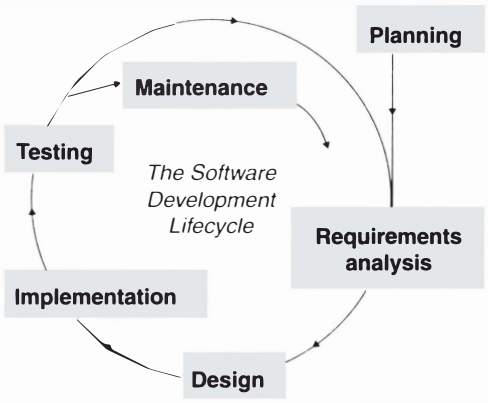
18

Software Architecture



- How do you classify software architectures?
- What are data flow architectures?
- What are three-tier architectures and their generalizations?
- What makes database-centric systems a separate type of architecture?
- What are service-oriented architectures?
- What IEEE standards are there for expressing designs?
- What do real-world architectures look like?

Figure 18.1 The context and learning goals for this chapter

This part of the book, concerned with design, began by describing design goals and principles and then described patterns of design that recur throughout. This chapter describes design at the high level, and the chapter that follows at the detailed level.

A *software architecture* describes the overall components of an application and how they relate to each other. Its design goals, as discussed in Chapter 15, include sufficiency, understandability, modularity, high cohesion, low coupling, robustness, flexibility, reusability, efficiency, and reliability. For a given software

- Dataflow architectures
  - Pipes and filters
  - Batch sequential
- Independent components
  - Client-server systems
  - Parallel communicating processes
  - Event systems
  - Service-oriented (added)
- Virtual machines
  - Interpreters
  - Rule-based systems
- Repository architectures
  - Databases
  - Hypertext systems
  - Blackboards
- Layered architectures

Figure 18.2 Shaw and Garlan's categorization of software architectures

Source: Shaw, M.G. and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline," Prentice Hall, 1996.

development project, there may be several possible appropriate architectures, and selecting one depends upon the goals that one wants to emphasize.

Flexibility, to choose one of these qualities, is a key goal of many architectures—the ability to accommodate new features. This usually involves introducing abstraction into the process. For example, we might want the architecture for the Encounter video game case study to support not just this particular game, but any role-playing video game.

Attaining one desirable design property may entail a trade-off against others. For example, a designer who uses abstractions to obtain a flexible architecture may make it harder to understand.

18.1 A CATEGORIZATION OF ARCHITECTURES

Shaw and Garlan [1] have classified software architectures in a useful manner. Their classification, somewhat adapted here, is shown in Figure 18.2. Section 18.3 explains these architectures. There is a wide variety of problems requiring software solutions, and there is a wide variety of architectures needed to deal with them. In most cases, the architecture is unique to the problem. Sometimes, one of the architectures identified by Shaw and Garlan matches the problem; in many cases they simply provide ideas on which to base the architecture. This is similar to architecture in house construction, in which classical and standard ideas provide inspiration for great architecture but are not simply copied.

18.2 SOFTWARE ARCHITECTURE ALTERNATIVES AND THEIR CLASS MODELS

The software architect develops a mental model of how the application is meant to work, often with five to seven components. The architect's mental model depends on the application in question, of course, but may benefit from architectures that others have developed in the past, just as a suspension bridge design benefits from the study of previously built suspension bridges. This section elaborates on the architectures classified by Shaw and Garlan. They categorize architectures as data flow, independent components, virtual machines, repository architectures, and layered architectures. Figure 18.2 summarizes these and their subcategories, and the rest of this section explains them. It also adds service-oriented architectures.

18.2.1 Data Flow Architectures

Some applications are best viewed as data flowing among processing units. Data flow diagrams (DFDs) illustrate such views. Each processing unit of the DFD is designed independently of the others. Data

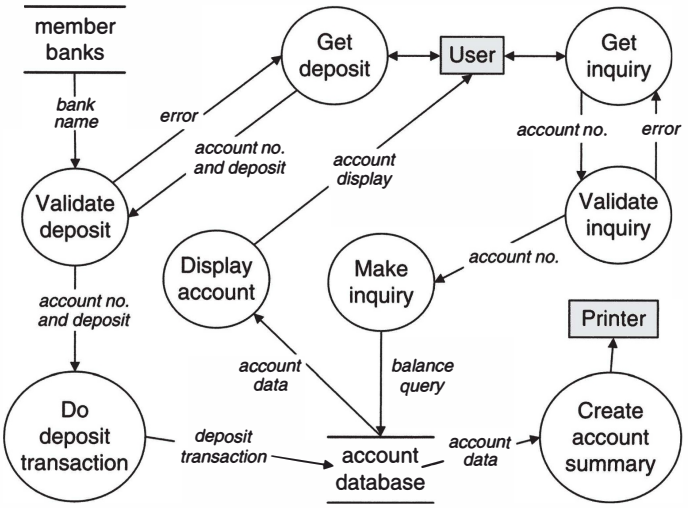


Figure 18.3 Partial data flow diagram for an ATM application

emanates from sources, such as the user, and eventually flows back to users, or into sinks such as account databases. The elements of the DFD notation were explained in Chapter 16. A banking application is shown in Figure 18.3.

Data flow from the user to a "Get deposit" process, which sends the account number and the deposit amount to a process designed to check these data for consistency. If they are consistent, the data may be sent to a process that creates a transaction, and so on. DFDs can be nested. For example, the "Create inquiry transaction" can itself be decomposed into a more detailed data flow diagram.

The functions of a data flow diagram may reside on more than one physical platform. Figure 18.4 shows one possible allocation.

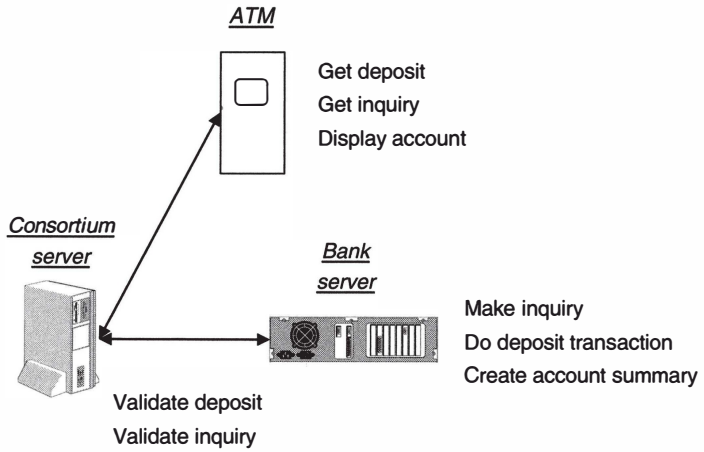


Figure 18.4 Platforms in data flow architectures, and an example

18.2.1.1 Pipe and Filter

One kind of data flow architecture, shown in Figure 18.5, is referred to as the *pipe and filter* architecture. In this kind the processing elements ("filters") accept streams as input (sequences of a uniform data element) at any time, and produce output streams. Each filter must be designed to be independent of the other filters. The architectural feature in Figure 18.5 is implemented by UNIX pipes, for example.

Pipe and filter architectures have the advantage of modularity. An example is shown in Figure 18.6. In it, the application maintains accounts as transactions arrive at random times from communication lines. The architecture includes a step for logging transactions in case of system failure. The withdraw function would have withdrawal input such as *JohnDoeAccountNum12345Amount$3500.00*, or just *JohnDoe12345$3500.00*'—that is, a character stream and bank address input such as *BankNum9876*. The processing elements, shown in ellipses, wait until all of the required input has arrived before performing their operation.

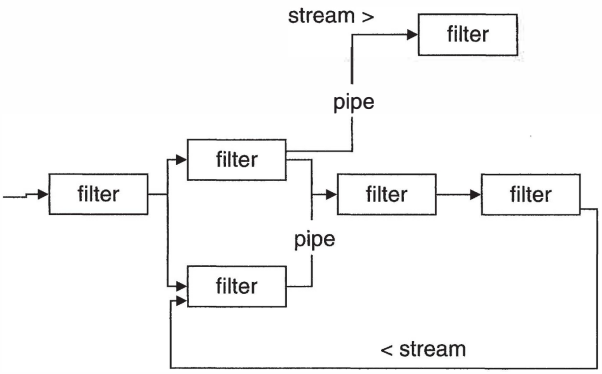


Figure 18.5 Pipe and filter architectures

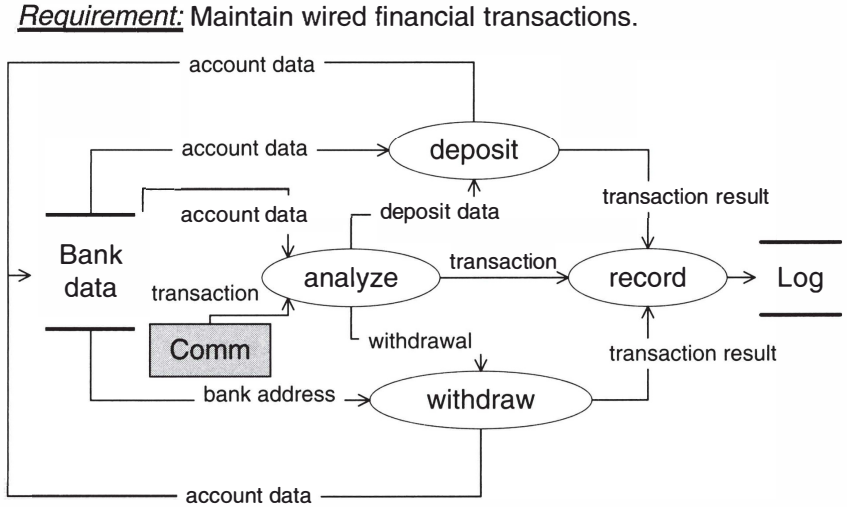


Figure 18.6 Example of a pipe-and-filter architecture, to maintain wired financial transaction

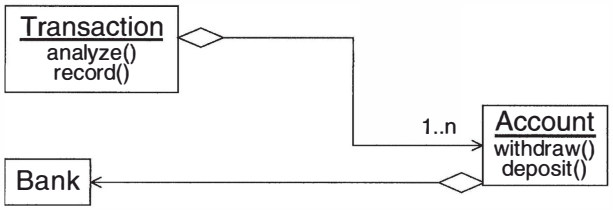


Figure 18.7 Obtaining a class model from a data flow architecture—bank account example

There is no absolutely uniform way to map data flow diagrams (DFDs) onto class models; however, functional units of the DFD can frequently map directly onto methods of classes, as shown in Figure 18.7.

