Use-Case Controller

Authors Ademar Aguiar
FEUP, INESC Porto
ademar.aguiar@fe.up.pt

Alexandre Sousa
ISMAI, ParadigmaXis, SA
avs@ismai.pt

Alexandre Pinto
ParadigmaXis, SA
alexandre.pinto@paradigmaxis.pt

The *Use-Case Controller* pattern deals with the problem of mapping use case specifications to an implementation that is cost-effective and easy to maintain. The pattern suggests delegating the responsibility for managing the use-case flow of execution to a *use-case controller* object, thus localizing this knowledge and its eventual future changes. *Use-case controllers* provide a uniform way of coordinating actor events, user interfaces and system services. Additionally, use-case controllers enlarge the potential variability of use cases, either by enabling the plug-in of use case extensions and inclusions, or by replacement of user interface components and system components.

Example Consider an order-processing system for an online order company that is a reseller of products from several suppliers. When customers want to purchase products, they place an order accompanied with payment information. While adding products for an order, customers are able to save it for later use. Orders can be freely cancelled after being placed, but before being shipped.

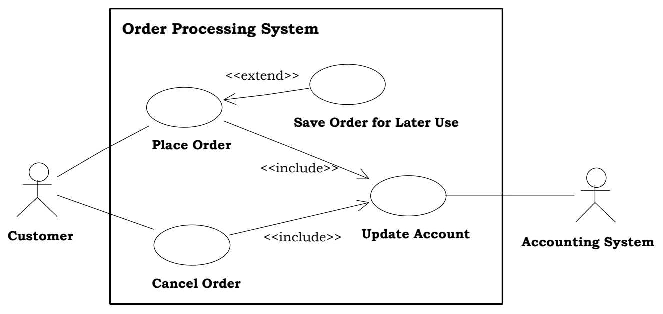


Figure 1. Use case diagram for part of an order-processing system.

The outside view of the order-processing system described above is represented in Figure 1. in the form of a use-case model. Use-case models are a popular way of capturing user intent and system functionality as it appears to an outside user [8][10][11][12]. Use cases help to define the boundaries of a system and describe the ways in which that system interacts with external entities, the *actors*. A use case is "*a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor*" [5].

When mapping a use case model to an object-oriented implementation, several refinements and tasks are undertaken, such as: detailed description of use cases, user-interface design, identification of participating classes, and assignment of responsibilities to classes.

Four different artefacts related with the `Place Order` use case are shown in Figures 2, 3, 4 and 5, respectively: a brief textual description, an activity diagram, classes participating in a use-case realization, and an user-interface.

Use Case	**Place Order**
Description	The use case starts when a customer selects the option to place an order. The customer identifies himself to the system and enters the products he wants, for which the system supplies a description, price and a running total. After entering payment information, the customer is able to submit the order. The system then verifies the order submitted, saves it as pending, and forwards payment information to the accounting system. When the payment is confirmed, the order is marked confirmed, an order ID is sent to the customer, and the use case ends.

Figure 2. Brief textual description of the Place Order use case.

