16.10 Controller

Solution Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- Represents the overall system, device, or subsystem (*facade controller*).
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session (*use-case or session controller*).
    - o Use the same controller class for all system events in the same use case scenario.
    - o Informally, a session is an instance of a conversation with an actor. Sessions can be of any length, but are often organized in terms of use cases (use case sessions).

Corollary: Note that "window," "applet," "widget," "view," and "document" classes are not on this list. Such classes should *not* fulfill the tasks associated with system events, they typically receive these events and delegate them to a controller.

Problem Who should be responsible for handling an input system event?

An input **system event** is an event generated by an external actor. They are associated with **system operations**—operations of the system in response to system events, just as messages and methods are related.

For example, when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended." Similarly, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.

Example In the NextGen application, there are several system operations, as illustrated in Figure 16.13, showing the system itself as a class or component (which is legal in the UML).

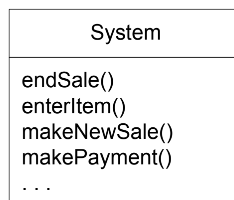

Figure 16.13 System operations associated with the system events.

> During analysis, system operations may be assigned to the class *System*, to indicate they are system operations. However, this does *not* mean that a software class named *System* fulfills them during design. Rather, during design, a Controller class is assigned the responsibility for system operations (see Figure 16.14).

Who should be the controller for system events such as *enterltem* and *endSalel*

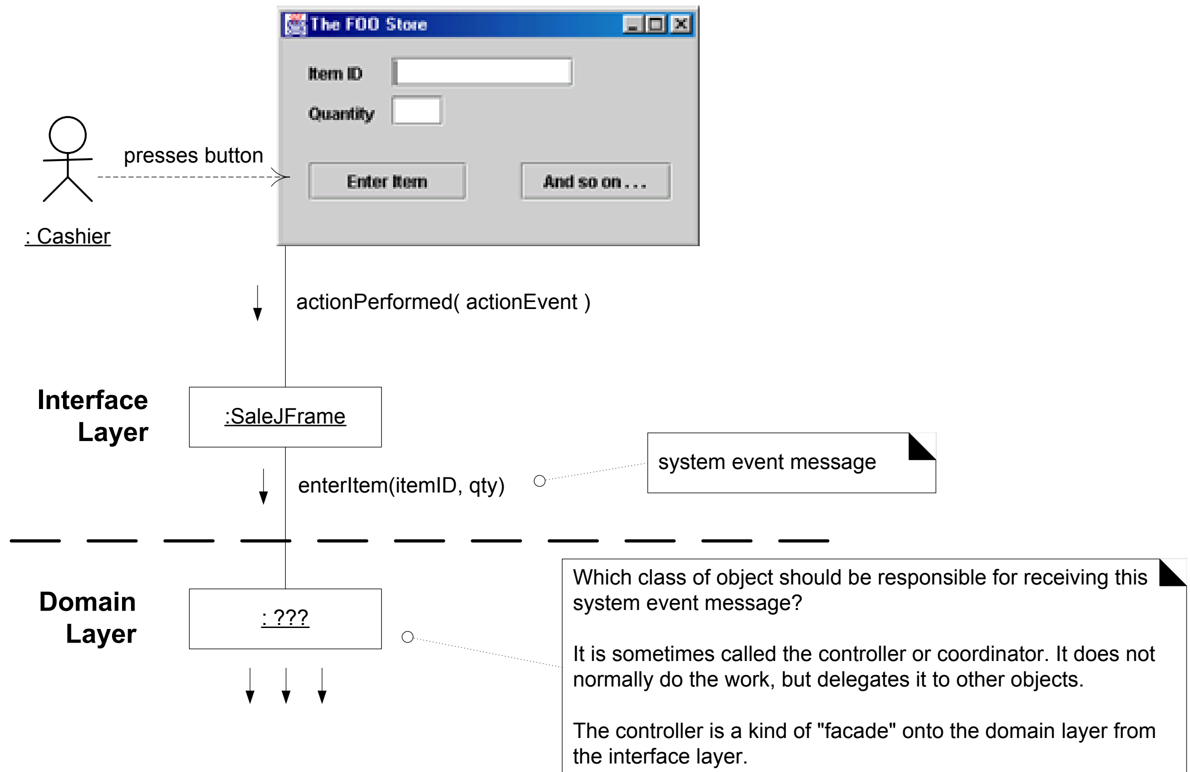

Figure 16.14 Controller for enterltem?

By the Controller pattern, here are some choices:

represents the overall "system," device, or subsystem	*Register, POSSystem*
represents a receiver or handler of all system events of a use case scenario	*ProcessSaleHandler, ProcessSaleSestsion*

In terms of interaction diagrams, it means that one of the examples in Figure 16.15 may be useful.