The increasing use of distributed computing is accelerating the application of stream-oriented computing because remote function calling is often implemented by converting the call to a stream of characters. This is done in Web services, for example. These use serialization, which converts objects to XML character streams. In addition, I/O is often implemented using streams, and performing I/O in a language such as Java often amounts to a filtering process of the kind we have discussed.

18.2.1.2 Batch Sequential

In the special case where the processing elements are only given batches of data, the result is a *batch sequential* form of data flow. As an example, consider a banking application that computes the amount of money available for mortgage loans (secured by properties) and the amount available for unsecured loans. A data flow diagram (DFD) is suggested by Figure 18.8.

This DFD is batch sequential because the functions are executed using all of the input data of a given run, taken together. For example, we collect the funds available for mortgage loans by using all of the account data. This is in contrast with the transaction example in Figure 18.6, in which there are many "virtually continuous" transactions, each using selected data from their sources.

Figure 18.9 shows one mapping into a class model in which the functions of the data flow are realized as methods of the *Bank* class. The "batches" of processing are executed by running the relevant methods of this class.

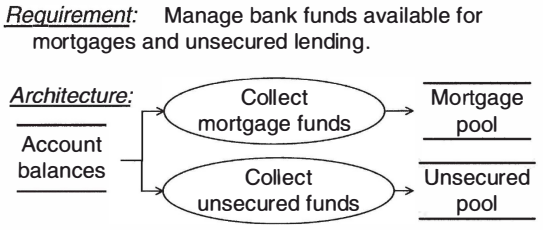


Figure 18.8 Example of a batch-sequential data flow architecture—creating a mortgage pool

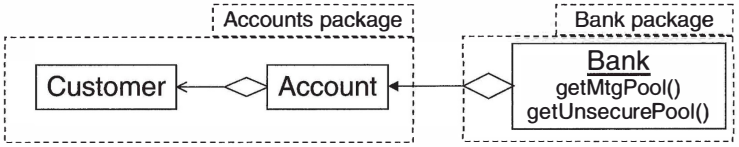


Figure 18.9 Class model for batch sequential data flow—creating a mortgage pool example

For decades, data flow has been the most common way of expressing architectures, and it is bound to be useful for some time to come. Engineers naturally think of data flowing from one processing "station" to the next and of processing taking place at each station. The disadvantages of data flow diagrams include the fact that they do not map very cleanly to code, whether object-oriented or not. An exception to this applies to specific data flow languages (some being actually graphic in nature), which are built around the very concept of data flow. We will use the data flow model once again when we discuss detailed design in Chapter 19.

18.2.2 Independent Components

The *independent components* architecture consists of components operating in parallel (at least in principle) and communicating with each other from time to time. An obvious instance of this can be found on the World Wide Web, where millions of servers and browsers operate continuously in parallel and periodically communicate with each other.

"Components" are portions of software that do not change and that do not require knowledge of the software using them. .NET assemblies and JavaBeans are example component technologies. Components satisfy guidelines aimed at making them self-contained. They use other components by aggregation, and generally interact with other components through events.

The case studies include a discussion of the Eclipse project. Eclipse is a development platform designed to accommodate *plug-ins*. These are independent components that can be created by developers for various purposes, and added to the platform without affecting existing functionality.

18.2.2.1 Tiered and Client-Server Architectures

In a *client-server* architecture, the server component serves the needs of the client upon request. Client-server relationships have the advantage of low coupling between the participating components. When more than one person performs implementation it is natural to parcel out a package of classes to each developer or group of developers, and developers typically require the services of classes for which others are responsible. In other words, developers' packages are often related as client and server. The problem is that these services are typically in varied states of readiness as the application is in the process of being built.

A server component acts more effectively when its interface is narrow. "Narrow" means that the interface (essentially a collection of functions) contains only necessary parts, is collected in one place, and is clearly defined. As explained in Chapter 17, the Facade design pattern establishes just such an interface to a package of classes. Facade regulates communication with the objects in its package by exposing only one object of the package to code using the package, hiding all of the other classes.

Client-server architectures were a steady feature of the 1980s and 1990s. Many of them replaced mainframe/terminal architectures. Client/server architectures have subsequently become more sophisticated and more varied. Some are now designed as three-tier architectures instead of the original two tiers (client and server). The third tier lies between the client and the server, providing a useful level of indirection. A common allocation of function is to design the GUI for the client, the database management system, or procedure management for the middle layer, and assorted application programs and/or the database itself for the third layer. The middle layer can be a common data "bus" that brokers communication. Alternatively, the middle layer can operate via a standard such as Microsoft's .NET *assemblies*. Finally, the World Wide Web can be considered a breed of client/server architecture in which "one server/tens of clients" is replaced by "one server/millions of clients."

18.2.2.2 The Parallel Communicating Processes Architecture

Another type of "independent component" architecture identified by Shaw and Garlan is named *parallel communicating processes*. This architecture is characterized by several processes, or threads, executing at the same time. In his classic book [2], Dijkstra showed that conceiving a process such as the combination of parallel parts can actually simplify designs. An example of this is a simulation of customers in a bank. Traditionally, many such simulations were designed without parallelism by storing and handling the events involved. Such designs can sometimes be simplified, however, if the movement of each customer is a separate process (e.g., a thread object in Java). Such a parallel communicating process design has the advantage that it matches more closely to the activities that it simulates. A good reference to parallel communicating processes in the Java context is [3]. The parallel processes may run on a single platform or on separate platforms, as illustrated in Figure 18.10.

Encounter uses this architectural parallel element by having the foreign character Freddie move independently from area to adjacent area while the game progresses. This thread "communicates" whenever Freddie finds himself in the same area as the player character.

A UML notation that expresses parallelism was discussed in Chapter 16. This notation, used in Figures 18.11 and 18.12, shows an architecture for a banking application designed to handle multiple transactions occurring simultaneously on automated teller machines (ATMs). This particular architecture allows the customer to initiate transactions without having to wait for completion, even when the transaction takes significant time. The application may, for example, inform a customer that funds he deposited will not be immediately available—that is, not wait for the completion of that process before making the announcement.

When customer *n* uses an ATM, an object for customer *n* is created (Step 1 in Figure 18.11). Customer *n* creates *session m* (2). Session *m* then retrieves an *Account* object such as customer *n* checking (3). The retrieval is performed asynchronously, and a thread, or parallel process, is created because it may take time. This allows the customer to carry out other business in the meantime. The method call retrieving the *Account* object is denoted by a stick arrow head indicating that the *Session* object proceeds in parallel with this construction, returning to the customer object. Since we are dealing at an architectural level, we are omitting details here such as showing the thread objects involved and showing how the *Session* objects know that this is its required

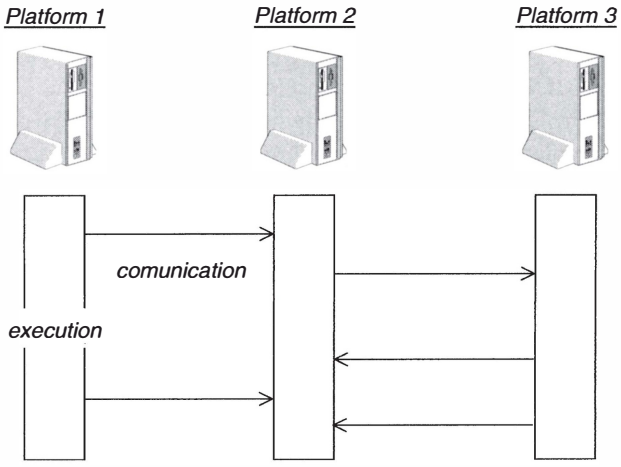


Figure 18.10 Platforms for communicating processors

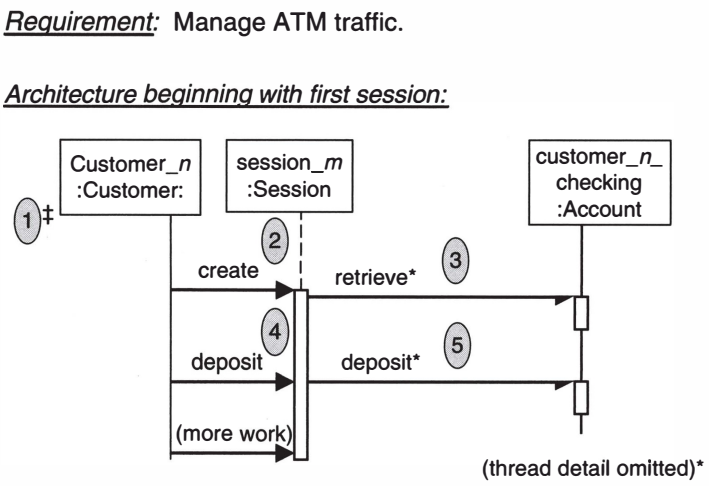


Figure 18.11 Example of parallel communicating processor architecture—managing ATM traffic, fragment of sequence diagram

call. The *Customer* object immediately performs a deposit transaction by sending a message to the *Session* object (4). The *Session* object executes the deposit by sending a deposit message asynchronously to the *Account* object, spawning a new thread (5). Other work can go on—including by other sessions—while the deposit is processed.

