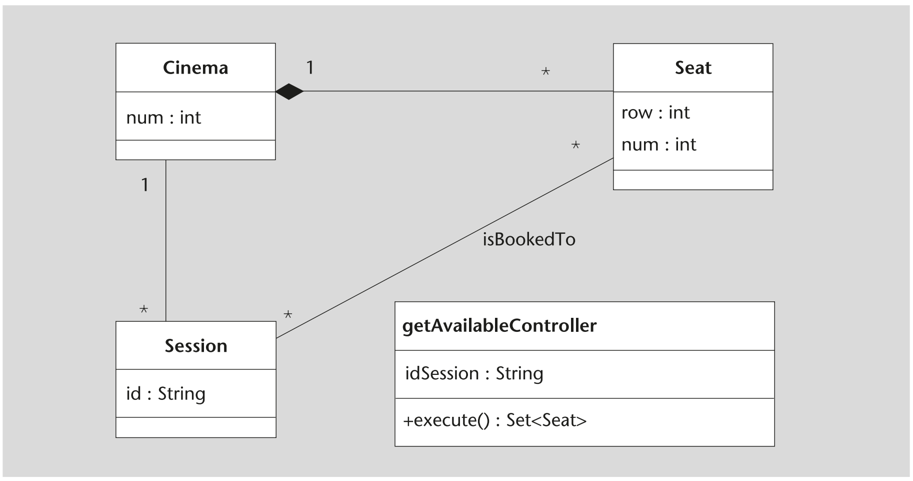
Context

Some treatment or calculation on an information must be made.

Solution

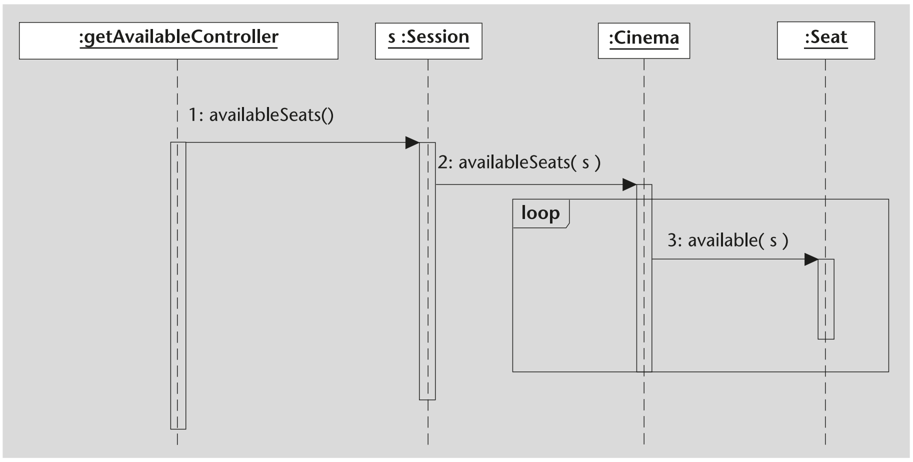
Assigning the responsibility of making the treatment or calculation to the class that contains the necessary information. We will call this class Expert (regarding this treatment or calculation).

For example, let us suppose we are developing a seat booking system for a cinema. We want to design the treatment related to query the list of available seats for a session:



To design the operation execute() of the controller, we will start from the session identifier. Therefore, the first expert to decide is the one that knows the identifier of a session: the same session. Once the session is determined, we will have to query the list of seats in the cinema where the session is held. The expert regarding this list is the cinema.

Finally, it will be necessary to know, for each seat on the list, whether or not it is booked for that session. In this case, the expert is the seat. The resulting design, then, is as follows:



5.3. Pure fabrication

Context

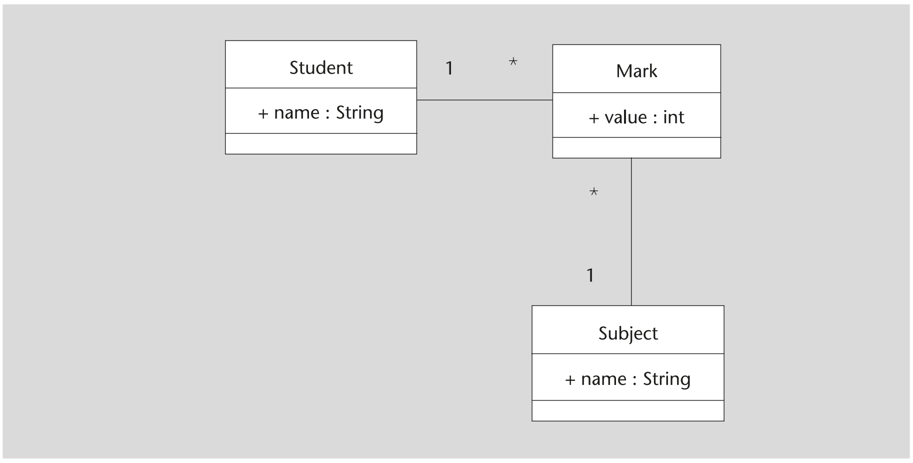
Some treatment or calculation independent of the domain logic must be made.

Solution

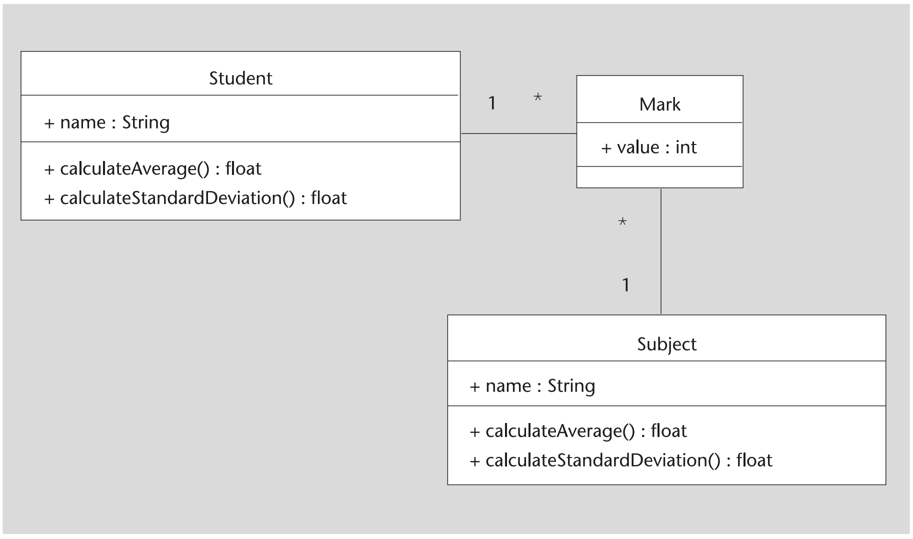
> Assigning a set of highly cohesive responsibilities to an artificial class that does not represent a concept of problem domain; a class made up to withstand high cohesion, low coupling and reuse.

It is called *Pure fabrication* because the new class is a *fabrication* of the imagination (it does not exist in the real world) and, since its responsibilities are very cohesive, its design is very clean, or *pure*.

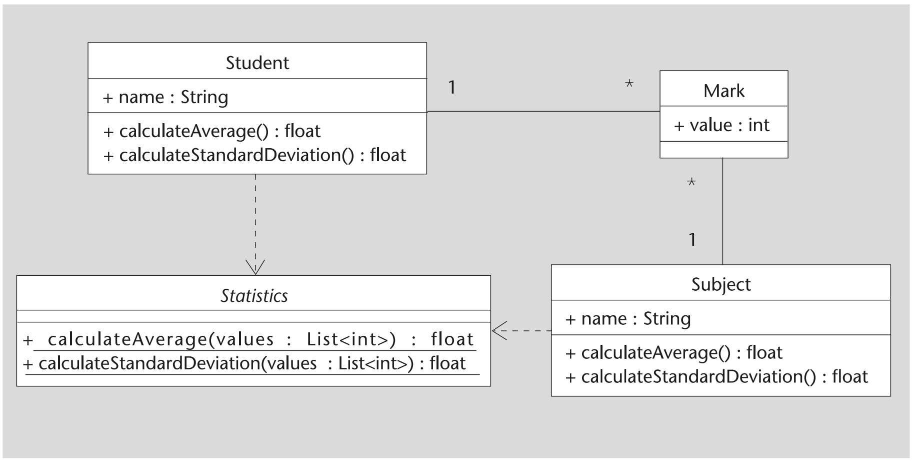
For example, let us suppose that the system we are developing must calculate the average and standard deviation of a student's grades, and also those of a subject starting from the following conceptual model:



We can assign the responsibility of making the calculation of the average and standard deviation of the student's grades (according to the Expert pattern) to the class Student, and also to the class Subject of the subject, but this solution will not allow us to easily reuse the calculation:


Student
+ name : String
+ calculateAverage() : float
+ calculateStandardDeviation() : float
1
*
Mark
+ value : int
*
1
Subject
+ name : String
+ calculateAverage() : float
+ calculateStandardDeviation() : float

Applying the Pure fabrication pattern, we can assign the responsibility of calculating the average and standard deviation so that we can reuse it and we only have to implement it once:


Student
+ name : String
+ calculateAverage() : float
+ calculateStandardDeviation() : float
1
*
Mark
+ value : int
*
1
Statistics
+ calculateAverage(values : List<int>) : float
+ calculateStandardDeviation(values : List<int>) : float
Subject
+ name : String
+ calculateAverage() : float
+ calculateStandardDeviation() : float

5.4. Creator

Context

We must create instances of a class.

Solution

Assigning to class B the responsibility of creating an instance of class A if one or more of the following cases are met:

B *adds* objects from A.

B *contains* objects from A.

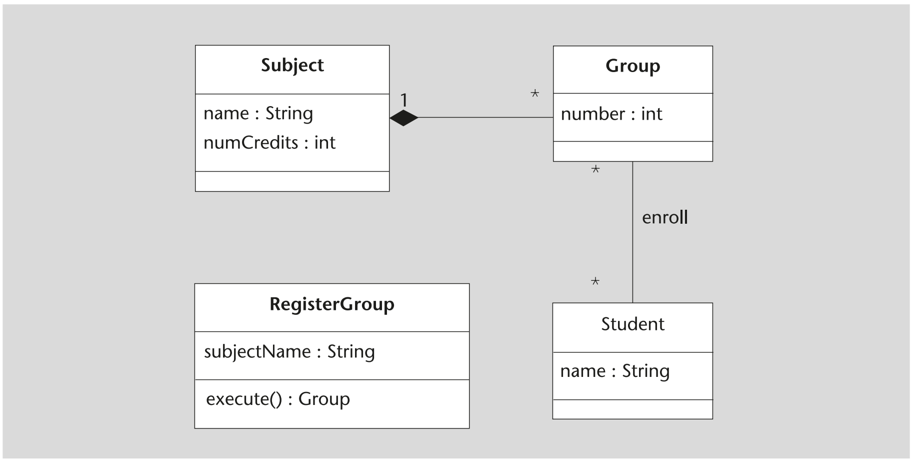
B *registers* objects from A.