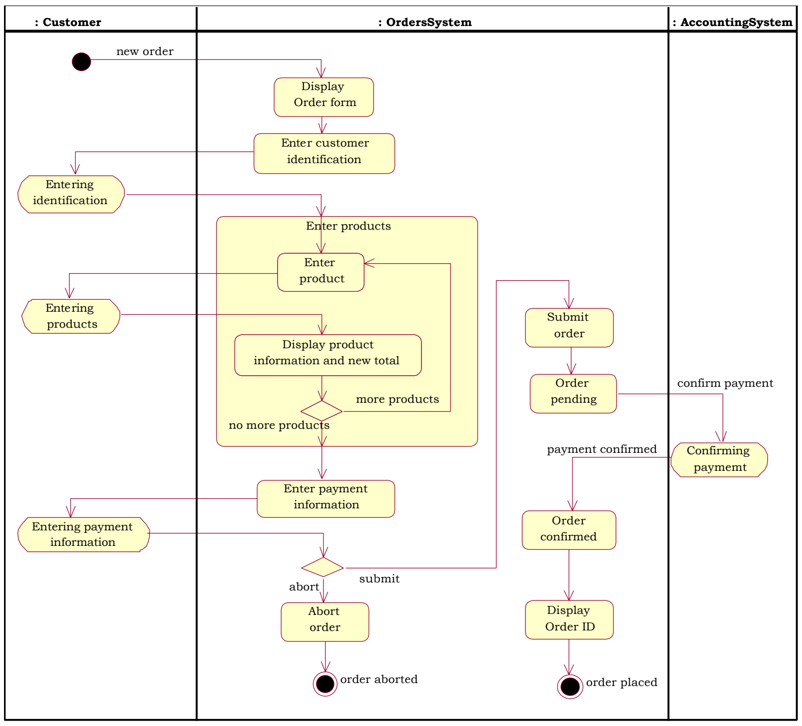


Figure 3. Activity diagram of the Place Order use case.

The implementation of a use-case is responsible for coordinating and sequencing the interactions between the system and its actors, while preserving the integrity of the data and the flow defined in the use case specification. Three different types of objects usually participate in a use-case realization, each one encapsulating a different kind of behaviour. These types were initially described in [8] and are now widely known and adopted by UML [5] as the stereotypes «boundary», «entity» and «control»:

- *interface objects* are generally used to translate actor's actions to the system into events in the system, and vice-versa;
- *entity objects* are used to handle information that outlives use cases and often persistent;
- *control objects* are used to encapsulate behaviour related to a specific use case that isn't naturally placed in any of the previous two types of objects, such as coordination, sequencing, transaction, and control of other objects. Control objects usually last only during the execution of one use case.

In Figure 4. are shown the classes that participate in the realization of the *Place Order* use case and also their corresponding types.

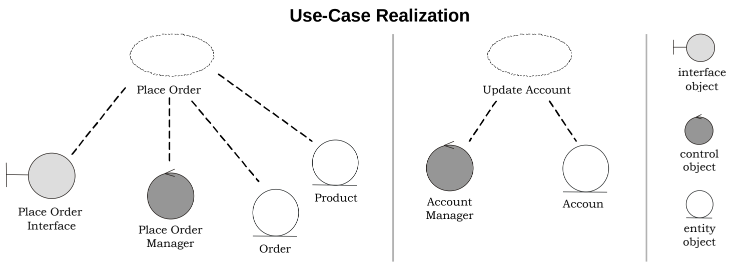
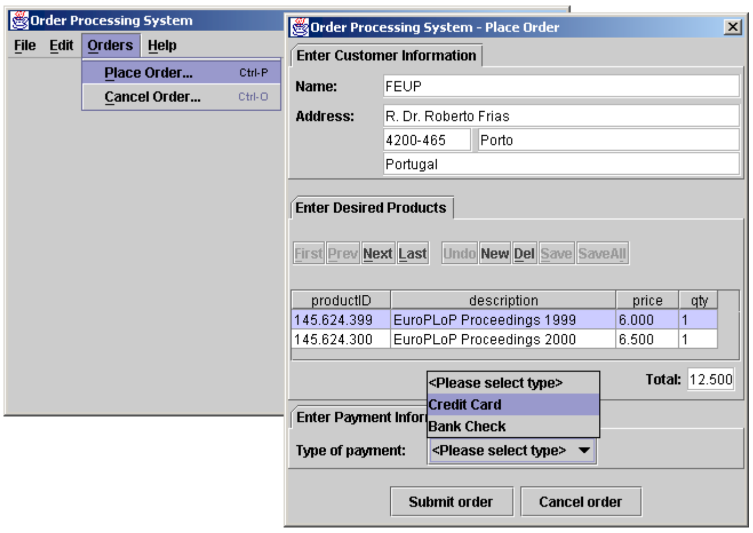


Figure 4. Classes participating in the realization of Place Order use case.

In some situations, different users executing the same `Place Order` use case may require different user interfaces, such as: a web interface for customers, graphical windows interface for customer reps, or a textual interface for typist operators. As a result, a use case implementation is often required to support the replacement of user interface components and system components with others also satisfying the requirements given by the use case.