In parallel, other *Customer* objects such as customer *s* are creating and operating on other sessions such as session *k*. This is shown in Figure 18.12.

Figure 18.13 shows the beginning of a class model that handles this kind of architecture.

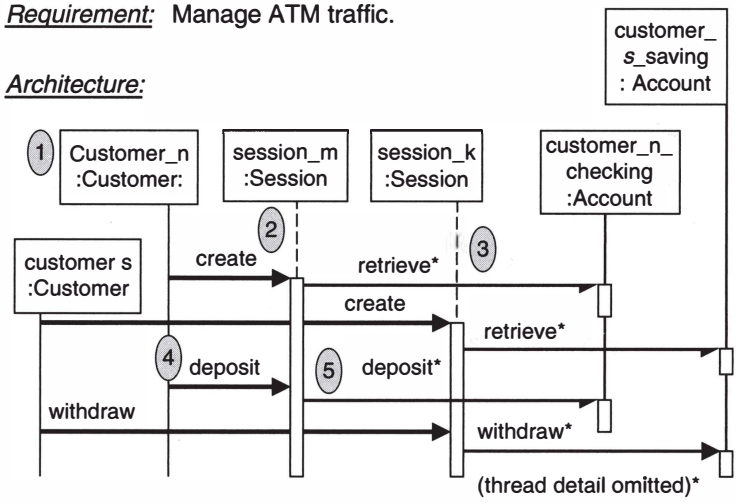


Figure 18.12 Example of parallel communicating processor architecture—managing ATM traffic—sequence diagram

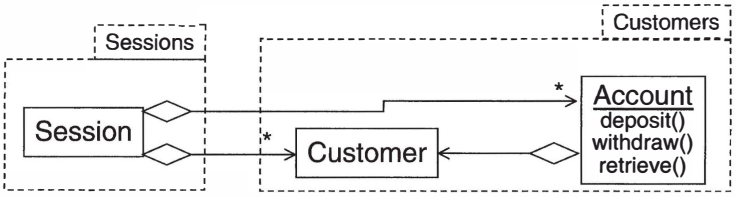


Figure 18.13 Example of parallel communicating processor architecture—managing ATM traffic—class model

18.2.2.3 Event Systems Architectures and the State Design Pattern

Let's turn to *event systems*, the third type of "independent component" architecture classified by Shaw and Garlan. This architecture views applications as a set of components, each of which waits until an event occurs that affects it. Many contemporary applications are event systems. A word processor, for example, waits for the user to click on an icon or menu item. It then reacts accordingly, by storing the file, enlarging fonts, and so on. Event systems are often fulfilled as state transition systems, which were introduced in Chapter 11.

When a system behaves by essentially transitioning among a set of states, the State design pattern should be considered for the design. This design pattern was explained in Chapter 17. For example, we have described the overall requirement for Encounter in terms of the state diagram in Figure 11.26 of Chapter 11. Encounter transitions among the *Setting up*, *Waiting*, *Setting qualities*, *Reporting*, and *Engaging* states, among others. Our design should capture this behavior effectively. It should also be capable of gracefully absorbing new states and action handling as the game becomes more complete, without disrupting the existing design. For these reasons, we will apply the State design pattern described in Chapter 17 to Encounter.

The State pattern solves the problem of how to use an object without having to know its state. In the context of Encounter we want to be able to write controlling code that handles mouse actions but does not reference the possible states that the game can be in, or the specific effects of the mouse actions. This makes it possible to add new game situations without disrupting this controlling code.

Figure 18.14 begins to show how the State design pattern can be used to handle the states and actions of Encounter. The framework class *RPGame* ("Role-playing game") has an attribute called *state*, which is of type *GameState*. The subtype of *state* (i.e., which subclass of *GameState* it belongs to) determines what happens when *handleEvent()* is called on an *RPGame* object. The code for *handleEvent()* in *RPGame* passes control to the *handleEvent()* function of state.

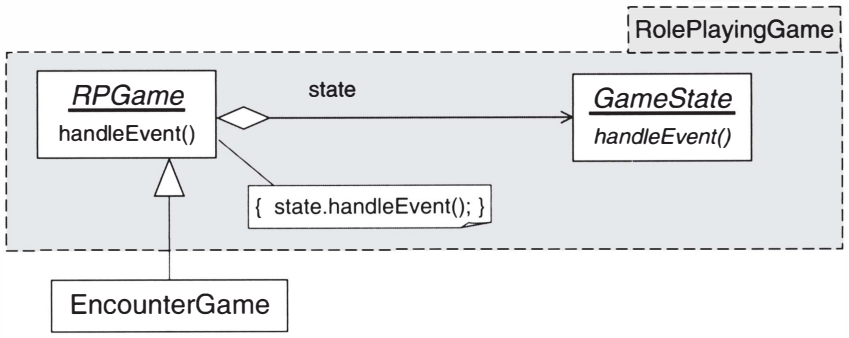


Figure 18.14 Beginning of the State design pattern applied to Encounter

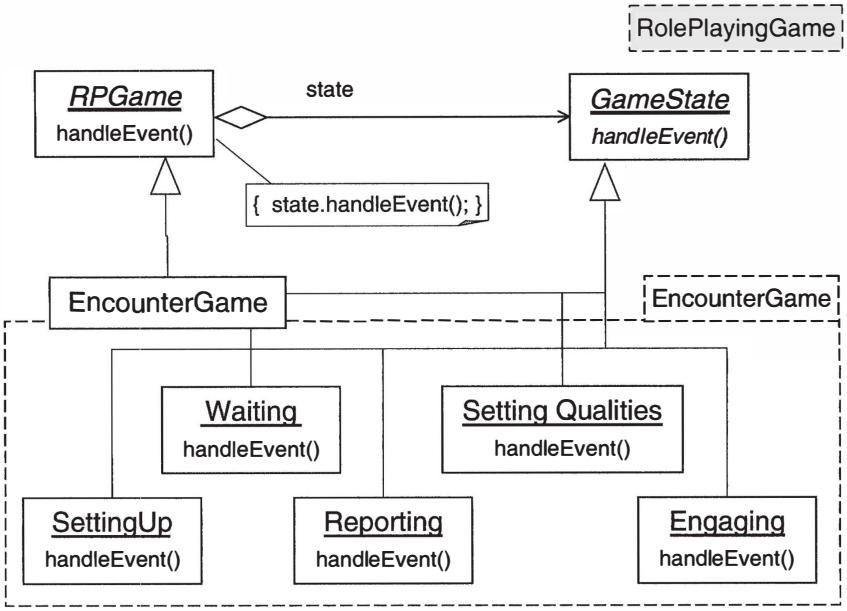


Figure 18.15 State design pattern applied to the Encounter video game

As shown in Figure 18.15, each subclass of *GameState* implements *handleEvent()* in its own manner. For example, if Encounter is in *SettingQualities* state, and the event is the arrival of a foreign character, then the window permitting the setting of character quality values disappears because this is what the method *handleEvent()* in *SettingQualities* is programmed to do. An additional consequence of this particular event/state combination is that Encounter transitions to the *Engaging* state, as required by the state-transition diagram. This transition is implemented by code such as

```
EncounterGame.setState(new Engaging());
```

The next time an event occurs in the game, the *handleEvent()* function of *Engaging* will execute, reflecting the fact that the game is now in the *Engaging* state.

18.2.3 Virtual Machines

A *virtual machine* architecture treats an application as a program written in a special-purpose language. Because an interpreter for such a language has to be built, this architecture pays off only if several "programs" are to be written in the language, generating several applications.

The implementation of a complete virtual machine requires the building of an interpreter. The interpretation requires us to execute an operation—let's call it *interpret()*—on a program written in our language. The interpretation of a primitive element alone (e.g., a CPU in the example of Chapter 17, where the parts of a "CPU" are not relevant) is generally simple (for the example, this could be simply to print "take

CPU out of its box"). The problem is how to execute an *interpret()* function when applied to a more complex "program." To do this, we may use the Interpreter design pattern.

Virtual machine architectures are advantageous if the application consists of the processing of complex entities, and if these entities, such as the orders in the example, are readily describable by a grammar.

An additional example requiring a virtual machine is an application that provides simple user-level programming of a special purpose language. A nonprogrammer user, for example, is capable of writing a script—a simple program—such as the following:

```
Balance checking / add excess to account + subtract deficit from saving;
Save report / c:Reports + standard headings + except replace "Ed''
by "Al'' PrintReport / standard headings
e-mail report to Jayne@xyz.net.
```

A virtual machine architecture parses and interprets such scripts. The idea of such architectures is illustrated in Figure 18.16.