B *closely uses* objects from A.

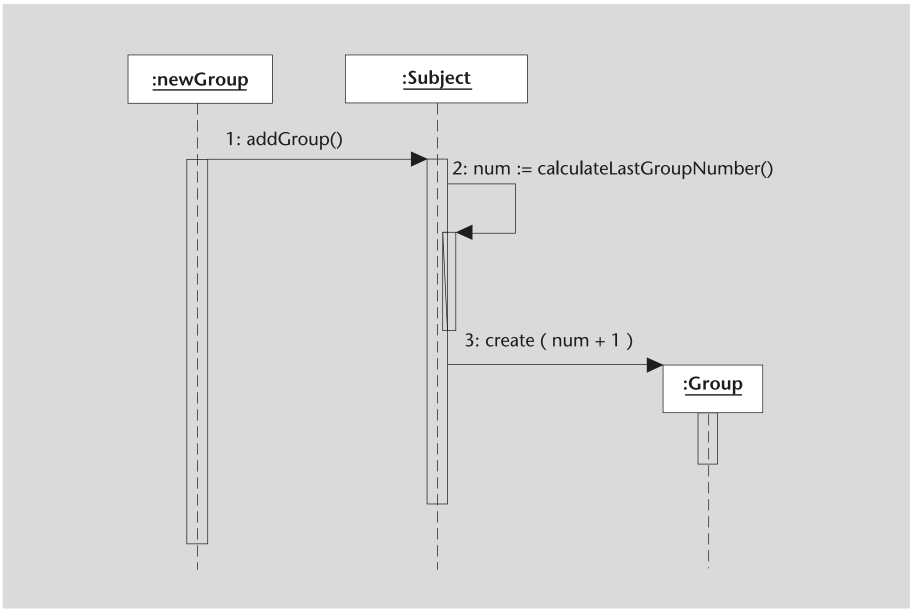
B *possesses the initialisation data* that will be passed to an object of A when it is created (and, therefore, B is an Expert with regard to the creation of A).

If more than one option can be applied, it is advisable to opt for a class B that *adds* or *contains* objects from A.

For example, let us suppose we are developing a student enrollment support system for a university:



Who will be responsible for creating a new group for a subject? According to the creator pattern, since subject adds group instances, it will be responsible for the creation of the group instances. Therefore, the solution will be as follows:



6. Design patterns

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

See also

More information on design patterns can be found in section "Design patterns" of the "Introduction to patterns" module.

6.1. State

Context

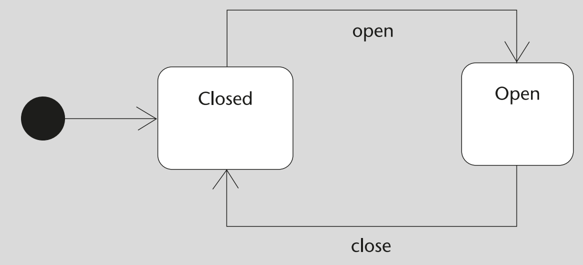
The objects of a class vary their behavior depending on their state.

We have operations with large conditional blocks that depend on the state of the object that invokes the operation. State combinations are common in more than one block and/or operation.

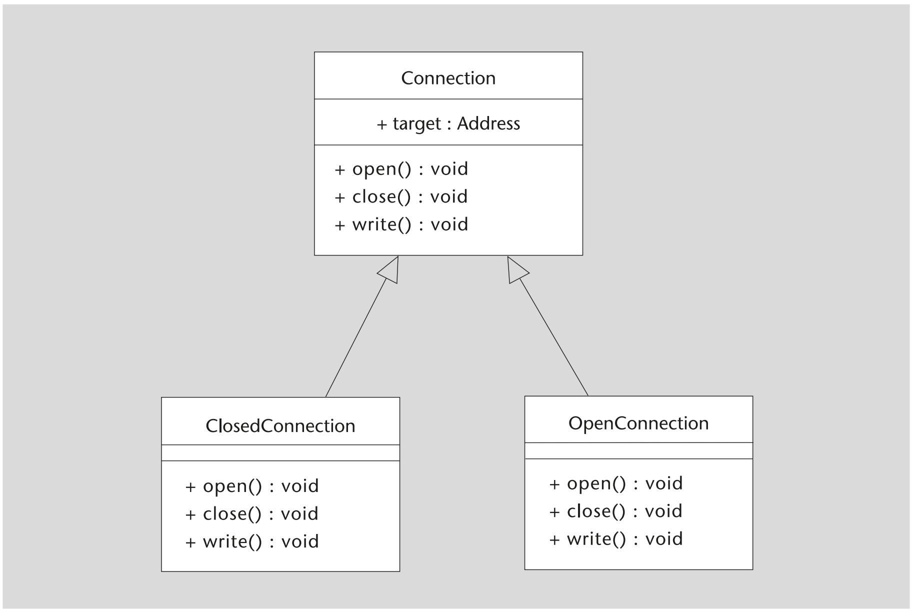
Problem

We want to take advantage of polymorphism to choose the appropriate behavior at all times without using conditionals, but we cannot use dynamic generalisation to specify a different subclass for each state.

For example, let us imagine that during the development of our system, we have encountered a class called *Connection* that represents a connection to a remote server. Its state diagram is as follows:



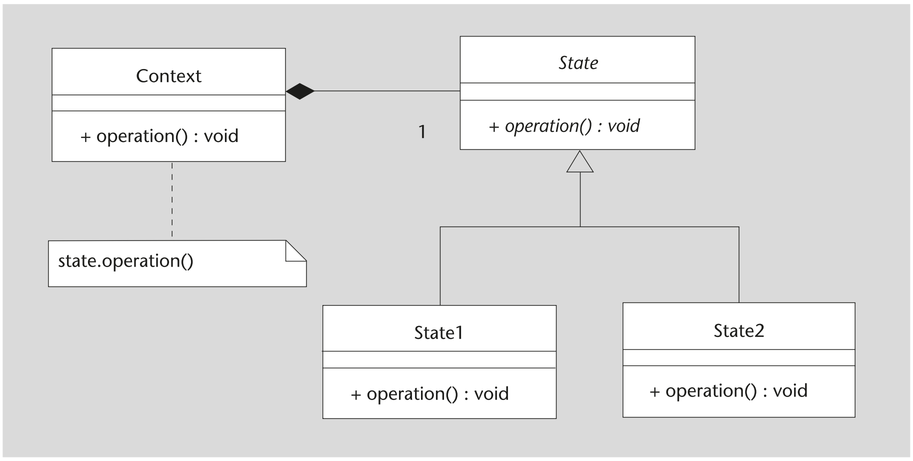
This class has three operations: open, close and write. The behavior will be different depending on the state of the connection: for example, when the close operation is invoked on an open connection, an end-of-connection message is sent to the server and the state of the connection (which becomes closed) is modified, while, when it is closed, calling the close operation causes an error. We can think of a possible design that would solve our problem:



This solution, however, is not valid for us because the language we will use does not support dynamic generalisation (that is, it does not allow a connection instance to change subclass during system execution), so we need another way to solve the problem.

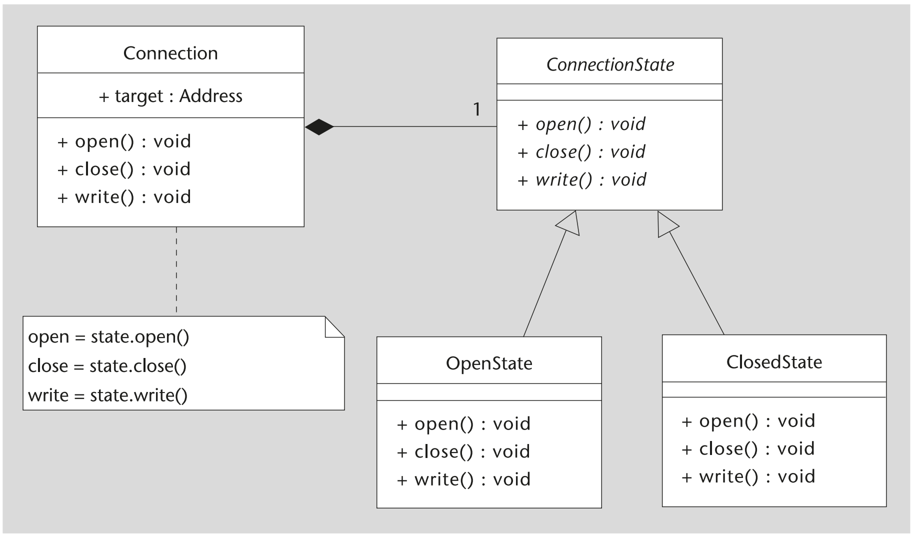
Solution

Splitting the part of the class that varies depending on the state of the rest and handling each state as a class that will implement the expected behavior when the object is in that state.



The class Context delegates on the class State operations that vary depending on the state, and if necessary it is added as a parameter.

Applied to our case, we would obtain the following result:



Now, to change connection state, instead of having to change the subclass, connection instances only have to modify the specific instance of ConnectionState associated to them.

Consequences

- Design extension is facilitated by adding new states.

- It is necessary to decide who is responsible for making the transitions between states (the context or the specific state). This decision will imply a coupling between this class and the specific states (which will affect design extensibility).

- State transitions become explicit.

- State objects can be shared between different contexts.

- It is necessary to determine when State objects are created and destroyed (their creation and elimination may not be linked to the creation and destruction of the context).

Variants

In the solution, we have assumed that the association between the context and the state is one to one (that is, for each context object we have a state object). By modifying this cardinality, we reach two interesting situations: a context object that is in more than one state at the same time and/or a state object usefull for more than one context.

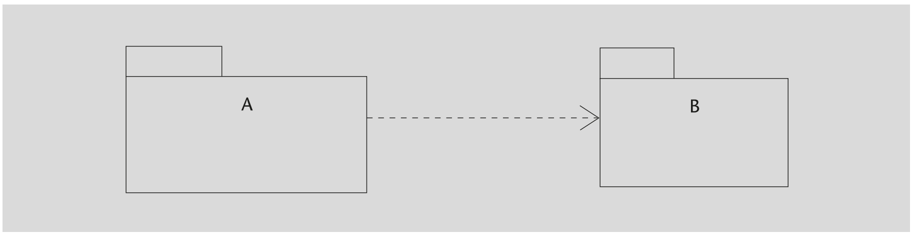
To have a context object associated to more than one satate at the same time, we will have to modify the cardinality of the association next to the state by adding an asterisk (*). That can be useful when the differences between states are only relative to the attributes and associations that can be had, but it will be problematic if there are operations that are implemented differently in two of the simultaneous states.