Name: FEUP
Address: R. Dr. Roberto Frias
4200-465 Porto
Portugal

productID	description	price	qty
145.624.399	EuroPLoP Proceedings 1999	6.000	1
145.624.300	EuroPLoP Proceedings 2000	6.500	1

Total: 12.500

<Please select type>
Credit Card
Bank Check

Type of payment: <Please select type>

Figure 5. User interface for the Place Order use case.

A use case implementation can be obtained using different approaches, which may have distinct impact in terms of modularity, coupling, reusability and changeability of the resulting code.

Context You are developing an interactive system following a use-case driven approach.

Problem Use case modelling is a popular method of determining user requirements [8][10][11][12] that helps to define the boundaries of a system and identifies system-actor interactions, i.e., use cases.

Before coding an application modelled with use cases, architectural and design decisions must be made. Use cases can be realized using a combination of three types of models: collaborations of subsystems and classes, state machines, and activity models [2][6]. Use case maps are another alternative that combines behaviour and structure in one view and shows the allocation of scenario responsibilities to system components [17][18].

Independently of the way it is implemented, a use case must coordinate system actions in response to user actions and to take control over:

- the events and input coming from user interface components;
- the invocation of services provided by system components;
- the execution of related use cases, such as extensions and inclusions, both helpful to improve modularity and reusability.

In the development of a family of applications, it is also desirable to have use-case implementations that can be used in several applications, although combined and configured in different ways [9].

When designing use case realizations that satisfy the requirements above mentioned, the following questions arise:

– *How do you design a use case implementation so that you can map easily and efficiently your use case models to an organization of components?*

– *How do you design a use case implementation that can be flexibly configured with their participating components, namely user interfaces and system components?*

– *How can we implement use cases in a way that facilitates their reuse across a family of applications?*

Forces A solution to such a design problem should balance the following forces:

- A use case implementation should preserve the integrity of the use case flow specified.
- The behaviour of a use case implementation should be easily configurable with other use cases, either by extending it or by replacing some of its parts.

- The same use case implementation should be executable with different user interfaces and system components, regarding that they satisfy the requirements given by the use case.
- A use case implementation should be easy to relate to the corresponding use-case model and thus preserve the traceability to and back earlier development models.
- Applications should be configurable by assembling use case implementations, once it has been ensured that these implementations are compatible and coherent in terms of the user interfaces and system services required.

Solution The solution proposed in this pattern follows the guidelines for designing use case realizations presented in [8][9][7] and specializes the Model-View-Controller pattern [4] for the specific purpose of controlling the flow of a use case.

The pattern suggests the division of a particular use case implementation into three kinds of participants:

- A *model* that provides a single interface to a set of entity objects that encapsulates system information and behaviour outliving a use case execution;
- A *view* that wraps boundary objects through which actors communicate with the system, and inversely;
- A *controller* that manages the flow of interactions between users and system, and coordinates associated *views* and *models*.

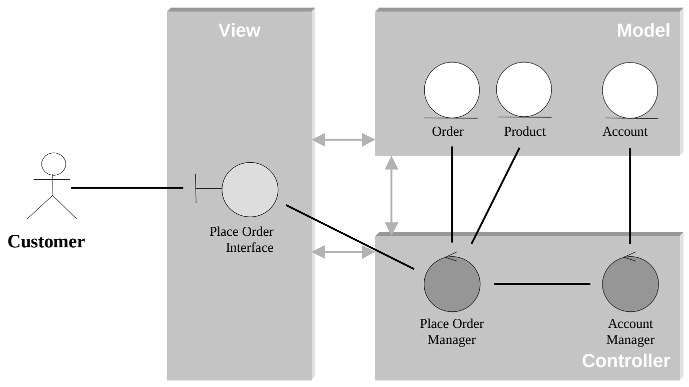


Figure 6. Relationships between object types and MVC metaphor.