18.2.4 Repository Architectures

An architecture built primarily around data is called a *repository* architecture. The most common of these are systems designed to perform transactions against a database. For example, an electric company maintains a database of customers that includes details about them, such as power usage each month, balance, payment history, repairs, and so on. Typical operations against this database are adding a new customer, crediting a payment, requesting a payment history, requesting a list of all customers more than three months in arrears, and so on. A typical design for this kind of repository architecture is shown in Figure 18.17. This figure mixes the flow of data between entities (solid lines) and control (dashed lines). "Control" means that one of the entities prompts the operation of the other—for example, turns it on and off.

Other examples of applications with repository architectures include interactive development environments (IDEs). IDEs apply processes such as editing and compiling to a database of source and object files.

Our Encounter example in its simplest form does not include a database. If, however, it were to grow into a game with many individual characters, then we might require a database rather than a flat file for storing the characters. This would certainly be true if we wanted to allow the user to call up statistics such as "list the characters with strength under 10," and so on. Structured Query Language (SQL) is a common way to express queries (see, for example, [4]).

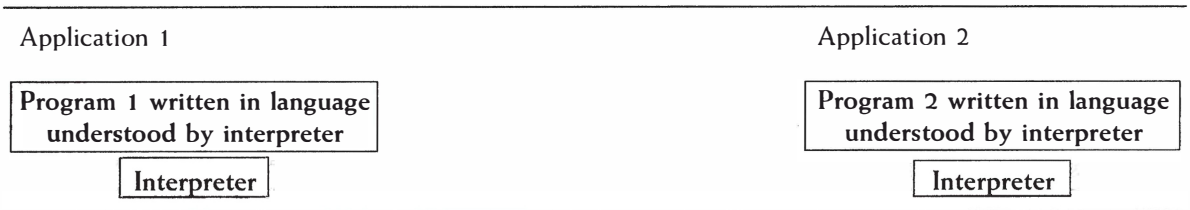


Figure 18.16 Virtual machine architectures—leveraging the interpreter concept to facilitate the implementation of multiple applications

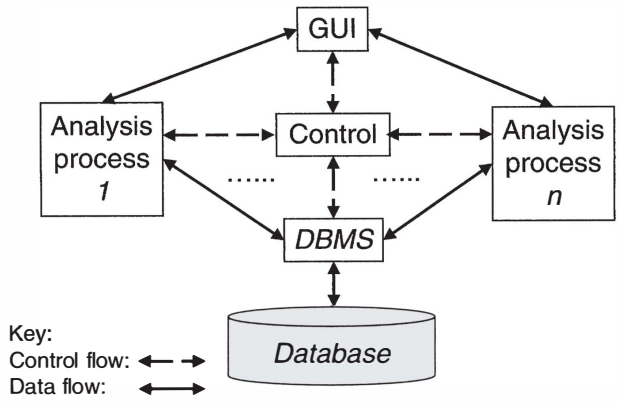


Figure 18.17 A typical repository architecture

Blackboard architectures, developed for artificial intelligence applications, are repositories that behave in accordance with posting rules. The reader is referred to [5] and [6] for a detailed treatment of blackboard architectures.

The final type of repository architectures we mention here is the hypertext architecture. The most common use of hypertext is on the Web. An application that manages the artifacts of a software engineering application is another example.

The word "repository" is often used in industry to denote an application that provides a unified view of a collection of databases (i.e., not just one). This relates to data warehouses. Repositories do not change the structure of these databases, but they allow uniform access to them. This is a special case of repository architectures as defined by Garlan and Shaw.

Repository architectures occupy a significant fraction of applications, since so many architectures make databases their core. When the processing is negligible compared to the formatting of data from the database, repository architectures are appropriate. On the other hand, the presence of a large database can sometimes mask the fact that a large amount of processing may drive the architecture. Ad hoc database programming (e.g., "stored procedures") can easily mushroom into messy applications, which perhaps should be conceived differently from the repository model.

18.2.5 Layered Architectures

An architectural *layer* is a coherent collection of software artifacts, typically a package of classes. In its common form, a layer uses at most one other layer, and is used by at most one other layer. Building applications layer by layer can greatly simplify the process. Some layers, such as frameworks, can serve several applications.

We have already seen the layered approach applied to the Encounter application, where classes in the Encounter packages inherit from classes in the framework packages. This is shown in Figure 18.18. The figure shows how we might organize the use of a 3-D graphics engine as a layer accessible from the Role-Playing Game layer.

Figure 18.19 shows an example of a layered architecture for an Ajax bank printing application. There are four layers in this architecture, and Figure 18.19 shows dependency in the reverse direction compared to Figure 18.18. The application layer, Ajax Bank Printing, has to do with printing and formatting. It is built upon (i.e., uses) the Accounts and the Ajax Bank Common Class layers. The latter are built upon a vendor-supplied layer, which contains general utilities such as sorting and searching. Typically, a layer is realized as a

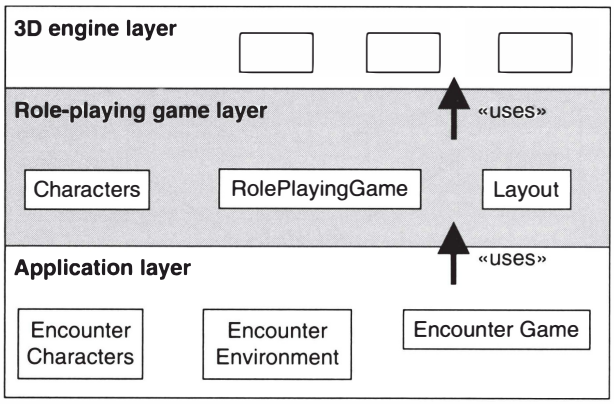


Figure 18.18 Layered architectures, and example of 3D engine and video game

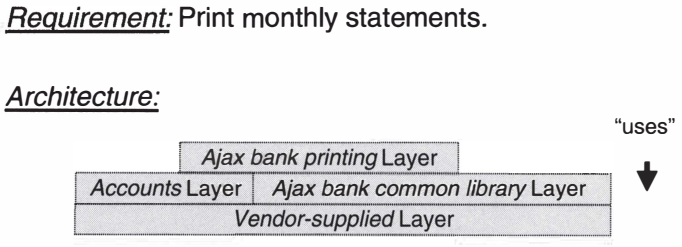


Figure 18.19 Example of layered architecture for Ajax Bank—highest level view

package of classes. For example, the Ajax common library comprises classes used throughout Ajax applications, and addresses such issues as the bank's logo and its regulations.

The "using" relationship can be inheritance, aggregation, or object reference. In the example, only aggregation is applied between layers, as shown in Figure 18.20.

18.2.6 Service-Oriented Architectures

Service-Oriented Architectures (SOAs) are gaining in usage. They are closely related to the idea of software as a service and cloud computing, and warrant inclusion with those discussed above. SOAs are combinations of *services*: components that provide functionality according to an interface specification. They differ from many other application architectures in that they describe a set of interoperable components that can be dynamically harnessed to create functionality—rather than as a way to create a single application.

SOAs are in the spirit of facade objects, and include Web services as a means of implementation. SOAs are not necessarily object-oriented. In the case of Web services there is no assurance of globally defined classes as we have provided for *Facade* in prior examples. For example, suppose that an SOA is for a business-to-business application concerning *orders*. In an SOA, we would not assume that a unique *Order* class is known to and usable by all service suppliers and consumers. Web services in particular deals with this by defining a schema for an *order* data structure, and referencing the schema when Web services involve orders. This uses a Web service capability known as *Web Service Description Language* and has the effect of making an *Order* class known to clients. This is summarized in Figures 18.21, 18.22, and 18.23.

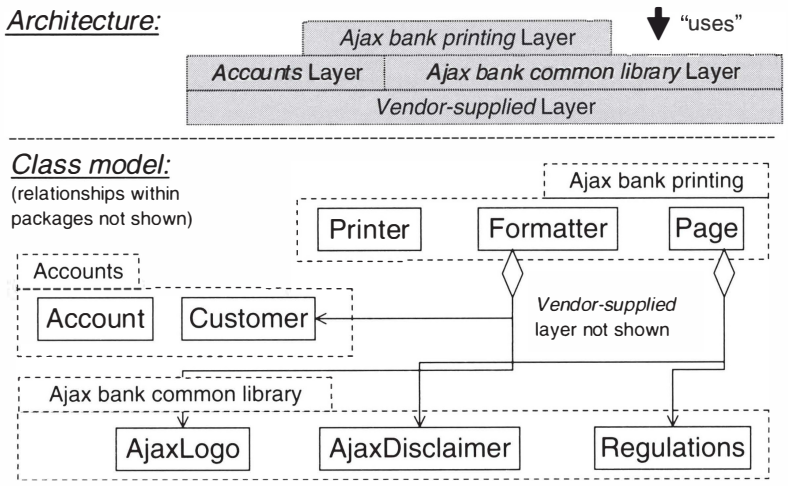


Figure 18.20 Layered architecture example using aggregation

Based on components that provide functionality according to an **interface** spec.
- Principally via Web services
- In the spirit of facade objects
- Not necessarily OO

Example: An application concerning *orders*.
- Wouldn't assume an *Order* class known to all
- Instead: Define an *order* **schema**; reference when Web services involve orders

---

Figure 18.21 Service-oriented architectures, 1 of 3

Service-oriented architectures envisage a network (mostly an Internet)-dominated environment in which applications (and parts of applications) are not permanently wedded to each other. Rather, SOAs seek to allow dynamic linking of services. For example, suppose that you want to write an application that orders stationery for a company. You would want the application to identify all qualified vendors, check prices and availability, call for bids and terms, select a vendor, and place the order. To do this, the application can't be permanently wedded to a set of vendors. For this reason, SOAs are built around a *registration* system. Figure 18.23 illustrates the four steps involved in publishing and accessing a service. "Querying" is like looking up a business in a telephone book. "Binding" means contacting the service in order to invoke it.