To have a state object associated with more than one context, we have to modify the cardinality of the association next to the context by adding an asterisk (*). In this case, we must put in the context object all the attributes of all the states, since if they were in the state object its value would be shared by all contexts. This solution is especially interesting if state objects are hard to create and/or keep in memory.

6.2. Facade

Context

Our system is structured in subsystems or layers. A subsystem (or layer) A uses another subsystem (or layer) B:



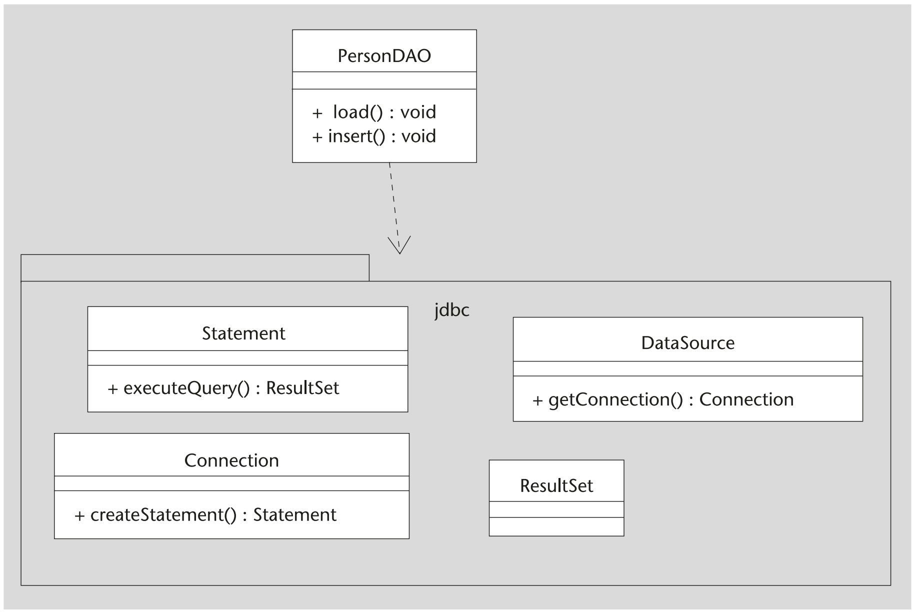
Problem

We want to reduce coupling between two subsystems.

We want to access a simplified version of a subsystem.

We want to offer a single entry point to a subsystem.

For example, let us suppose the system we are developing has a class called *PersonDAO* that contains the operations to access the people database. This class must access the programming interfaces of the database, so that it is coupled with all the classes of the database subsystem:


PersonDAO
+ load() : void
+ insert() : void
jdbc
Statement
+ executeQuery() : ResultSet
DataSource
+ getConnection() : Connection
Connection
+ createStatement() : Statement
ResultSet

This coupling will be repeated for all classes that access the database. For example, if we add a class CustomerDAO to access customer data, it will also be coupled with the entire database subsystem.

How can we prevent so many classes from depending on the database subsystem?

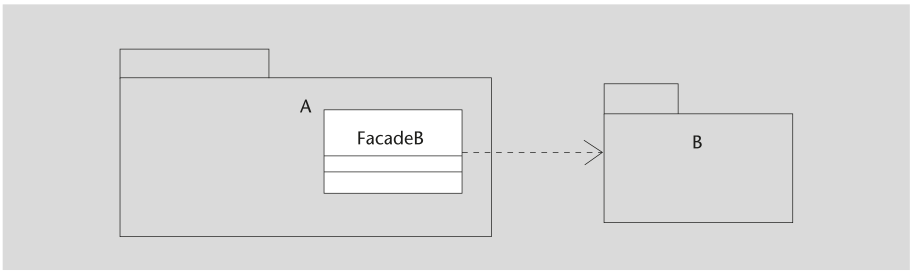
Solution

Creating a new class Facade that represents the entire subsystem or layer accessed.

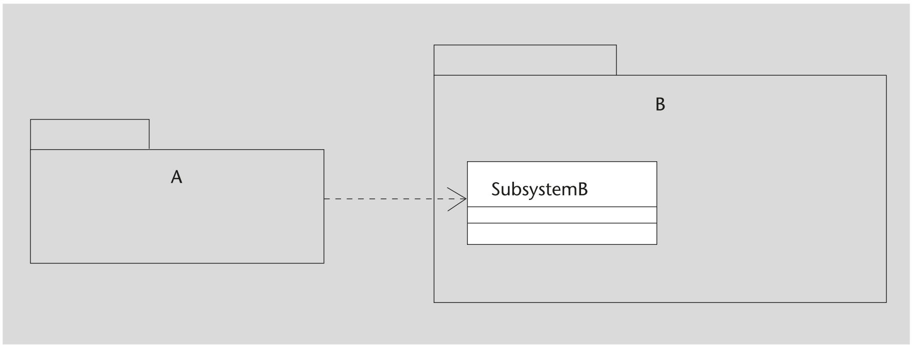
The Facade concentrates the dependence on a single point so that changes in the subsystem used will have minimal impact on the classes of the client subsystem. In addition, it will concentrate part of the complexity derived from the use of the subsystem that represents and will hide the internal class structure.

The Facade may belong to one or the other subsystem depending on our needs and possibilities.

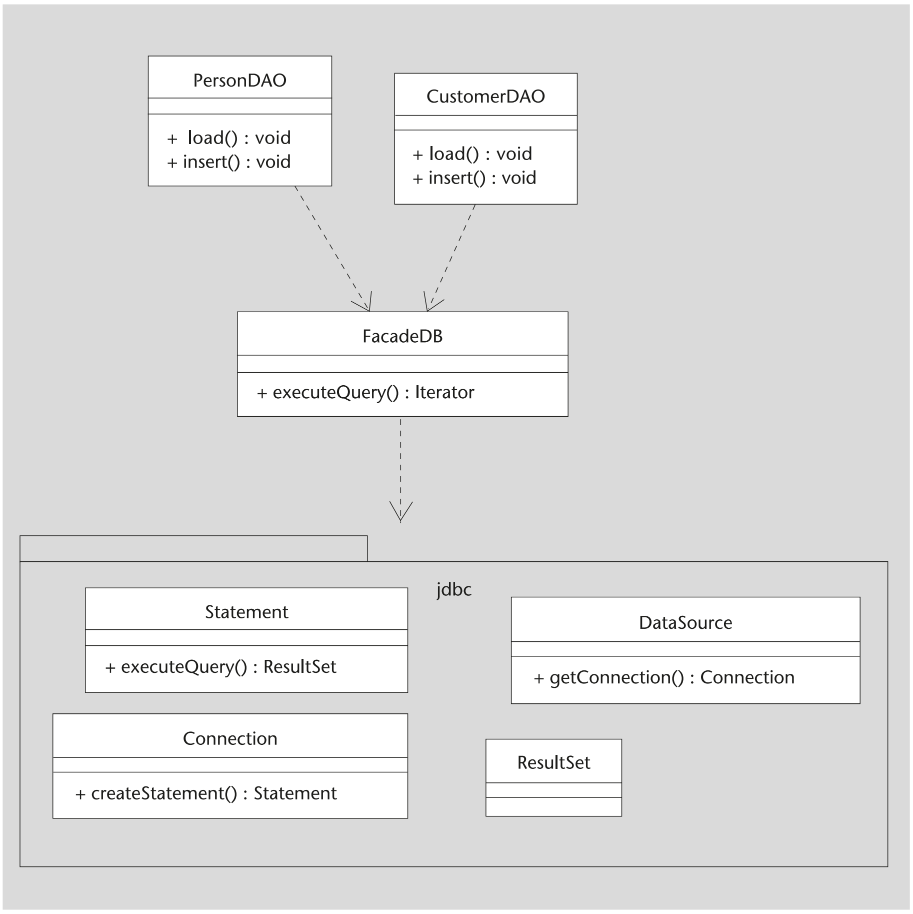
If what we want is to reduce coupling between A and B or accessing a simplified version of B, we will add a Facade to the client subsystem (A) that concentrates the coupling and/or simplifies our vision of B:



But if we want to offer a single entry point to B, we will add a Facade to B that will be the only public class of the subsystem (the rest of the classes may have package visibility) in order to make sure that there is no other way of entry:



In our example, we want to reduce coupling with regard to the database subsystem and we cannot modify it because it is a subsystem developed by another company. Therefore, what we will do is creating a facade within the client subsystem:



As we can see, in the new design, dependencies are more concentrated. Any change in the database subsystem will affect only the class FacadeDB.

Consequences

- Dependencies are concentrated in a single class and, therefore, if there is a change in the subsystem used, it will only affect the facade and will not spread to the rest of the client subsystem classes.
- The solution is more complex. If the interface offered by subsystem B is stable and simple enough, the facade might not add enough value to justify its use.

6.3. Iterator

Context

An aggregate object wants to grant access to its elements.

An object wants to grant access to the objects it has associated.

Problem

We do not want to expose the data structure that we internally use to store the references to associated or aggregated objects (the internal representation of the aggregation or association) to those who access it.

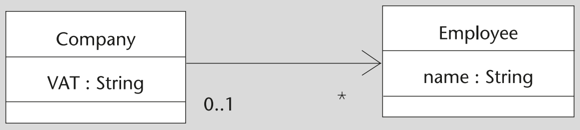
In this way we want to avoid coupling of our client classes towards this structure (a fact that would make difficult changing it for another), as well as possible manipulations of this structure without our control.

We want to provide different strategies to access the elements (for example, with different sorting criteria).

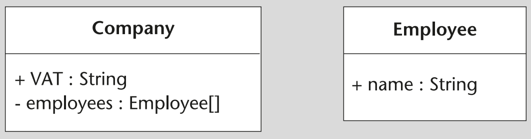
We do not want the aggregate class to be responsible for keeping track of what elements it has already shown to each client class that can access it.

We do not want to assign the responsibility of knowing how the internal data structure is traversed to our client classes.

For example, let us suppose the system we are developing has the following classes:



Since a company must maintain a set of references to employees working in it, we decided to use an array to store these references:



We want to add an operation to Company that allows us to consult all the company's employees. If we return the array directly, we will be exposing the internal structure of the class Company (with the problems that this entails).

Solution

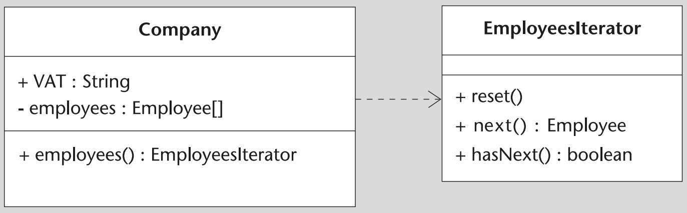
Creating a class Iterator that will know the internal structure, how it must be traversed and how it must be changed (for example, notifying it to the class) and that will encapsulate this knowledge.