The *model* can be described as a facade [3] to a set of services eventually distributed by several subsystems. Although many different services can be included in this interface, the intent is to include only those strictly needed to implement the corresponding use case. A first outline of this facade can be derived from the corresponding diagram with classes participating in the use case.

Views are interface objects needed by the use-case implementation to present information to the user. Views are responsible for the translation of user actions to events meaningful from the point of view of the use-case specification. In other words, views support bi-directional communication between the system and its users.

The *controller* is the component responsible for ensuring the correct flow of a use case, both its basic thread and its variants. A controller receives meaningful events from components and, based on the use case specification, decides the valid sequence of action steps to perform. The internal state of a controller records the execution of a particular scenario, thus knowing why it was performed. A controller is also responsible for managing the associations with other participating components, namely the *model, views* and other related use cases (*extensions*, *inclusions* or *generalizations*).

Structure

The *Use-Case Controller* pattern defines five types of participants: *UseCaseController*, *View*, *Model, Extension* and *Inclusion*.

Class UseCaseController	***Collaborations***
Responsibilities • Ensures the correct execution of an use case • Handles use case events from views and models	• View • Model • Extension • Inclusion

An instance of *UseCaseController* is an object that encapsulates the knowledge needed to ensure the correct execution of one particular use case specification: sequence of action steps, states, preconditions, post-conditions, and components, both for accepting user input and for performing system services.

As a rule of thumb, one *UseCaseController* will be created for each use case. When a use case is started, one instance of the corresponding use-case controller is created to control the flow of the interactions involved. Depending on the size and complexity of the controller, one might decide to create several use-case controllers for a particular use case, or the opposite.

A use-case controller is not intended to interact directly with either interface objects or system objects, but only to *model* and *view* objects, thereby promoting a layer at the use-case level.

Class Model	***Collaborations***
Responsibilities • Provides system information and functionality needed during use case execution	• Use-case Controller

Class View	***Collaborations***
Responsibilities • Presents information to actors • Receives actors input	• Use-case Controller

A use-case controller is typically associated with a *model* facade, through which it has access to the subset of data and functionality needed to perform system-related actions. On the other hand, to communicate with the actors, a use-case controller relies on *view* facades. A *view* is usually an aggregation of several interface objects, such as windows, dialogs, menus and buttons.

Use case inclusions and extensions can be mapped directly to an implementation by connecting a use-case controller with other use-case controllers through *hook* objects.

Class Extension	***Collaborations*** • Use-case Controller
Responsibilities • Implements the <<extend>> association • Knows when and where the extension is valid	

Class Inclusion	***Collaborations*** • Use-case Controller
Responsibilities • Implements the <<include>> association • Knows where the inclusion is to be done	

Using an *extension hook*, a use-case controller can be connected with others that augment its behaviour at predefined points and circumstances. The execution of extensions interrupts the execution of its base use-case and is usually triggered by conditions internal to the base use-case controller, either at defined extension points or during its overall execution. Base use-case controllers do not need to know their extensions.

An *inclusion hook* is similar to an extension hook, except that their execution is needed to realize the behaviour of the base use-case. Inclusions are known by the caller use-case controller [8][5][1], and initiated as and when needed during the execution flow of the base use-case. Base use-case controllers must know their inclusions.

In addition to including and extending relationships, use cases can also be related by generalization relationships, which mean that a use case may specialize a more general one [1]. This kind of relationship between use cases can be implemented by inheritance of use-case controllers, models and views, thus it is easily accommodated by the solution proposed.

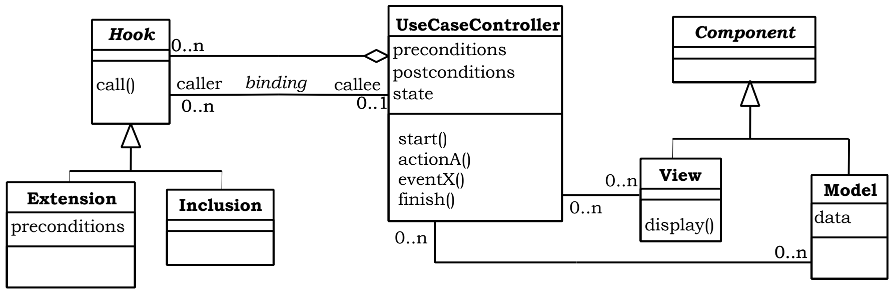


Figure 7. Class diagram representative of the pattern structure.