(The reference for Figures 18.21, 18.22, and 18.23 is [7].)

- "Fire and forget"
  - **Stateless** as much as possible
- **Extensible**
  - Additional functionality easily added
- **Discoverable**
- **Account for Quality of Service**
  - For example, security

Figure 18.22 Service-oriented architectures, 2 of 3

- "Fire and forget"
  - **Stateless** as much as possible
- **Extensible**
  - Additional functionality easily added
- **Discoverable**
- **Account for Quality of Service**
  - For example, security

Figure 18.23 Service-oriented architectures, 3 of 3

Service-oriented architectures frequently use a *business process layer*. This defines the components of the business such as the customer data base and business. Credit policies are examples of the latter. The *service interface layer* defines the services available that are based upon the business processes. It specifies functionality such as listing all customers in the database and checking a transaction for conformance to business rules. The *application layer* consists of applications built using the service interface layer. These points are shown in Figure 18.24.

18.2.7 Using Multiple Architectures within an Application

Applications typically use several subsidiary architectures within an overall architecture. Figure 18.25 shows how the framework for role-playing video games could use several of the architecture types listed by Garlan and Shaw. It could make sense, for example, to organize the *Artifacts* package as a database. The game characters could be viewed as parallel communicating processes. We will design the overall control of the game as an event-driven system.

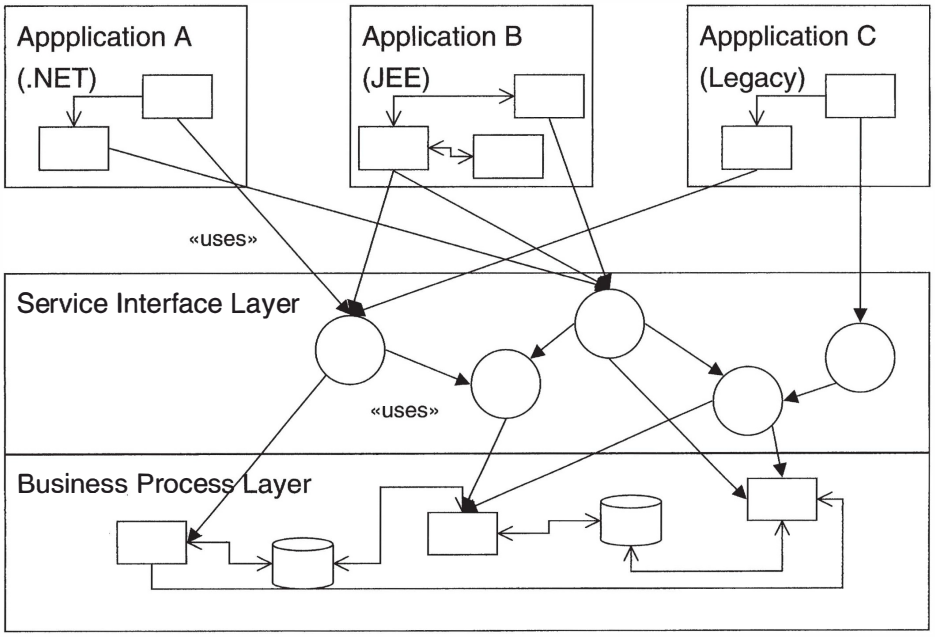


Figure 18.24 Layering for service-oriented architectures

Source: Adapted from Erl, Thomas, "Service-Oriented Architecture: Concepts, Technology, and Design," Prentice Hall, 2006.

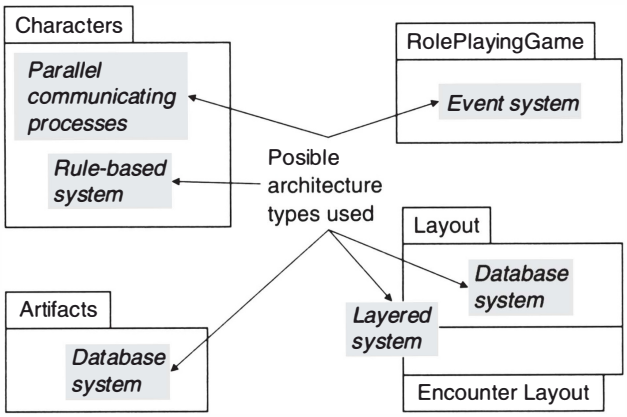


Figure 18.25 Example of the use of multiple subsidiary architectures—Encounter video game extension

18.3 TRADING OFF ARCHITECTURE ALTERNATIVES

Since the choice of a software architecture is so important, it is wise to create more than one alternative for consideration. As an example, we will create and compare two architectures for the video store application.

For the first candidate, we separate the application into three major parts: The "back-end" database, the middle part, which contains the business logic, and the GUIs. This is a *three-tier* architecture. It is often an appropriate choice when some or all of the tiers reside on physically separate platforms. In particular, if the GUIs are all on PCs, the middle layer on a server, and the databases controlled by a database management system, then three-tier architectures map neatly to separate hardware and software units. Note that there is no necessity that hardware decompositions be the same as software architectures; we may want a logical (conceptual) view of an application to be entirely independent of the hardware platforms hosting it. (These are the *physical* view vs. the *logical* view.) Applying three tiers to the video store application, we could obtain the architecture in Figure 18.26.

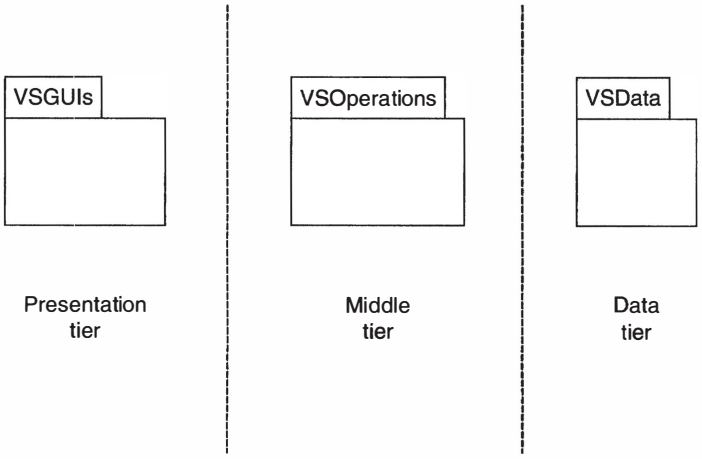


Figure 18.26 Three-tier architecture alternative

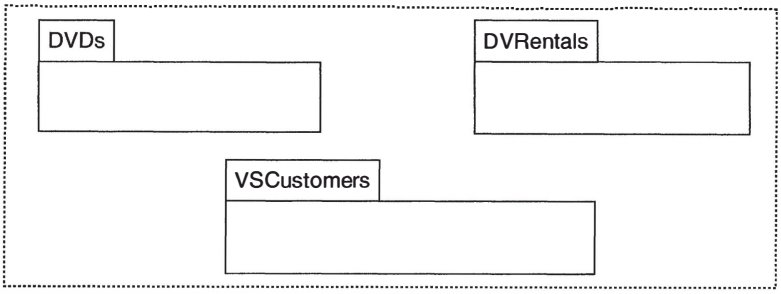


Figure 18.27 Alternative architecture for a video store application

The strength of this architecture is that it's easy to understand. Another strength is that since the GUI part is separate from the rental operations it can be changed without disturbing the latter. One weakness is the coupling between the *GUIs* package and the *VSOperations* package; there may be several GUI classes corresponding to the *Customer* class, for example. There is also coupling between the classes in the *VSData* package and the classes in *VSOperations*. A second architecture candidate is shown in Figure 18.27.

This architecture groups all of the classes pertaining to the videos in a package. The *Rentals* package contains classes that relate videos and customers. The *Customers* package contains the classes pertaining to customers, including associated GUIs. A third option would be to group all displays in a package. Figure 18.28 summarizes one opinion of these architecture alternatives.

18.4 TOOLS FOR ARCHITECTURES

Various computer-aided software engineering (CASE) tools are used to facilitate the software engineering process. Some tools represent classes and their relationships, such as Rational Rose by IBM Corporation. These tools facilitate the drafting of object models, linking them with the corresponding source code and sequence diagrams.

In selecting a modeling tool, a list of the requirements for the tool is drawn up using procedures similar to the requirements analysis process for software application development. Here is an example of some requirements for modeling tools.

- Essential: Facilitate drawing object models and sequence diagrams
- Essential: Create classes quickly

	Three-tier	Alternative
Understandable?	Yes	Yes
Flexible?	Yes: GUI easy to change	Yes: Basic building blocks easy to identify
Reusable?	Not very: Each layer is special to video store rentals	Yes: Easy to generalize to generic rentals
Easy to construct?	Perhaps	Yes: Clear potential to use Facade

Figure 18.28 A comparison of architectures—example

- Essential: Edit classes easily
- Desirable: Should cost no more than $X per user
- Desirable: Zoom into parts of the model
- Desirable: Possible to jump directly from the object model to the source code
- Optional: Reverse engineering available (i.e., create object models from source code)
- Optional: Use color coding for status of class implementation

Tool packages frequently try to span architecture, detailed design, and implementation. Various vendors are developing the capability to hyperlink from source code to documentation and vice versa. Implementation-oriented tools such as Javadoc can sometimes be useful to supplement the design process. Javadoc is useful for navigating packages because it provides an alphabetical listing of classes and the parent hierarchy of each class.