We can associate an instance of the iterator with each client so that it is the iterator who has the responsibility of knowing which elements it has shown and which not to each client.

The class Iterator will provide an interface that will not depend on the internal representation with generic operations to traverse the elements of the structure:

- reset() Lets us restart traversing.
- next() Advances to the next item and returns it.
- hasNext() Tells us whether there are still elements to traverse.
- We can also add further operations (*remove*, *last*, etc.) according to the specific needs of each case.

In our case, the solution applying the Iterator pattern would be the following:



In this case, if we want to traverse the employees of a company we would do:

```
EmployeesIterator employees = company.employees();
while(employees.hasNext()) {
  currentEmployee = employees.next();
}
```

A possible implementation of EmployeesIterator would be the following:

```
public class EmployeesIterator {
  Employee[] data;
  int currentPosition;
  EmployeesIterator(Employee[] employeesCompany) {
    data = employeesCompany;
    currentPosition = -1;
  }
  public void reset() {
    currentPosition = -1;
  }
  public boolean hasNext() {
    return currentPosition + 1 < data.length;
  }
  public Employee next() {
    currentPosition++;
    return data[currentPosition];
  }
}
```

Consequences

- We can modify the internal representation without having to modify the client classes.
- We can vary the strategy used to access the elements (for example, modifying the sorting criteria).
- We can restrict the ability to modify aggregate elements. If we expose the internal representation, clients could modify it without the aggregate class noticing. An Iterator could notify the aggregated class.
- The interface of the aggregate object is simplified because of the reduction of responsibilities (it delegates them to the Iterator).

6.4. Factory method

Context

An object must create objects of another class.

Problem

The object that creates other objects cannot anticipate their class.

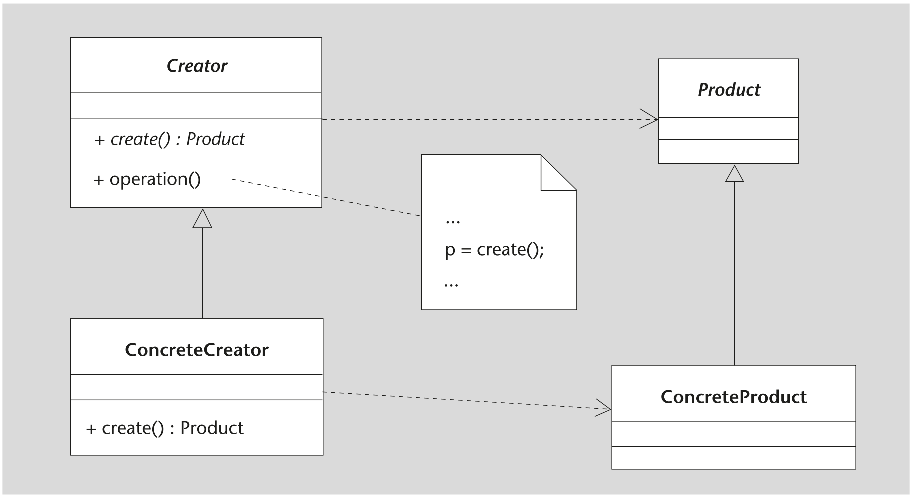
We want to delegate the responsibility of specifying the class of the objects that must be created in our subclasses.

A class delegates a responsibility to one or more helper classes and we want to isolate the responsibility of deciding what kind of help we will use.

Let us suppose, for example, that we are developing a system that must notify certain events to users. The class in charge of managing notifications must create a message and send it, but we want to delegate in the subclasses the selection of the type of message to send (for example, an email or a MMS message to a mobile phone).

Solution

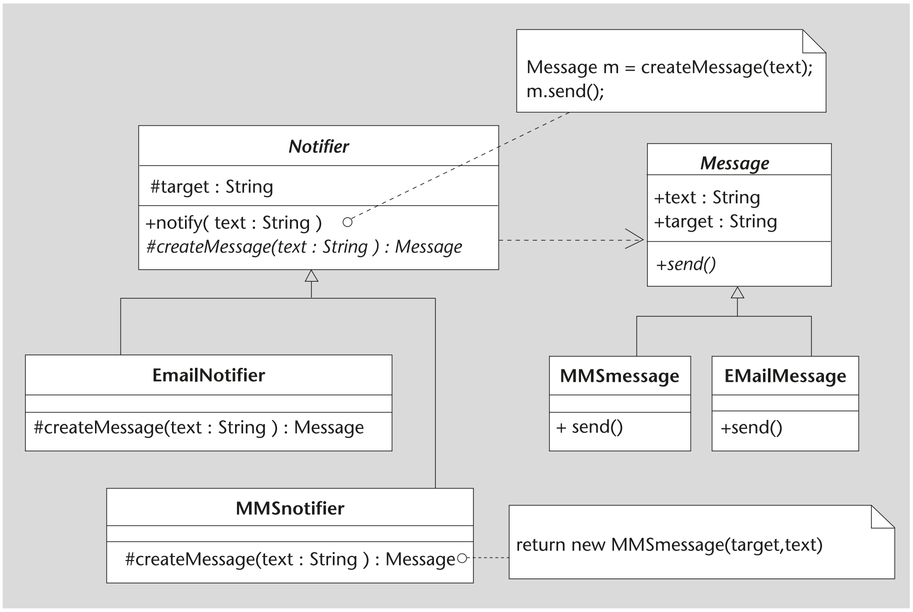
Defining, in the abstract class Creator (the one that creates the objects), a "create" method in charge of the creation. We will call this method *factory method*. Providing, to each subclass of Creator, an implementation of the factory method that creates a specific type of object.



In our example, we will apply the factory method, making the subclass decide which type of message will be created. In this way, users who work with a notifier of the type EmailNotifier will receive the messages by email and those who use a MMSnotifier will receive them on their mobile phone. In addition, if we want to add new notification mechanisms to our system, we only have to add new subclasses of Notifier and Message, thus respecting the open-closed principle:

See also

More information on the open-closed design principle can be seen in section "Open-closed principle (OCP)" of this module.


Message m = createMessage(text);
m.send();
Notifier
#target : String
+notify(text : String)
#createMessage(text : String) : Message
Message
+text : String
+target : String
+send()
EmailNotifier
#createMessage(text : String) : Message
MMSmessage
+ send()
EMailMessage
+send()
MMSnotifier
#createMessage(text : String) : Message
return new MMSmessage(target,text)

6.5. Template method

Context

We have an algorithm for which we want some subclasses to be able to redefine some of its parts.

Problem

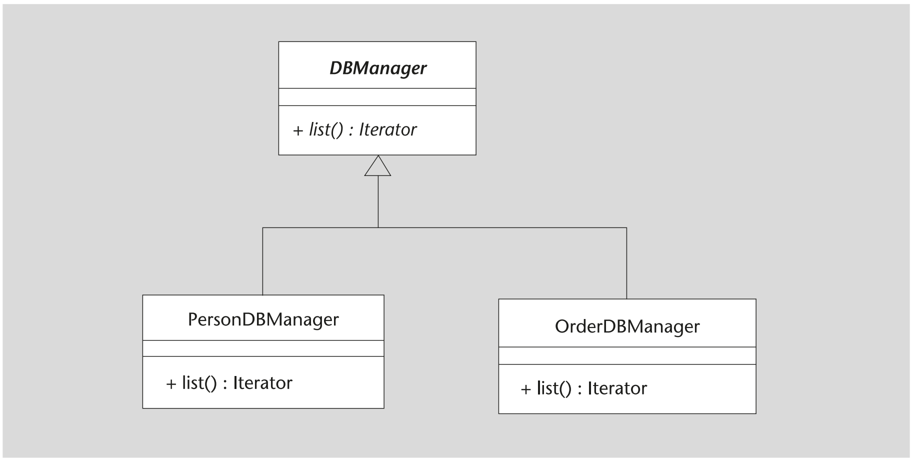
We want the subclasses to redefine only the part that is specific to them and avoid having to rewrite the common parts, which would violate the non-repetition principle.

See also

More information on the non-repetition principle can be seen in section "Don't repeat yourself (DRY)" of this module.

We want to restrict the set of possible modifications by setting a general structure and some extension points.

For example, let us suppose that in the system we are developing we have a series of classes that manage the access to the database:



The implementations of list() in PersonDBManager and OrderDBManager follow the same general algorithm:

1) To obtain a connection to the database.

2) To build a query.

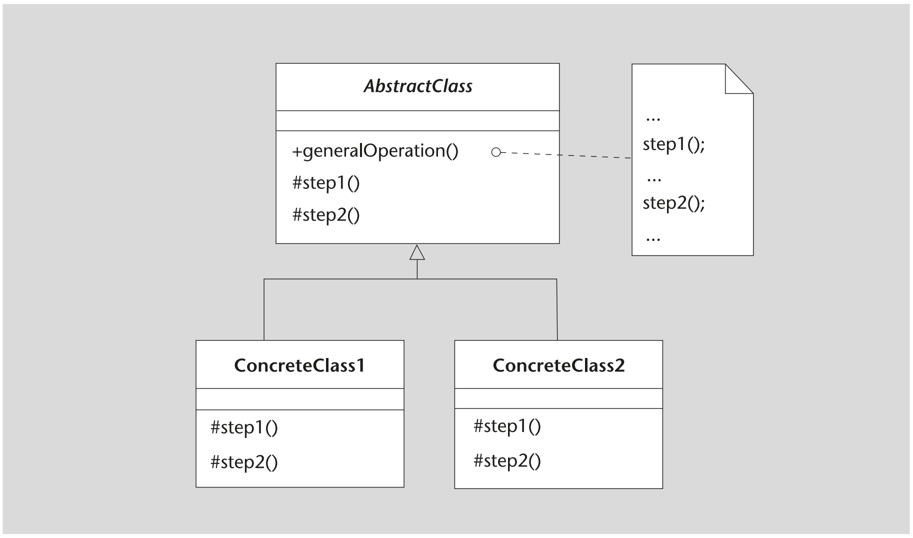
3) To execute the query.

4) To handle the result.

5) To close the connection to the database.

Solution

Defining an abstract class that implements the general algorithm by invoking abstract operations that subclasses can redefine.



In our example, we will define a list() operation that will implement the general algorithm. The specific parts of each subclass will be left to abstract operations that the subclasses will have to implement:

```
public abstract class DBManager {
  public Iterator list() {
    Connection c = getConnection();
    String query = buildQuery();
    Result r = c.execute(query);
    List result = handleResult(r);
    c.close();
    return result.iterator();
  }
  protected abstract String getQuery();
  protected abstract List handleResult(Result r);
  (...)
}
```

In this way, each subclass must define its own parts of the algorithm and will be able to redefine other parts of the algorithm if the superclass allows it.