In Figure 7. it is shown the UML class diagram representative of the pattern structure.

Implementation The following guidelines are suggested to implement this pattern.

1. *Identify candidate use cases* by looking in the use case model for use cases that may benefit from the solution proposed by this pattern. Use cases usually involve interface objects and entity objects. In a first approach, all use cases can be defined as candidates. However, use cases whose behaviour is completely placed in interface objects and entity objects do not need to be implemented with use-case controllers because no behaviour is left to the control object. The same happens when one decide that control objects are better encapsulated within interface objects. On the other hand, a control object can be so complex that it is better to encapsulate it in two or more use-case controllers. Examples of functionality typically placed in use-case controllers are transaction-related behaviour, sequencing behaviour specific to one or more use cases, or behaviour to couple together interface objects and entity objects [8].

   * In the example presented, *Place Order* can be considered a good use case candidate, as it involves specific functionality that do not assign well either to interface objects or entity objects. On the other hand, the use case *Save Order for Later Use* is not a good candidate because its behaviour is very simple and do not involve coordination of interface objects and entity objects.

2. *For each one of the use cases above create a use-case controller class.* Although it is possible, and sometimes convenient, to create a use-case controller class for each use case candidate, this decision must be made by the developer after refinement of the use-case participant classes and, thus, knowing well the events exchanged between them. At this stage, it can be decided to assign one, two or more use-case controllers to a single use-case, or even none. Typically, a use-case controller class includes methods to start and finish its execution, to invoke specific use-case actions, and to fire events to interface objects and entity objects.

   * The use-case *Place Order* can be conveniently implemented with one single use-case controller class.

```
abstract public class UseCaseController {
  abstract public void start();
  abstract public void finish();
}

public class PlaceOrderUseCase extends UseCaseController {
  …
  public void start() { displayForm(); }
  public void finish() { … }
  public void displayForm() { … }
  public void setCustomerInfo(String name, String address) { … }
  public void newItem() { … }
  public void saveItem(String item) { … }
  public void deleteItem() { … }
  public void setPaymentInfo(String paymentType) { … }
  public void submitOrder() { … }
  public void cancelOrder() { … }
}
```

3. *Create the model and view objects needed to implement each use case.* Use-case controller classes are responsible to coordinate the communication between entity objects and interface objects. After analysing the classes needed to perform the use case's flow of events, the behaviour must then be distributed to the interacting objects. All participating classes can be divided at least in two groups: a model, aggregating all entity objects; and views, which aggregate interface objects.

   Different strategies can be used to allocate the functionality to entity objects, interface objects and control objects [7][8]. The strategy to choose must be decided from application to application. In most cases, the balanced strategy, where control is placed separately from both interface and entity objects, is the one that reaches higher locality in changes of the functionality.

   * The model for *Place Order* use case can be simply obtained by providing a common access-point to the entity objects needed, as shown in the code below, or including also helper methods that make more convenient the usage of the model, such as methods for automating data conversions. Views may be obtained using a similar approach.

```
public class PlaceOrderModel {
   public Order order;
   public Vector products;
   public Vector paymentTypes;
}

public class PlaceOrderDialog extends javax.swing.JDialog
{
  PlaceOrderUseCase useCase;
  public PlaceOrderDialog(UseCaseController useCase) {
        this(null);
        this.useCase = useCase;
        ...
        submit.setText("Submit order");
        submit.addActionListener(new java.awt.event.ActionListener() {
              public void actionPerformed(java.awt.event.ActionEvent event) {
                    useCase.submitOrder();
              }
        });
  }
}
```

4. *Create extension and inclusion hooks for related use cases.* Extensions and inclusions can be simply implemented with direct calls between use cases. Extension and inclusion hooks must be implemented when pretending to connect use-case implementations at run-time. In these cases, some implementation mechanism must be supported by use-case components for achieving the effect of plug-points and plug-ins.