Interactive development environments (IDEs) are delivered with compilers and are widely used as partial modeling tools. Object-Oriented IDEs generally show inheritance in graphical form, and developers are frequently attracted to these tools because of their closeness to the compilation and debugging process. Eclipse, used as a successful software engineering case study itself in this book, is an example of a widely used IDE.

Component assembly tools create applications by dragging and dropping icons that represent processing elements. Java Bean environments are typical of these. Within such environments, beans (Java objects whose classes conform to the Java Beans standard) can be obtained from libraries, customized, and related to each other by means of events. The Java Beans standard was created with the express purpose of facilitating such simple assemblies by means of graphical tools.

A disadvantage of using modeling tools is the project's dependence on a third-party vendor. In addition to the complications of the application and the project itself, engineers must be concerned with the viability of the tool's vendor. Suppose that the vendor goes out of business or tool upgrades become too expensive: How will the project be affected? Despite these issues, the use of design and development tools has increased steadily. The right tools leverage productivity, and economic factors favor their usage.

18.5 IEEE STANDARDS FOR EXPRESSING DESIGNS

The IEEE Software Design Document (SDD) standard 1016-1998 provides guidelines for the documentation of design. The table of contents is shown in Figure 18.29. IEEE guidelines explain how the SDD could be written for various architectural styles, most of which are described above. The case study uses the IEEE standard with a few modifications to account for an emphasis on the object-oriented perspective. As shown in the figure, Sections 1 through 5 can be considered software architecture, and Section 6 can be considered the detailed design, to be covered in the next chapter.

18.6 EFFECTS OF ARCHITECTURE SELECTION ON THE PROJECT PLAN

Once an architecture has been selected, the schedule can be made more specific and more detailed. In particular, the order in which parts are built can be determined. For example, in the case study it makes sense to first develop the *Characters* framework package, followed by the specific *EncounterCharacters* application package. Since these packages will not be completed in the first iteration of the spiral, we name the

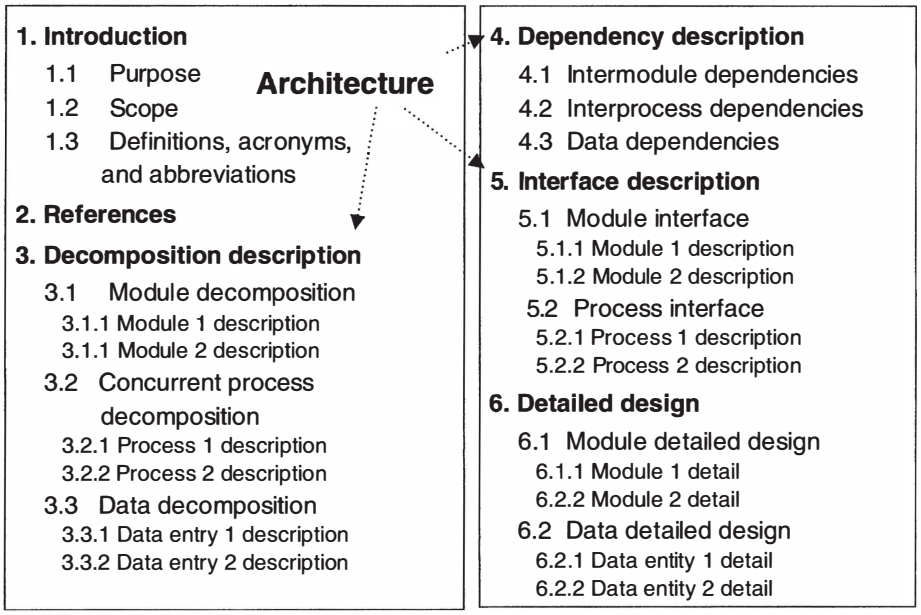


Figure 18.29 The architecture parts of IEEE 1016-1998—SDD table of contents

Source: IEEE Std 1016–1998.

corresponding tasks "Characters I" and "EncounterCharacters I." Arrows indicate dependence of packages on others. For example, "EncounterCharacters I" cannot be completed without the completion of "Characters I," as shown in Figure 18.30. The schedule shows that "Integration and Test I" cannot begin until all of the other tasks in Iteration I have been completed.

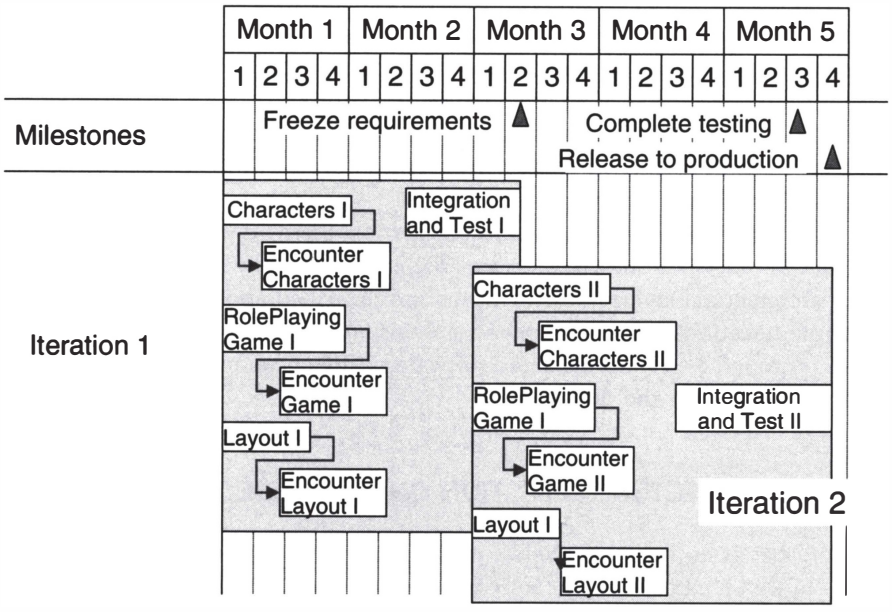


Figure 18.30 Schedule following architecture selection

18.7 CASE STUDY: PREPARING TO DESIGN ENCOUNTER (STUDENT PROJECT GUIDE CONTINUED)

This section describes a scenario of how the Encounter project team may have gone about the creation of the Encounter architecture.

18.7.1 Preparing

In accordance with the SPMP, Karen Peters was the design leader, with Ed Braun backing her up and inspecting all of the design work. Karen aimed to develop two thoroughly prepared alternative architectures for the Encounter project and bring them to the team's preliminary design review. She was determined to avoid unpleasant haggling over architectures that were devised by different engineers. She felt that ego rather than technical issues predominated in such cases. She had even worse memories of architecture "compromises" that were created to reconcile competing architectures. These frequently resulted in poor designs that everyone had to live and work with daily for months, if not years. On the other hand, Karen did not want to produce an architecture in isolation. She decided that she and Ed would select architecture candidates together, research them thoroughly, and present the choices to the team.

18.7.2 Selecting Architectures

Al Pruitt had been thinking about the design of the game application, and he gave Karen a sketch of an ad hoc GUI-driven architecture. He pointed out that it was simple and would be quick to implement. Ed and Karen reviewed Garlan and Shaw's classification of architectures to determine whether Encounter appeared to match any of the architecture alternatives.

They first asked whether Encounter could be described as the flow of data from one processing element to another. The data would have to be the positions of the game characters and/or their quality values. This view did not seem to them to match their conception of the game.

Next, they turned to Garlan and Shaw's "independent components" architectures, the first of which was "parallel communicating processes." To Karen this seemed to be a possible match because each game character could be considered one of the processes. Each could be run on a separate parallel thread, and these threads would communicate whenever the characters encountered each other. They noted this as a candidate architecture.

They considered "client server" next, but it was unclear to them what the "client" and "server" roles would be, so they dismissed this alternative. "Event systems," the next type listed, appeared to be a candidate architecture since the game responded to either user-initiated events, such as the pressing of an area hyperlink to enter an area, or to the arrival of the foreign character in the same area as the player character. They noted this as another candidate architecture.

Next, Ed and Karen considered "virtual machines," asking whether each game execution consisted essentially of the interpretation of a script. This did not appear to them to be the case.

They considered whether the game could be thought of as built around a repository of data (a "repository system"). The data could be the values of the characters and the status of the game. They decided that this might indeed be a possibility if there were many characters and many artifacts, because in that case, the manipulation of large amounts of data might predominate. Since Encounter was not to be data-centric, however, they rejected this candidate.

Finally, they considered a layered architecture. The question here was whether Encounter could be viewed as a series of class groupings with each grouping using one or two of the others. Karen felt that there would indeed be at least two useful layers: one for role-playing games in general, and one for Encounter. They made a note of this candidate architecture, and ended their consideration of Garlan and Shaw's options.

Now they listed the architecture candidates.

Al Pruitt's GUI-driven architecture

Parallel communicating processes

Event systems

Layered

They discussed which of these described the overall architecture and which were subsidiary to the overall architecture. To decide among these candidates they evaluated them in terms of the qualities described in this chapter. Their scheme gave 2 as the highest value and 0 as the lowest. In the case of close scores, the team would have questioned their scores closely. In addition, as they learned more about the application, they understood the need to revisit this table up to the time that they had to commit to architecture. One advantage of the table (Table 18.1) is that it allowed them to more easily reevaluate their reasoning.