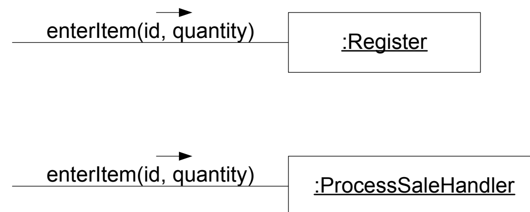

Figure 16.15 Controller choices.

The choice of which of these classes is the most appropriate controller is influenced by other factors, which the following section explores.

During design, the system operations identified during system behavior analysis are assigned to one or more controller classes, such *as Register,* as shown in Figure 16.16.

Discussion Systems receive external input events, typically involving a GUI operated by a person. Other mediums of input include external messages such as in a call processing telecommunications switch, or signals from sensors such as in process control systems.

In all cases, if an object design is used, some handler for these events must be chosen. The Controller pattern provides guidance for generally accepted, suitable choices. As illustrated in Figure 16.14, the controller is a kind of facade into the domain layer from the interface layer.

It is often desirable to use the same controller class for all the system events of one use case so that it is possible to maintain information about the state of the use case in the controller. Such information is useful, for example, to identify out-of-sequence system events (for example, a *makePayment* operation before an *endSale* operation). Different controllers may be used for different use cases.

A common defect in the design of controllers is to give them too much responsibility.

> Normally, a controller should *delegate* to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself.

Please see the "Issues and Solutions" section later for elaboration.

The first category of controller is a facade controller representing the overall system, device, or a subsystem. The idea is to choose some class name that suggests a cover, or facade, over the other layers of the application, and that provides the main point of service calls from the UI layer down to other layers. It

could be an abstraction of the overall physical unit, such as a *Register*[4], *TelecommSwitch, Phone,* or *Robot; a* class representing the entire software system, such as *POSSystem,* or any other concept which the designer chooses to represent the overall system or a subsystem, even, for example, *ChessGame* if it was game software.

Facade controllers are suitable when there are not "too many" system events, or it is not possible for the user interface (UI) to redirect system event messages to alternating controllers, such as in a message processing system.

If a use-case controller is chosen, then there is a different controller for each use case. Note that this is not a domain object; it is an artificial construct to support the system (a *Pure Fabrication* in terms of the GRASP patterns). For example, if the NextGen application contains use cases such as *Process Sale* and *Handle Returns,* then there may be a *ProcessSaleHandler* class and so forth.

When should you choose a use-case controller? It is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling, typically when the facade controller is becoming "bloated" with excessive responsibilities. A use-case controller is a good choice when there are many system events across different processes; it factors their handling into manageable separate classes, and also provides a basis for knowing and reasoning about the state of the current scenario in progress.

In the UP and Jacobson's older Objectory method [Jacobson92], there are the (optional) concepts of boundary, control, and entity classes. **Boundary objects** are abstractions of the interfaces, **entity objects** are the application-independent (and typically persistent) domain software objects, and **control objects** are use case handlers as described in this Controller pattern.

A important corollary of the Controller pattern is that interface objects (for example, window objects or widgets) and the presentation layer should not have responsibility for fulfilling system events. In other words, system operations should be handled in the application logic or domain layers of objects rather than in the interface layer of a system. See the "Issues and Solutions" section for an example.

The Controller object is typically a client-side object within the same process as the UI (for example, an application with a Java Swing GUI), and so is not exactly applicable when the UI is a Web client in a browser, and there is server-side software involved. In the latter case, there are various common patterns of handling the system events that are strongly influenced by the chosen server-side technical framework, such as Java servlets. Nevertheless, it is a common idiom to create server-side use-case controllers with either a servlet for each use case or an Enterprise JavaBeans (EJB) session bean for each use case. The

4. Various terms are used for a physical POS unit, including register, point-of-sale terminal (POST), and so forth. Over time, "register" has come to embody the notion of both a physical unit, and the logical abstraction of the thing that registers sales and payments.

server-side session object represents a "session" of interaction with an external actor.