Consequences

- This pattern is widely used to improve the possibilities of code reuse, since it enables to reuse not only whole functionalities, but parts of them.

- The superclass may offer default implementations of some steps of the algorithm. Therefore, we must distinguish between operations that may be redefined from those that must to be redefined no matter what (and from the rest of the operations used); in many languages, this can be done via abstract operations.

- In operations with default implementations, the subclass can extend this default implementation instead of completely replacing it.

6.6. Observer

Context

A change in an object implies changes in other objects.

Problem

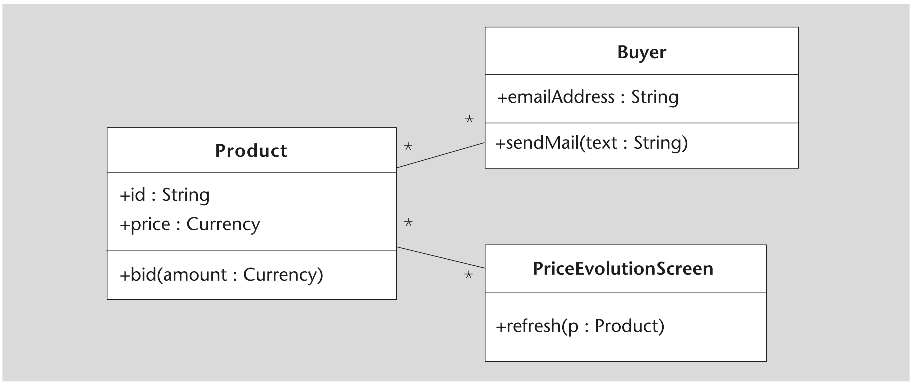
We want to avoid coupling between the class of the object in which a change occurs and that of the objects that have to receive the notification.

We want our system to be open to extension to allow new objects to be notified without having to modify the class of the object in which the change occurs.

See also

More information on the open-closed design principle can be seen in section "Open-closed principle (OCP)" of this module.

For example, let us suppose we are developing an electronic auction system and we have the following classes:

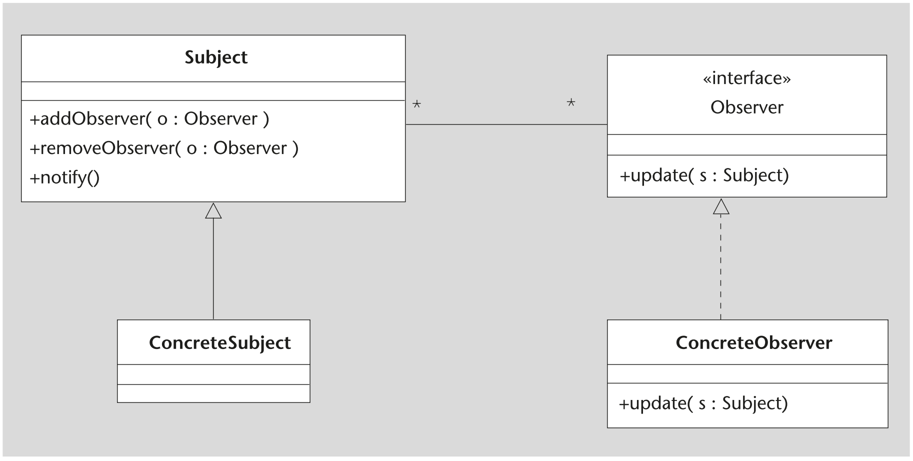


Each time a user bids for a product, the system must notify this change to the buyers by sending an email and refreshing some screens that display the latest bids that have occurred in the system.

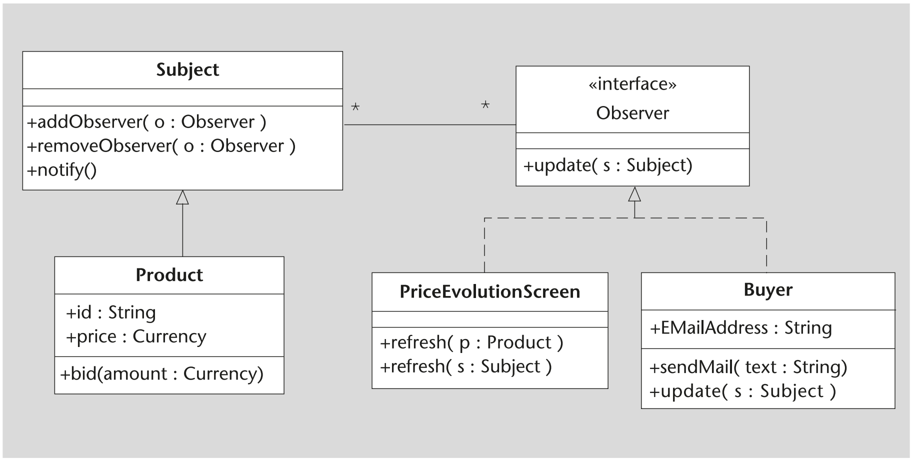
We want to prevent the class Product from being coupled with regard to the rest of the classes.

Solution

Creating a new abstraction for the notification mechanism and using inheritance and polymorphism to reduce coupling.



When there is a change in the state of the concrete subject, it invokes the operation notify() of its superclass which will take care of traversing the list of observers and invoking, for each one, the operation update(). In the case of auctions, the result of applying the Observer pattern would be as follows:



Thus, the implementation of the bid operation would be as follows:

```
public void bid(Currency amount) {
   this.price = amount;
   notify();
 }
```

The implementation of the subject will be independent of the concrete classes:

```
public void notify() {
   forEach(o: Observer in observers) {
     o.update(this);
   }
}
```

Finally, each observer will implement the operation update() in a different way:

```
  public class PriceEvolutionScreen  {
    public void update(Subject s) {
      refresh((Product)s);
    }
    (...)
  }
  public class Buyer {
    public void update (Subject s) {
        String message = generateMessage((Product)s);
        sendMail(message);
    }
    (...)
  }
```

Consequences

- Coupling occurs at a higher level of abstraction. The specific subject is only coupled with regard to a very simple interface.
- The specific subject does not have to worry about which or how many objects are interested in the notification. This allows us to dynamically add and remove observers.
- If the observers can cause changes in the specific subject that cause new updates, we will have to handle the case in a special way to avoid an infinite loop of updates.
- The notification protocol does not tell us what has exactly changed.

Variations

In the proposed solution, the subject sends itself as a parameter of the operation notify(). A possible variation would be sending other information with the notification (for example, the specific state changes that have occurred).

Another option is for the subject to not send any information and for the observer to be responsible for requesting the new state to all the subjects that it observes. This model is called *pull model* as opposed to the *push model*, which we have seen before.

6.7. Command

Context

We want to handle calls to operations as objects, for example, for the following:

- To be able to undo the operations.
- To execute them in another environment (in another process or node of the network) or at another time.
- To parameterise an object (for example, what action a user menu item must execute).
- To have a history of changes in order to reapply them if the system crashes.
- To structure the system around high-level operations (for example, having a class for each system operation).

Problem

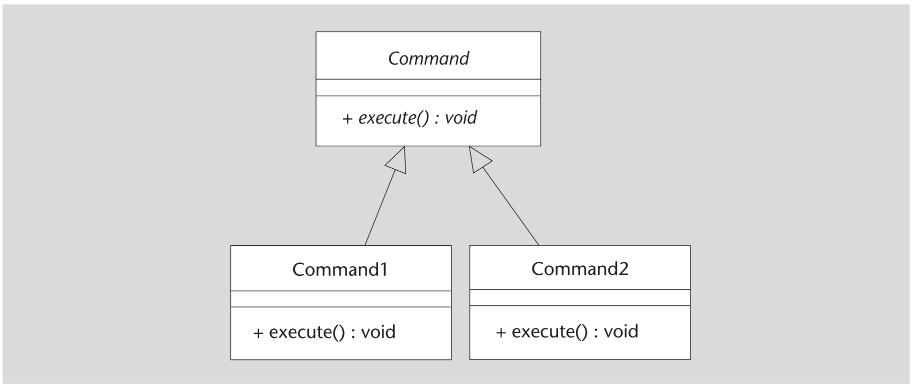
We need a mechanism able to handle an operation as an object because our programming language does not have this capability.

For example, we want the system we are developing to allow the user to undo the actions performed in the system (for example, if the user has modified some data, undoing the modification). If we could handle each user action as an object, we could save a list of the actions performed beeing able to undo them; but our language does not allow us to handle calls to operations as objects.

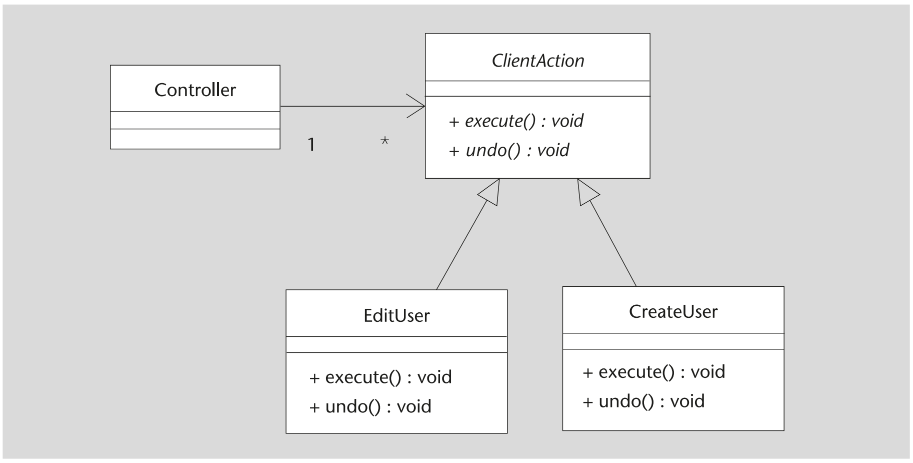
Solution

To create a class whose instances represent invocations of an operation.

We can convert the parameters and the result into attributes of the command. We can also establish an inheritance hierarchy between the different operations:



This solution can be applied to the previous example, handling each user action as an object and converting the set of actions into a list of objects that remember what they have done and are able to undo it:



Consequences

- The object that invokes the operation is decoupled with regard to the one that implements it.
- We can manipulate and extend orders like any other system class (for example, we can apply the Template method pattern).
- We can create command compositions that allow the joint execution of a series of commands (for example, to register macros).
- It is easy to extend our system by adding new operations without having to modify the existing ones.

6.8. Other design patterns

Next, other design patterns that have also been considered important, although its applicability is less general, are also briefly discussed.