Their conclusion was that layering was the primary architectural principle since there was a generic role-playing game layer and the Encounter layer itself. They envisaged the "event systems" architecture as subsidiary to the layers. They postponed a detailed discussion of parallel communicating processes for the game characters. Conceptually,

Table 18.1 Evaluation of architecture candidates by means of design qualities

Candidates	AI Pruitt's	Parallel communicating processes	Event systems	Layered
Qualities				
Sufficiency: handles the requirements	1	1	2	2
Understandability: can be understood by intended audience	0	2	1	2
Modularity: divided into well-defined parts	0	0	1	2
Cohesion: organized so like-minded elements are grouped together	1	0	2	2
Coupling: organized to minimize dependence between elements	0	1	0	1
Robustness: can deal with wide variety of input	1	0	2	1
Flexibility: can be readily modified to handle changes in requirements	1	0	1	1
Reusability: can use parts of the design and implementation in other applications	0	0	1	2
Information hiding: module internals are hidden from others	1	1	2	2
Efficiency: executes within acceptable time and space limits	1	2	0	1
Reliability:	0	1	1	2
TOTALS	**6**	**8**	**13**	**18**

their main architecture selection is reflected in Figure18.35

They decided to express the "event systems" architecture by means of states and transitions. Then they debated whether to use the State design pattern or a state/action table to describe the event/transitions, and decided to apply metrics to assist in choosing from the architectures under consideration, including Al Pruitt's.

They used e-mail to try to get agreement on the weighting of criteria for architectures (extension, change, etc.,—see Table 18.1 i.e., "Evaluation of architecture candidates by means of design qualities.") in advance of the meeting, without mentioning the architecture candidates themselves. They then e-mailed Al a draft of the comparison table in advance of the meeting to make sure that they had not missed any aspects of his proposal. Al pointed out that the choice of architecture was heavily dependent on the weighting, but was willing to accept the team's weighting. Karen and Ed drew up the spreadsheet Table 18.1, comparing Al Pruitt's architecture (alternative 2) and two others they had developed, and mailed it to the team members so that they would be prepared for the meeting.

18.7.3 Team Meeting (Preliminary Design Review)

At the meeting, Karen and Ed first confirmed agreement on the weighting of criteria. Because of their use of e-mail prior to the meeting, the team did not take long to iron out the few remaining inconsistencies. No mention was made yet of the architectures themselves. They presented the architecture alternatives to the team, showing the spreadsheet results. After some discussion and modification of their assessments, the team confirmed their selection of the layered architecture and the use of the State design pattern. Karen and Ed's thought process and presentation had been thorough. The team's discussion focused on how to improve the architecture selected. They solicited ideas for refining the architecture, but Karen did not try to rank the ideas or create a single refined version of the architecture at the meeting. She wanted to think through the suggestions offline.

18.7.4 Refining the Architecture

Karen and Ed were now faced with the task of decomposing each layer. They performed this by placing the two additional architectural elements in separate packages. In the role-playing game layer they formed a package for the state machine called *RolePlayingGame*. To handle the game characters, which move around in parallel, they created a *Characters* package. They also created a *GameEnvironment* package to contain classes describing the places in which the characters would move. Finally, they envisaged an *Artifacts* package for the future to describe the miscellaneous items such as shields and swords that would be involved. This package, postponed for future releases, would have a *Repository* architecture.

Their decomposition of the Encounter application layer was analogous since many of its classes had to inherit from the generic game level. They decided to create narrow access paths to the packages of this layer to prevent a situation in which any class could reference any other class. They felt that such unrestricted references would soon become impossible to manage during development and maintenance. To accomplish this narrow access they used the Facade design pattern for each application package. Ed had some reservations about doing this because it increased the amount of code the team would have to create and manage. Methods would not be called directly, he pointed out, but only through special methods of the facade objects, thereby increasing the total number of methods. It also introduced complications in that internal classes could not even be mentioned by objects external to the package (although their generic base classes could be), but Karen convinced him that the price was worth the benefit of having a clear interface for each package.

They obtained approval for their architecture at a subsequent team meeting.

18.7.5 Documenting the Architecture

Ed and Karen used Sections 1 through 5 of the IEEE standard 1016 to express the architecture in a Software Design Document (SDD). Since they had divided the application into two layers, one of which was slated for reuse, they decided to document the framework "role-playing game" layer in a separate SDD.

18.8 CASE STUDY: SOFTWARE DESIGN DOCUMENT FOR THE *ROLE-PLAYING VIDEO GAME* FRAMEWORK

We have two designs to describe. The first is that of the *Role-Playing Video Game* framework; the second is that of the Encounter role-playing game. The SDDs for both designs are split into two parts. The first parts, SDD Sections 1 through 5, shown below, consist of the architectural aspects of the designs. The second part, SDD section six, appearing at the end of Chapter 19, consists of the detailed designs. The dependence of Encounter on the framework is specified in the Encounter case study.

History of versions of this document:

x/yy/zzz K. Peters: initial draft

. . . .

x/yy/zzz K. Peters: revised, incorporating comments by R. Bostwick

x/yy/zzz K. Peters: moved details of classes to Section 3

1. Introduction

1.1 Purpose

This document describes the packages and classes of a framework for role-playing video games.

1.2 Scope

This framework covers essentials of role-playing game classes. Its main intention is to provide an example of a framework for educational purposes. It is not intended as a framework for commercial games since its size is kept small to facilitate learning.

1.3 Definitions, Acronyms, and Abbreviations

Framework: a collection of interrelated classes used, via inheritance or aggregation, to produce families of applications

RPG, Role-playing game: a video game in which characters interact in a manner that depends on their characteristics and their environment

2. References

Software Engineering: An Object-Oriented Perspective, by E. J. Braude (Wiley, 2001).

UML: The Unified Modeling Language User Guide, by G. Booch, J. Rumbaugh, and I. Jacobson (Addison-Wesley.

IEEE standard 1016-1987 (reaffirmed 1993) guidelines for generating a Software Design Document.

3. Decomposition Description

> Note to the Student:
> This section specifies how the framework classes for role-playing video games are to be grouped. This reflects the top-level decomposition: the detailed decomposition into methods, for example, is left for the detailed design (see case study at the end of the next chapter).

3.1 Module Decomposition

> This section shows the decomposition then explains each part in a subsection.

The framework consists of the *RolePlayingGame*, *Characters*, *Artifacts*, and *Layout* packages. These are decomposed into the classes shown in Figure 18.31. The classes in these packages are explained below. Unless otherwise stated, all classes in these packages are public. As indicated by the (UML) italics notation, all of the framework classes are abstract.

3.1.1 RolePlayingGame Package

This package is designed as a state-transition machine. The concept is that a role-playing game is always in one of several states. This package makes it possible to describe the possible states of the game and the actions that can take place in response to events. It implements the State design pattern (see [8]). The state of the game is encapsulated

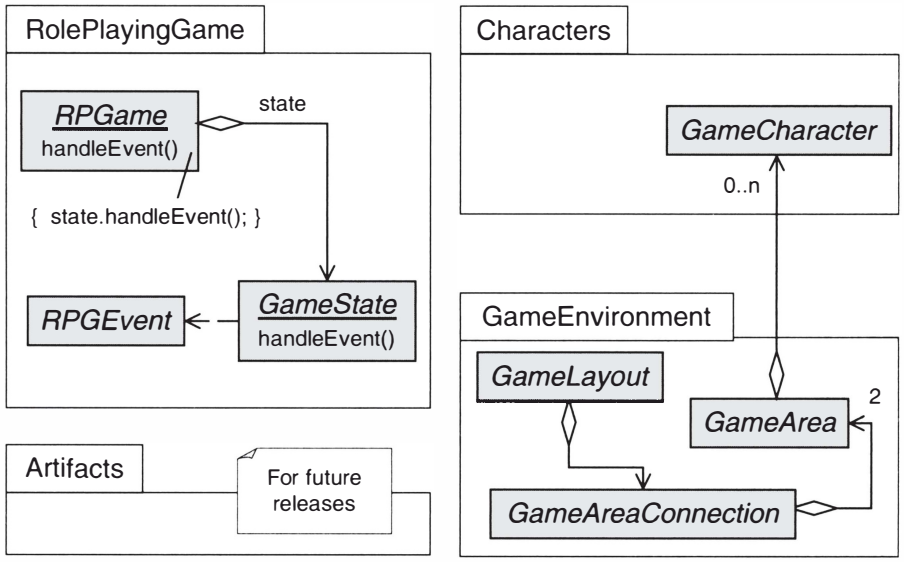


Figure 18.31 RPG framework for role-playing video games

(represented) by the particular *GameState* object aggregated by the (single) *RPGame* object. This aggregated object is named state. In other words, state is an attribute of *RPGame* of type *GameState*.

The function *handleEvent()* of *RPGame* is called to handle each event occurring on the monitor (mouse clicks, etc.). It executes by calling the *handleEvent()* function of state. The applicable version of *handleEvent()* depends on the particular subclass of *GameState* that state belongs to.

3.1.2 Characters Package

> It may seem strange to have a package containing just one class, but most artifacts in software design have a tendency to grow. Even if the pacakge does not grow, this does not disqualify its usefulness. For another example of a package with just one class, see *java.applet*, whose only class is *Applet* (but it also contains a few interfaces).

This package contains the *GameCharacter* class, which describes the characters of the game.

3.1.3 GameEnvironment Package

This package describes the physical environment of the game. The class *GameLayout* aggregates connection objects. Each connection object aggregates the pair of *GameArea* objects that it connects. This architecture allows for multiple connections between two areas. Each *GameArea* object aggregates the game characters that it contains (if any) and can detect encounters among characters.