   The patterns TEMPLATE METHOD and STRATEGY [3] document two well-known solutions to the problem of connecting a component at run-time. Use-case inclusions can be connected only at predefined plug-points of the base use-case, but use-case extensions can be connected at any point, assuming they satisfy the requirements of the base use-case.

* For clarity reasons, the example presented here is very simple and do not justify the implementation of a plug-in mechanism. However, both inclusion and extension hooks are implemented to demonstrate here a simple way of supporting hooks in the *Place Order* use-case.

```
public class PlaceOrderUseCase extends UseCaseController {
  …
  Hook updateAccount = null;
  Vector extensionHooks = new Vector();
  …
  public void connectUpdateAccount(UseCaseController useCase) {
        updateAccount = new Hook(useCase);
  }

  public void connectExtension(UseCaseController useCase) {
        extensionHooks.add(new Hook(useCase));
  }

  public void submitOrder() {
        …
        updateAccount.call();
        …
  }
}
```

5. *Package use-case controllers using a similar organization as the underlying use-case model.* One of the benefits of implementing use-case realizations using use-case controllers is that the code can be traced back to the originating use-case model. In systems with many use cases it may help to package use-case controllers using similar criteria to the one followed in the use-case model, thus creating a use-case layer at the application logic layer.

Consequences

The *Use-Case Controller* pattern provides the following *benefits*:

- *Localization of business rules and policies.* Use-case controllers work as glue between user interfaces and domain entities and place much of the application logic separate from dialogue and system components. Therefore, it will be easier to incorporate changes without affecting interfaces, system objects or database schemas, because future changes of the functionality will have a high locality in use-case controllers.
- *Easy replacement of user interfaces and system components.* Use-case controllers are separated from the user interface and system components. Therefore, multiple user interface components and system components can be implemented and used by a single controller, and can be easily substituted statically and dynamically.
- *Visibility of the originating use case model in the implementation code.* The pattern suggests a straightforward design and implementation of use-case realizations. Assuming that the pattern is consistently instantiated following some coding convention, use-case implementations will be easy to recognize at the code level and to be traced back to the respective model.

- *Flexible configuration of applications using pluggable use case implementations.* Use-case controllers encapsulate the most important functionality specified in the use case, and can be easily replaced even at run-time. Therefore, the pattern allows the configuration of applications by plug-in of use case components having use-case controllers at their heart.

In contrast, the pattern imposes the following *liabilities*:

- *Increased number of classes.* As the pattern introduces a conceptual separation between user interface, model and controller objects, more classes are needed to implement the same functionality. This aspect must be considered having in mind the benefits of an increased separation of concerns and changeability.
- *Increased complexity.* The pattern suggests the assignment of a control object for each use case, which can result in code more complex than the original. When the use-case behaviour is not complex enough to justify the usage of a separate control object, the control object should be avoided.
- *Control flow of the application slightly harder to trace directly from a strict reading of the code* as one must be aware of the design principles and coding convention that lead to the application logic being organized around the concept of use cases. On the other hand, once known the underlying use-case model, the readability and modularity of the implementation is largely improved.
- *Close relation between models and views to controllers.* Controllers directly invoke functionality from models and views, thus implying that changes to interfaces of models and views may have impact on controllers. This problem can be reduced with the use of FACADE, ADAPTER or BRIDGE [3].
- *Potential instability of the implementation* mainly when the underlying use case model is unstable or badly organized. For these situations it may help to decompose use-cases into more fine-grained components, usually called actions [8] or action steps [1], which are generally more stable as they closely map from essential system requirements. The aggregation of these actions around use-cases can improve usability but at the cost of being more specific and constrained. Therefore, it is important to separate this use-case layer from the overall application logic, to improve system changeability.

Related Patterns

MODEL-VIEW-CONTROLLER [4] differs from this pattern, because its applicability is wider and thus less useful for the particular problem of implementing use case realizations.