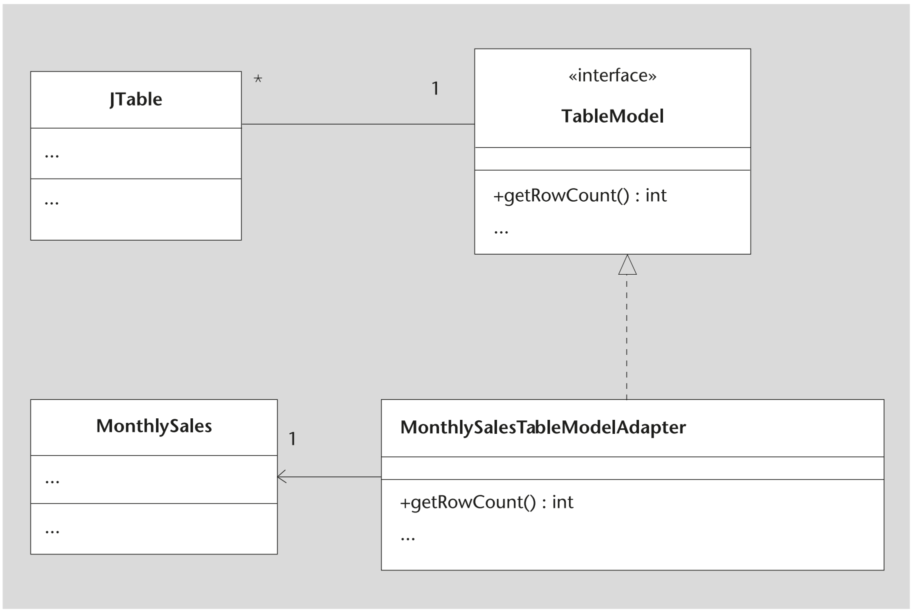
6.8.1. Adapter

It enables the use of a class using a different set of operations than it originally offered.

For example, in the Java Swing graphic application structure, the component JTable uses the interface TableModel to obtain the data it must display. If we want to display the data of our own data structure, we can write an adapter to be able to offer to the component JTable the interface it expects without having to modify our class:

See also

A detailed explanation with examples on adapters can be found in section "Example" of the "Introduction to patterns" module.



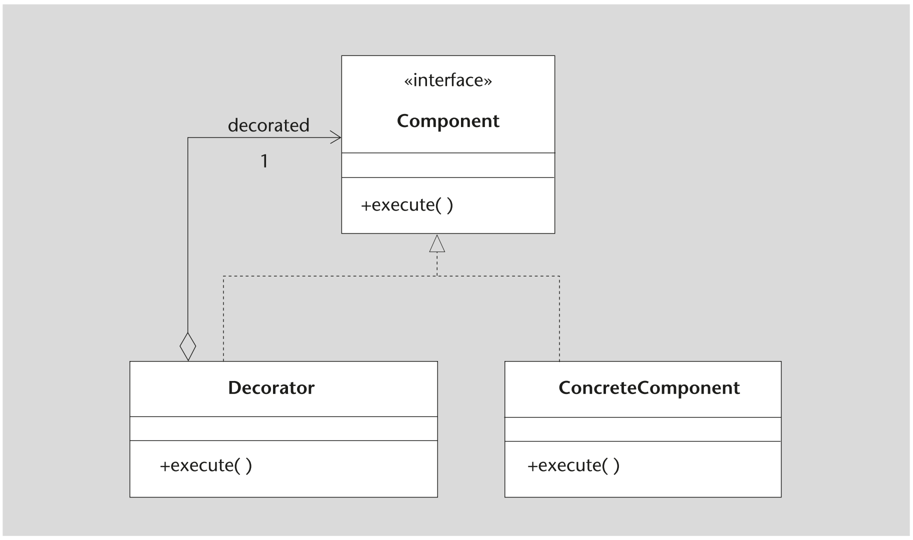
6.8.2. Decorator

It enables the addition of responsibilities to an object dynamically.

The Decorator has the same interface as the object that it decorates, and it implements the additional responsibilities delegating the rest of the responsibilities to the decorated object:

See also

The Decorator is closely related to the Proxy, which is described in section "Proxy" of this module.



For example, let us suppose we are developing a system and we want to make a performance study. We want to be able to add to the Commands that implement our business operations the responsibility of controlling how long do they take to execute. As we want to enable and disable this capability at execution, instead of modifying the classes we have we will add a new class: DecoratorPerformance.

See also

More information on the Command pattern can be seen in section "Command" of this module.

```
public class DecoratorPerformance implements Command {
  Command decorated;
  long timeUsed;
  (...)
  public void execute() {
    long start = System.currentTimeMillis();
    decorated.execute();
    timeUsed = System.currentTimeMillis() - start;
  }
  (...)
}
```

6.8.3. Strategy

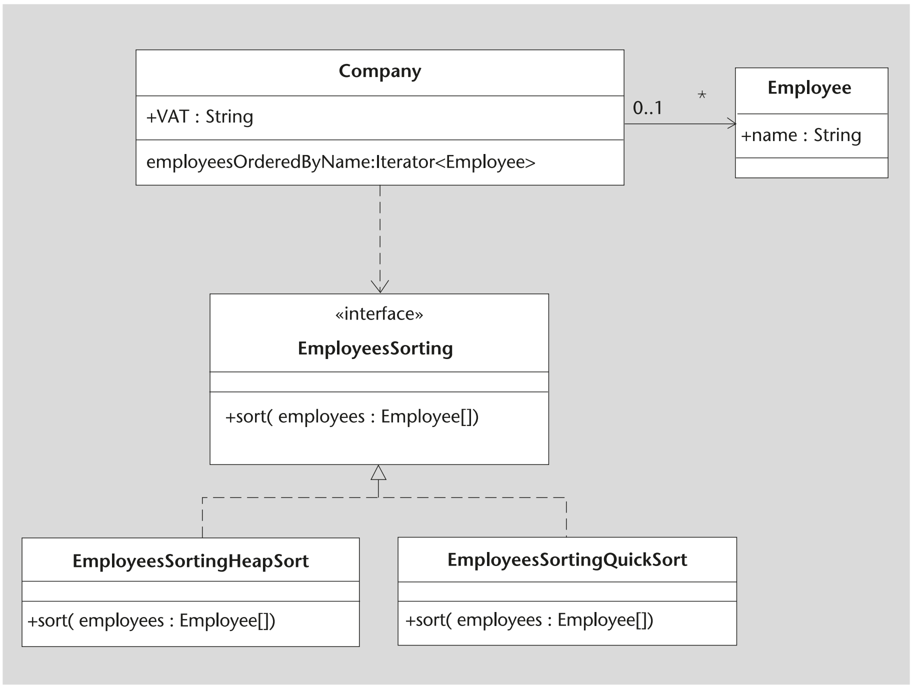
It allows the definition of a family of algorithms and makes them interchangeable with each other. In this way, we can choose one dynamically.

Strategy and Template method

The Strategy design pattern allows solving very similar problems to those that the Template method pattern solves, which can be found in section "Template method" of this module. Often one or the other can interchangeably be used. The main difference lies in the fact that the Strategy pattern uses delegation instead of inheritance.

Strategy allows a general algorithm to be separated from the specific details of each possible implementation, thus facilitating the reuse of the general algorithm independently of the details. In addition, it allows the reuse the implementations of the details for more than one general algorithm since it does not use inheritance but delegation.

Let us suppose that in the system we are developing we have to sort a list and we want to be able to choose which sorting algorithm to use:


Company
+VAT : String
employeesOrderedByName:Iterator<Employee>
0..1
*
Employee
+name : String
«interface»
EmployeesSorting
+sort(employees : Employee[])
EmployeesSortingHeapSort
+sort(employees : Employee[])
EmployeesSortingQuickSort
+sort(employees : Employee[])

In the resulting design, the operation employeesOrderedByName of a company instance will behave in one way or another depending on the implementation of EmployeesSorting that you use. In addition, this behavior could change dynamically if we replace the instance of EmployeesSorting used within the class Company by one of another subclass.

6.8.4. Singleton

It allows us to ensure that there is only one instance of a certain class in the whole system.

The implementation of a Singleton class will not have constructors accessible from outside the class and, therefore, it will be able to be instantiated only by itself.

```
public class Singleton {
  private static Singleton instance = null;
private Singleton() {}
public static Singleton getInstance() {
    if (instance == null) {
     instance = new Singleton();
    }
    return instance;
  }
}
```

Let us suppose we want all classes that access the database in our application to do so via the same instance, so that we can easily control the number of connections established against the database. To achieve this, we have to design the database access class as a singleton class.

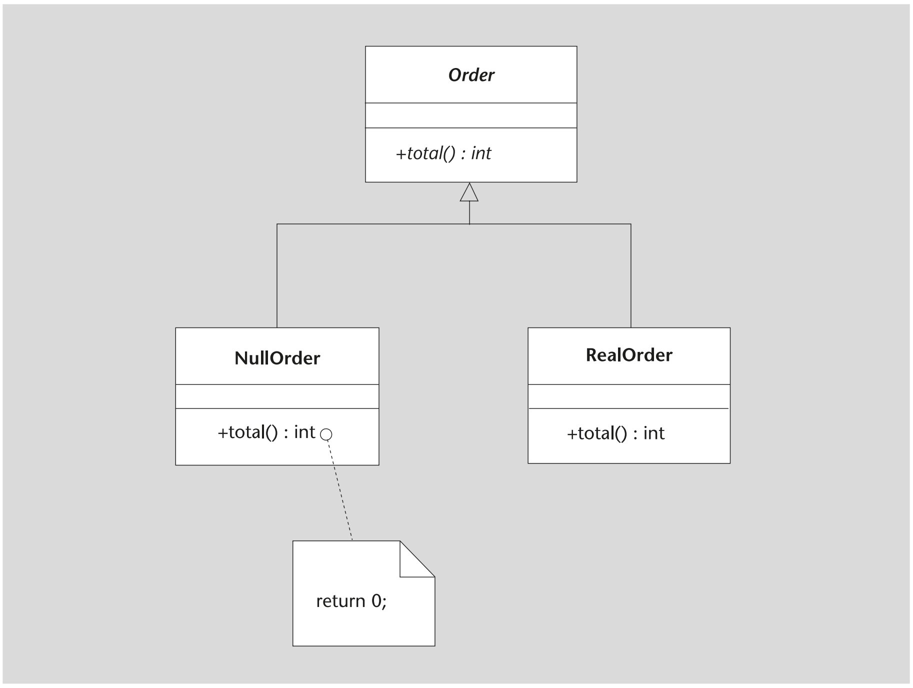
6.8.5. Null object

It allows us to handle the case of the null value as a valid instance.