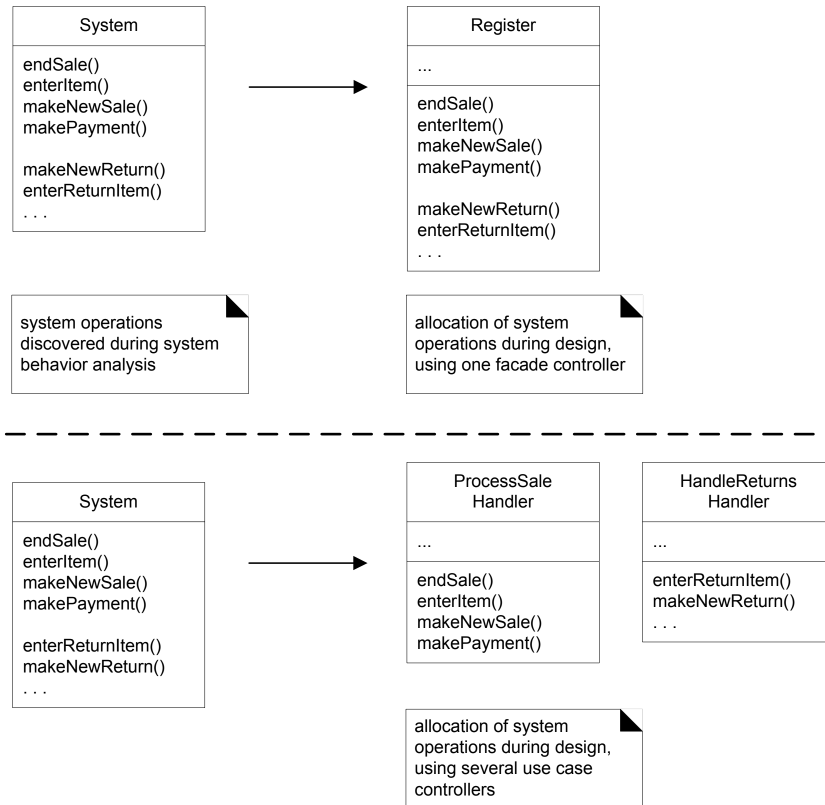

Figure 16.16 Allocation of system operations.

If the UI is not a web client (for example, it is a Swing or Windows GUI), but the application calls on remote services, it is still common to use the Controller pattern. The UI forwards the request to the local client-side Controller, and the Controller may forward all or part of the request handling on to remote services. This design lowers the coupling of the UI to remote services, and makes it easier, for example, to provide the services either locally or remotely, through the indirection of the client-side Controller.

To summarize, the Controller receives the service requests from the UI layer and coordinates their fulfillment, usually by delegation to other objects.

Benefits

- *Increased potential for reuse,* and *pluggable interfaces*—It ensures that application logic is *not* handled in the interface layer. The responsibilities of a controller could technically be handled in an interface object, but the implication of such a design is that program code and logic related the fulfillment of application logic would be embedded in interface or window objects. An interface-as-controller design reduces the opportunity to reuse logic in future applications, since it is bound to a particular interface (for example, window-like objects) that is seldom applicable in other applications. By contrast, delegating a system operation responsibility to a controller supports the reuse of the logic in future applications. And since the application logic is not bound to the interface layer, it can be replaced with a different interface.
- *Reason about the state of the use case*—It is sometimes necessary to ensure that system operations occur in a legal sequence, or to be able to reason about the current state of activity and operations within the use case that is underway. For example, it may be necessary to guarantee that the *makePay-ment* operation can not occur until the *endSale* operation has occurred. If so, this state information needs to be captured somewhere; the controller is one reasonable choice, especially if the same controller is used throughout the use case (which is recommended).

Issues and Solutions

Bloated Controllers

Poorly designed, a controller class will have low cohesion—unfocused and handling too many areas of responsibility; this is called a bloated controller. Signs of bloating include:

- There is only a *single* controller class receiving *all* system events in the sys tem, and there are many of them. This sometimes happens if a facade con troller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the sys tem event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.
- A controller has many attributes, and maintains significant information about the system or domain, which should have been distributed to other objects, or duplicates information found elsewhere.

There are several cures to a bloated controller, including:

1. Add more controllers—a system does not have to have only one. Instead of facade controllers, use use-case controllers. For example, consider an application with many system events, such as an airline reservation system.

It may contain the following controllers:

Use-case controllers
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

2. Design the controller so that it primarily delegates the fulfillment of each system operation responsibility on to other objects.

Interface Layer Does Not Handle System Events

To reiterate: an important corollary of the Controller pattern is that interface objects (for example, window objects) and the interface layer should not have responsibility for handling system events. As an example, consider a design in Java that uses a *JFrame* to display the information.

Assume the NextGen application has a window that displays sale information and captures cashier operations. Using the Controller pattern, Figure 16.17 illustrates an acceptable relationship between the *JFrame* and Controller and other objects in a portion of the POS system (with simplifications).

Notice that the *SaleJFrame* class—part of the interface layer—passes the *enter-Item* message to the *Register* object. It did not get involved in processing the operation or deciding how to handle it; the window only delegated it to another layer.

Assigning the responsibility for system operations to objects in the application or domain layer—using the Controller pattern rather than the interface layer supports increased reuse potential. If an interface layer object (like the *SaleJ-Frame)* handles a system operation—which represents part of a business process—then business process logic would be contained in an interface (for example, window-like) object, which has low opportunity for reuse because of its coupling to a particular interface and application.

Consequently, the design in Figure 16.18 is undesirable.

Placing system operation responsibility in a domain object controller makes it easier to reuse the program logic supporting the associated business process in future applications. It also makes it easier to unplug the interface layer and use a different interface framework or technology, or to run the system in an offline "batch" mode.