COMPOSITE and FACADE [3] may be used to combine several user interface components or views in one single object. The same applies for entity objects and models, respectively.

ADAPTER [3] can help on the conversion of interfaces of system and user interface components. BRIDGE [3] can decouple user interface abstractions from their implementation so that both can vary independently without disrupting use-case controller's interfaces.

Use-case controllers can be implemented using the COMMAND [3] or COMMAND PROCESSOR [4] patterns so that their execution may be started uniformly using a single method.

CHAIN OF RESPONSIBILITY [3] or BUREAUCRACY [16] can be useful for implementing the handling of requests on a use-case controller.

J2EE patterns catalogue [19] include FRONT CONTROLLER, SESSION FACADE, DISPATCHER, COMPOSITE VIEW and AGGREGATE ENTITY patterns, all of them related to the problem dealt by this pattern, but not so adapted to the realization of use-cases. FRONT CONTROLLER and DISPATCHER, both together, aggregate functionality similar to that of use-case controller. FRONT CONTROLLER provides a centralized controller for handling requests and it typically works in coordination with a DISPATCHER component, responsible for view management and navigation. SESSION FACADE deals with the problem of reducing the number of use-case controller objects when implementing each use-case as a session bean. COMPOSITE VIEW and AGGREGATE ENTITY simplify the handling of multiple views and multiple entities by aggregating them in a uniform composite object, thus having a role similar to the *view* and *model* participants defined in this pattern.

Known Uses

In [7][8][9] are presented several examples of using controllers to manage the flow of use cases, although they are not detailed at the code level, but only at a design level.

SIMATWARE application framework [15] provides a dynamic configuration of applications based on the plug-in of use case components implemented with use-case controllers.

JAVA 2 ENTERPRISE EDITION has many uses of this pattern documented in [19]. J2EE PATTERNS catalogue includes specific patterns to help solving some of the problems that arise when instantiating this pattern to J2EE platform, namely: FRONT CONTROLLER, SESSION FACADE, DISPATCHER, COMPOSITE VIEW and AGGREGATE ENTITY patterns.

Credits

Thanks to those who had contributed to this work, for their ideas and critics on its early drafts, specially, Gabriel David and José Bonnet.

Special credits goes to Kevlin Henney, our EuroPLoP'2001 shepherd, for the inspiring questions and valuable feedback to this pattern.

References

[1] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2000.

[2] D. Rosenberg, K. Scott, Use Case Driven Object Modelling with UML: A Practical Approach, Addison Wesley Longman, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern Oriented Software Architecture – a System of Patterns, John Wiley and Sons, 1996.

[5] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modelling Language User Guide.* Addison-Wesley, 1999.

[6] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modelling Language, version 1.3*, Object Management Group, June 1999.

[7] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process*, Addison-Wesley, 1999.

[8] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, 1992.

[9] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley, Reading, Massachusetts, 1997.

[10] I. Jacobson, S. Bylund, P. Jonsson, Using Contracts and Use Cases to Build Pluggable Architectures, JOOP, May-June, 1995.

[11] I. Jacobson, *Use Cases in Large-Scale Systems*, ROAD, March-April 1995.

[12] J. Rumbaugh, Getting Started: Using use cases to capture requirements, JOOP, September 1994.

[13] K. Henney, *Design - Concepts and Practices*, ACCU Conference, www.accu.org, September 1999.

[14] OMG. *Unified Modeling Language Specification*, Object Management Group, www.omg.org, 1998.

[15] ParadigmaXis. *SIMATWARE Application Framework*, Internal Report, ParadigmaXis, SA, March 2001.

[16] R. C. Martin, D. Riehle, and F. Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley, 1997.

[17] R.J.A. Buhr, and R.S. Casselman: *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995.

[18] R.J.A. Buhr, Use Case Maps as Architectural Entities for Complex Systems, in Transactions on Software Engineering, IEEE, December 1998.

[19] SJC, *J2EE Patterns*, Sun Java Center, java.sun.com, March 2001.