For example, let us consider this code snippet:

```
int total;
Order c = DB.getOrder("W2132");
if (c != null) {
  total = c.total();
} else {
  total = 0;
}
```

The special treatment we must give the null object considerably increases code complexity. For this reason we consider handling both cases in a uniform way. To do so, the Null Object pattern proposes the creation a subclass of Order that represents the special case in which an Order is empty:



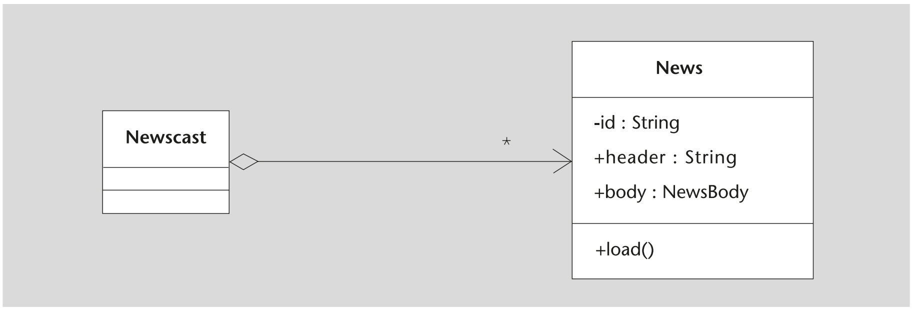
Our code snippet would be as follows:

```
int total;
Order c = DB.getOrder("W2132");
total = c.total();
```

6.8.6. Proxy

It allows us to control the access to an object from another in a transparent way.

Let us suppose, for example, that we are developing a news manegement system:



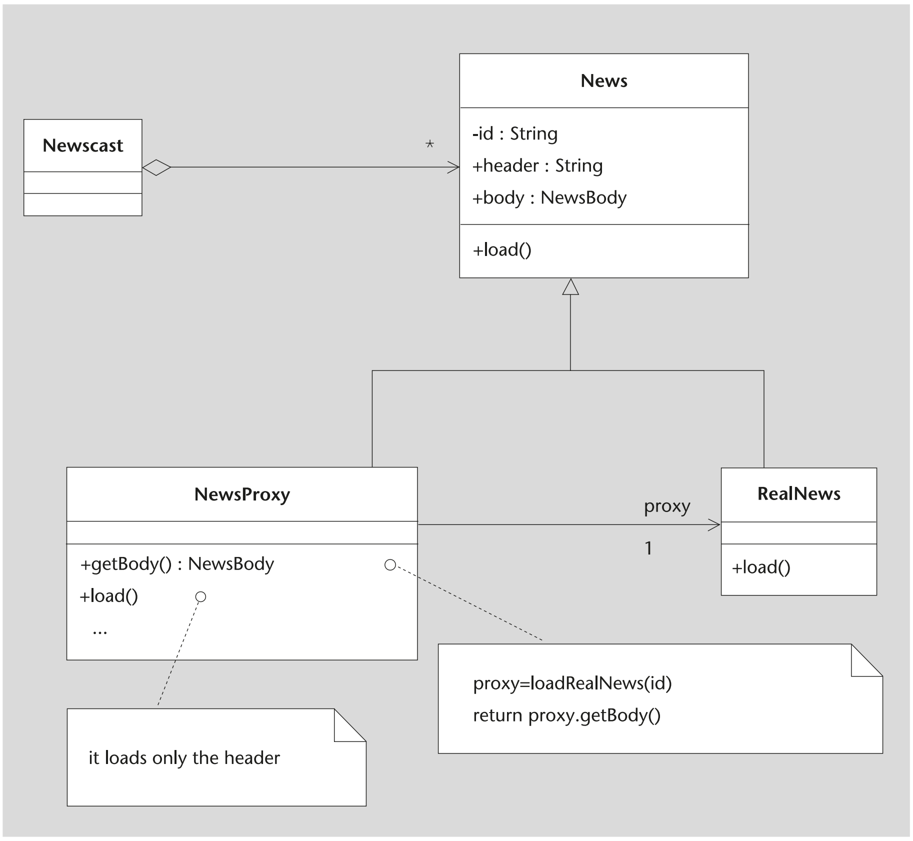
News will be stored in a repository and we will assume that loading their body is a heavy operation, since it can involve loading large amounts of text and, surely, some images. The newscast will normally display the news headlines and, when the user selects one, it will display its body.

To improve system performance, we want to prevent the news body from loading until necessary. But we would like to do it in a transparent way for the News and the Newscast that we have already designed.

The Proxy pattern proposes adding an object (the Proxy) that will control access to the news. When this proxy detects that the body of the news is beeing accessed (and only in this case), it will load it:

Example

We have assumed that access to the body public attribute is implemented by a getBody method in our programming language.



The problem of replacing the original object with its proxy remains to be solved, which is a necessary condition for the application of the pattern to be really transparent to the class Client. The way of solving it will be different in each case and it is outside the range of this pattern.

The Proxy can have other interesting uses such as a cache proxy (which will serve as a cache by saving invocations to the class represented), a remote proxy (which will serve to invoke operations to an object that is located in another node of the network), a security proxy (which will allow us to block requests that do not meet security requirements), etc.

This pattern, in its structure, is very similar to the Decorator pattern. The main difference, however, lies in its purpose: a decorator adds responsibilities to an object by modifying the behavior, while a proxy controls the access.

See also

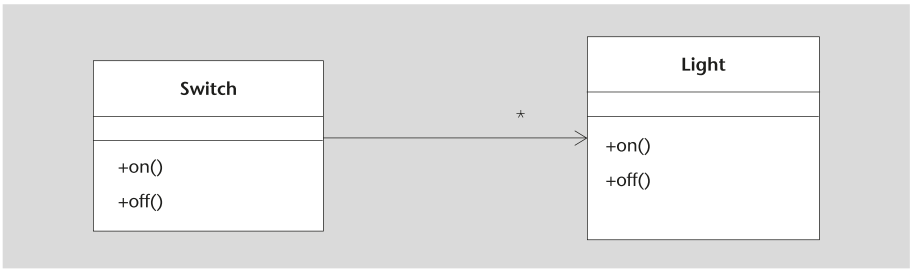
More information on the Decorator pattern can be seen in section "Decorator" of this module.

6.8.7. Abstract server

It enables the decoupling of a client class from a class that it uses which we will call *server*.

To do this, it introduces an abstraction that represents the use of the server made by the client.

For example, let us suppose we have the following classes:

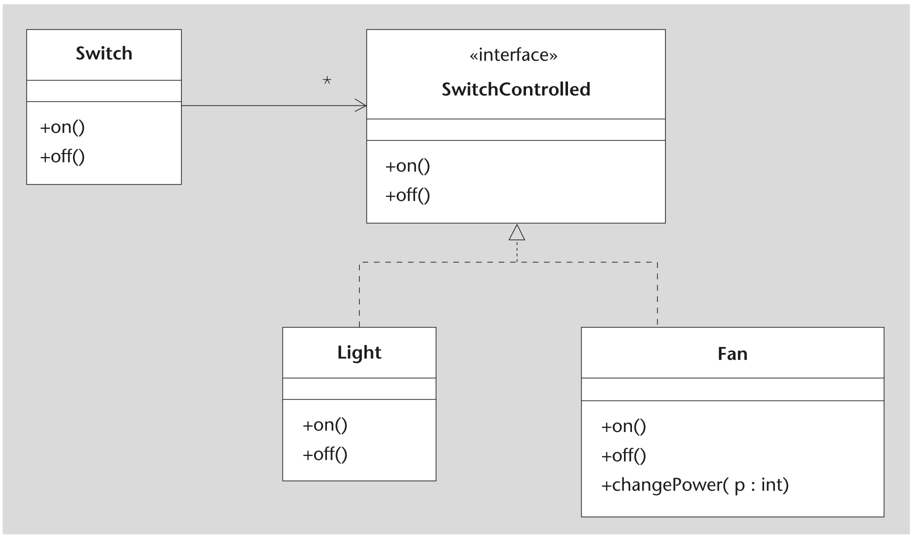


This design couples the class Switch with regard to the class Light. In addition, the open-closed principle is violated: What if we want a switch for a fan?

See also

More information on the open-closed design principle can be seen in section "Open-closed principle (OCP)" of this module.

The Abstract server pattern suggests to introduce an abstraction between Switch and Light that represents the use that switch makes of lights, fans and any other class that needs to be controlled later:



Note the detail of the interface name. The interface with the operations of Light is called *SwitchControlled* and not *Light*. That is due to the fact that Switch knows how to control the interface operations and not necessarily all of Light. That is why we say that the interface belongs to Switch and not to Light. With this design, any device that implements SwitchControlled can be controlled by an instance of Switch.

If the objects that must implement the new interface already exist, we will surely have to adapt the current interface to that which the Abstract server introduces. To do this, we can rely on the Adapter pattern.

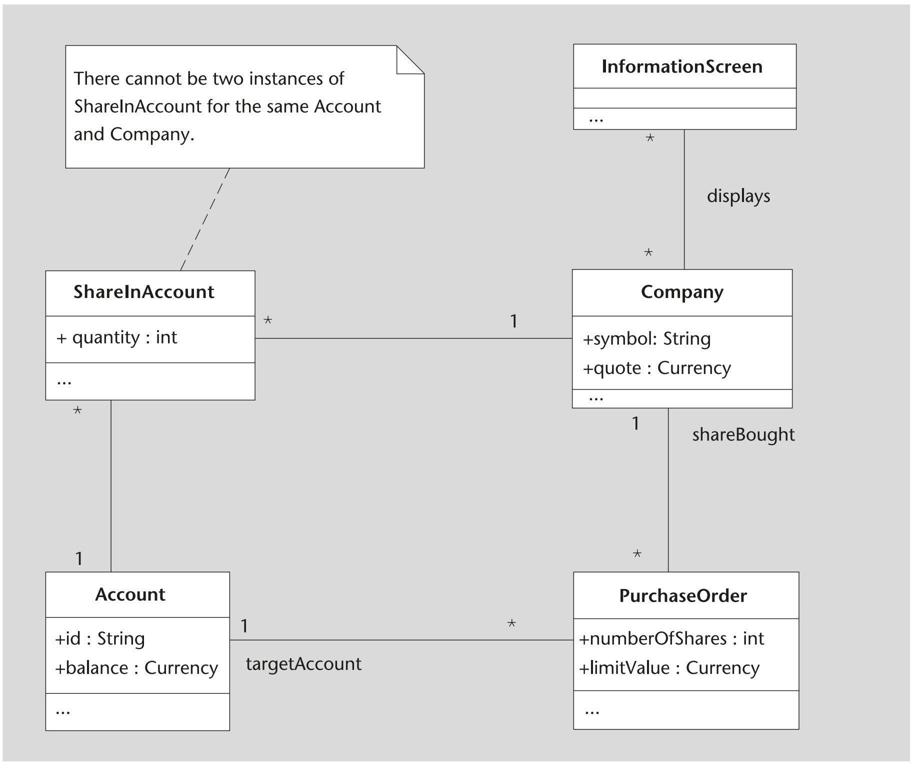
See also

More information on the Adapter pattern can be seen in section "Adapter" of this module.

Self-evaluation

The system we are developing manages information on stock prices, purchase orders, and also some screens that display the stock value of the different companies.

Users can program purchase orders so that when the price of a company's stock falls below a certain limit value (indicated in the order), the system automatically buys the number of shares that have been programmed in the order and adds them to the user's account:



Note

The class ShareInAccount is the result of normalising the association class of the "stock-Held" association between Account and Company.

1. We want to design an operation that we will call executePurchaseOrder that, when a purchase order is made, calculates the value of the shares purchased, verifies the balance of the account and, if the balance is sufficient, adds the amount of shares purchased to the portfolio of the account and discounts the price of the balance. Apply responsibility assignment patterns to assign the various responsibilities associated with this functionality.

2. Actually, the execution of the operation of the previous task should be done automatically when there is a change in the price of the shares of a company that makes any purchase order become executable. In addition, we want the information screens to be updated automatically each time the value of any of the shares displayed on the screen is modified. Make a design for this behavior using a design pattern that helps you avoid coupling the class Company to screens and purchase orders.

Answer key

1. For this operation we will have the following assignment of responsibilities:

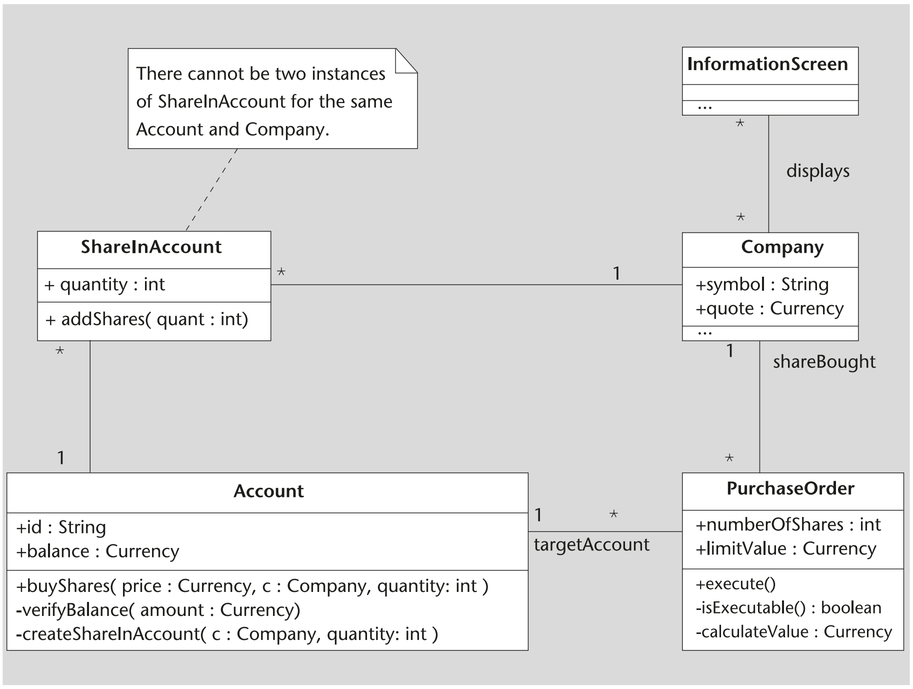
Verifying if Purchase order must be executed: to do so, we must consult the quotation value (it is Company who knows it) and compare it with the limit value of the order (it is Purchase Order who knows it). In this case, by cohesion we will assign this responsibility to the Purchase Order so that it is the only one that knows how it is decided if a purchase order is executble.

If the order is executable, it will be necessary to calculate the amount of the shares purchase, to verify the account balance and, if there is sufficient balance, to add the number of shares indicated in the purchase order to the target account. In this case, the Order is the expert to calculate the purchase amount, while the Account is the one who knows the balance and the shares.

The expert who knows how many shares of a given company an account has is the class ShareInAccount.

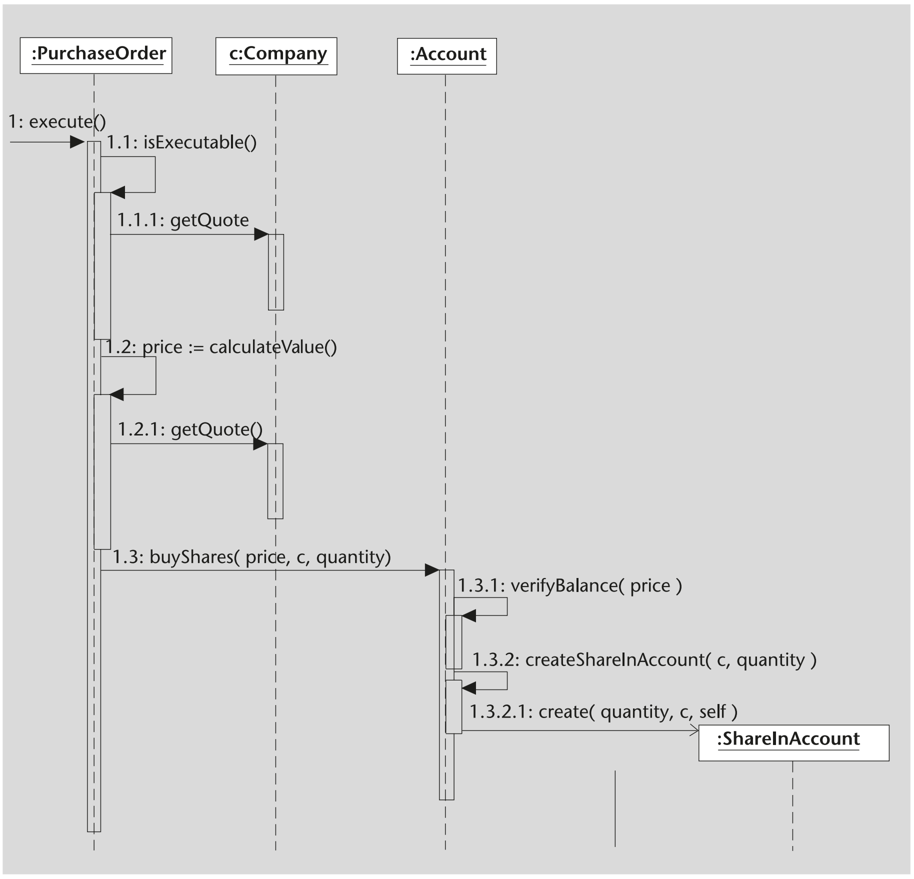
It may happen, however, that the Account does not previously have any Share of the same company and, therefore, a new instance of ShareInAccount must be created. In this case, according to the Creator pattern, it will be the Account who is responsible for creating this instance, since the Account instance contains its instances of SharesInAccount.

Following this criterion, then, we obtain the following diagram:


There cannot be two instances of ShareInAccount for the same Account and Company.
InformationScreen
...
*
displays
*
ShareInAccount
+ quantity : int
+ addShares(quant : int)
*
1
Company
+symbol : String
+quote : Currency
...
1
shareBought
*
*
1
Account
+id : String
+balance : Currency
+buyShares(price : Currency, c : Company, quantity: int)
-verifyBalance(amount : Currency)
-createShareInAccount(c : Company, quantity: int)
1
*
targetAccount
PurchaseOrder
+numberOfShares : int
+limitValue : Currency
+execute()
-isExecutable() : boolean
-calculateValue : Currency

If this purchase were to be made as a result of a user-generated event, there would also be a controlling class responsible for receiving this event and invoking the operation created of the class PurchaseOrder.

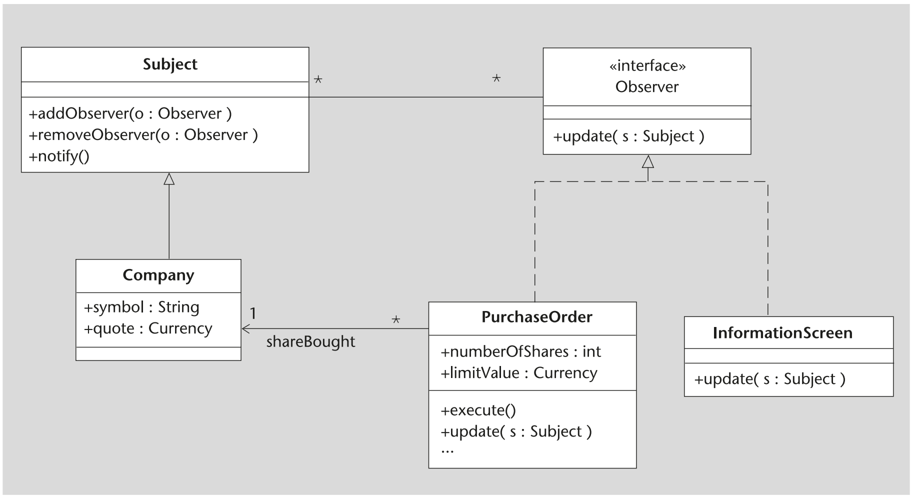
In the scenario where the purchase ends up being made (the quotation value is below the limit value and there is sufficient balance) and where the account did not previously have any company share, the sequence diagram resulting from the assignment of responsibilities previously made is as follows:


:PurchaseOrder
c:Company
:Account
1: execute()
1.1: isExecutable()
1.1.1: getQuote
1.2: price := calculateValue()
1.2.1: getQuote()
1.3: buyShares(price, c, quantity)
1.3.1: verifyBalance(price)
1.3.2: createShareInAccount(c, quantity)
1.3.2.1: create(quantity, c, self)
:ShareInAccount

2. The Observer pattern allows the notification of the changes that occur in a company to those objects that are interested (in our case, screens and purchase orders) with minimum coupling.

> **Note**
>
> Company will call the operation Notify when its price attribute is modified. The operation Update from PurchaseOrder will invoke the already designed operation Execute.

The structure of the solution will be as follows:



The association "display" between InformationScreen and Company is no longer necessary, since it was only used for notification purposes (which now are solved by the pattern). Similarly, the ability to navigate from Company to PurchaseOrder has been removed, since it is no longer necessary (although the other direction of navigation has been maintained because it is necessary beyond the notification). In this way, coupling between classes has been reduced.

Bibliography

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

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

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

[MAR] Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, New Jersey: Prentice Hall.

Additional bibliography

[BECK] Beck, K. (2002). *Test Driven Development: By example*. Massachusetts: Addison Wesley Professional.

[FOWDI] Fowler, M. (2004). "Inversion of Control Containers and the Dependency Injection pattern". martinFowler.com. Available in: http://www.martinfowler.com/articles/injection.html

[HUNT] Hunt, A.; Thomas, D. (2000). *The Pragmatic Programmer.* Massachusetts: Addison Wesley.

[MEY] Meyer, B. (1999). *Object-Oriented Software Construction*. Upper Saddle River, 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.