3.1.4 Artifacts Package (Not Implemented—for Future Releases)

This package is intended to store elements to be located in areas, such as trees or tables, and entities possessed by characters, such as shields and briefcases.

3.2 Concurrent Process Decomposition

The framework does not involve concurrent processes.

4. Dependency Description

> This section describes all the ways in which the modules depend on each other.

The only dependency among the framework modules is the aggregation by *GameArea* of *GameCharacter*.

5. Interface Description

All classes in these packages are public, and thus the interfaces consist of all of the methods in their classes.

18.9 CASE STUDY: SOFTWARE DESIGN DOCUMENT FOR ENCOUNTER (USES THE FRAMEWORK)

History of versions of this document:

x/yy/zzz K. Peters: initial draft

. . . .

x/yy/zzz K. Peters: added decomposition by use case model and state model

1. Introduction

1.1 Purpose

This document describes the design of the Encounter role-playing game.

1.2 Scope

This design is for the prototype version of Encounter, which is a demonstration of architecture, detailed design, and documentation techniques. The architecture is intended as the basis for interesting versions in the future. This description excludes the framework classes, whose design is provided in the SDD entitled "Role-Playing Game Architecture Framework."

1.3 Definitions, Acronyms, and Abbreviations

None

2. References

"Role-Playing Game Architecture Framework," section in *Software Engineering: An Object-Oriented Perspective*, by E. J. Braude (Wiley, 2001).

UML: The Unified Modeling Language User Guide, by G. Booch, J. Rumbaugh, and I. Jacobson (Addison-Wesley.

IEEE standard 1016-1987 (reaffirmed 1993) guidelines for generating a Software Design Document.

3. Decomposition Description

The Encounter architecture is described using three models: use case, class (object) model, and state. In addition, the relationship between the domain packages of Encounter and the framework described in the SDD entitled "Role-Playing Game Architecture Framework" will be shown.

> The IEEE standard is extended using Sections 3.4 and 3.5 in order to describe these models. Recall that the other possible model is data flow, which we have not considered useful in this case. In the particular case of this video game, we chose to use the state description as part of the requirement as well as the design.

3.1 Module Decomposition (Object Model)

> This section should not duplicate the "detailed design" section described in the next chapter. We do not go into detail here regarding the contents of the packages.

The package architecture for Encounter is shown in Figure 18.32. The three packages are *EncounterGame*, *EncounterCharacters*, and *EncounterEnvironment*. These have facade classes *EncounterGame*, *EncounterCast*, and *EncounterEnvironment* respectively. The facade class of each package has exactly one instantiation, and is an interface through which all dealings with the package take place. The remaining classes are not accessible from outside the package. (See Section 17.7.1 in Chapter 17 and [1] for a complete description of the Facade design pattern.)

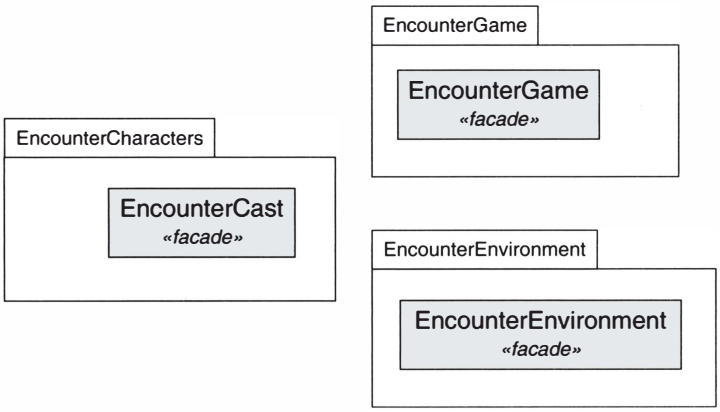


Figure 18.32 Architecture and modularization of Encounter game

3.1.1 EncounterGame Package

The *EncounterGame* package consists of the classes controlling the progress of the game as a whole. The package is designed to react to user actions (events).

3.1.2 EncounterCharacters Package

The *EncounterCharacters* package encompasses the characters involved in the game. These include character(s) under the control of the player together with the foreign characters.

3.1.3 EncounterEnvironment Package

The *EncounterEnvironment* package describes the physical layout of Encounter, including the areas and the connections between them. It does not include moveable items, if any.

3.2 Concurrent Process Decomposition

There are two concurrent processes in Encounter. The first is the main visible action of the game, in which the player manually moves the main character from area to adjacent area. The second consists of the movement of the foreign character from area to adjacent areas.

3.3 Data Decomposition

> Describes the structure of the data within the application

The data structures flowing among the packages are defined by the *Area*, *EncounterCharacter*, and *EncounterAreaConnection* classes.

3.4 State Model Decomposition

Encounter consists of the states shown in Figure 18.33.

> This state diagram was provided in the SRS, Section 2.1.1, where it was used to describe the requirements for Encounter. The remaining states mentioned in the requirements will be implemented in subsequent releases.

3.5 Use Case Model Decomposition

> This section is added to the IEEE specification, which does not address the use case concept. It has been added at the end of this section so as not to disturb the standard order.

Encounter consists of three use cases: *Initialize*, *Travel to Adjacent Area*, and *Encounter Foreign Character*. These use cases are explained in detail in the SRS, Section 2.2, and are detailed in sections later in this document.

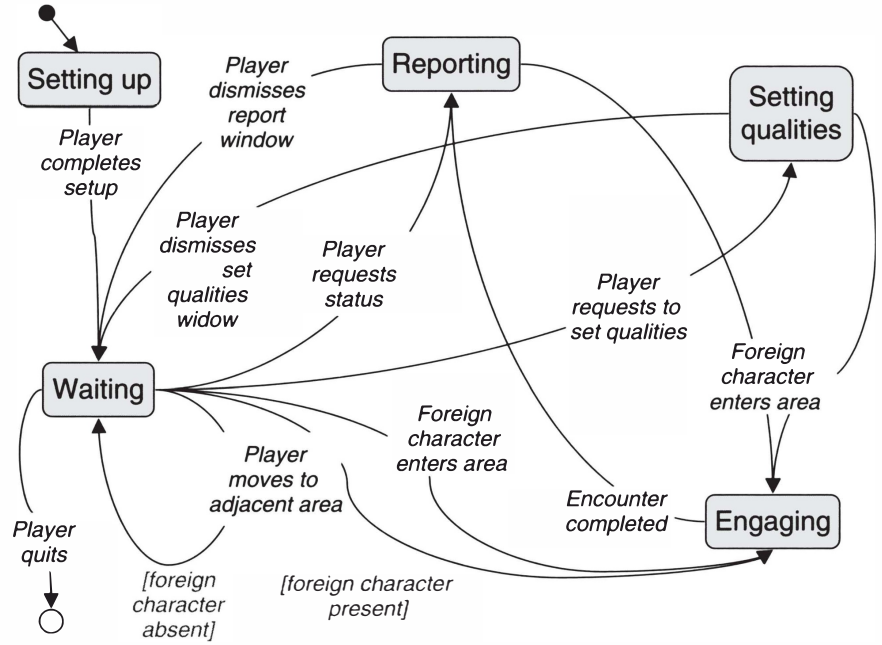


Figure 18.33 State-transition diagram for Encounter video game architecture

Details are given in the "detailed design" section.

There are no significant dependencies among the use cases.

4. Dependency Description

This section describes the dependencies for the various decompositions described in Section 3.

4.1 Intermodule Dependencies (Class Model)

The dependencies among package interfaces are shown in Figure 18.34.

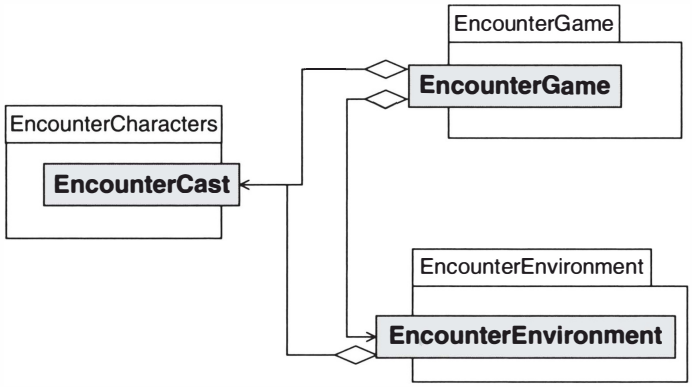


Figure 18.34 Architecture (modularization) of Encounter video game

The *EncounterGame* package depends on all of the other Encounter packages. The *EncounterEnvironment* package is designed to depend on the *EncounterCharacters* package. This is because the game's character interaction takes place only in the context of the environment. In particular, *Area* objects are responsible for determining the presence of the player's character together with the foreign character.

Dependencies among noninterface classes are explained later in this document.

> Such dependencies are detailed design specifications.

4.2 Interprocess Dependencies

When an engagement takes place, the process of moving the main character about and the process controlling the movement of the foreign characters interact.

4.3 Data Dependencies

The data structures flowing among the packages are defined by the classes, whose mutual dependencies are described in Section 6 of this document.

4.4 State Dependencies

Each state is related to the states into which the game can transition from it.

4.5 Layer Dependencies

The Encounter application depends on the *Role-Playing Game* framework as shown in Figure 18.35. Each application package uses exactly one framework package.

5. Interface Description

This section describes the interfaces for the object model. Note that several of the classes described are defined in the design description of the *Role-Playing Game* Framework.

5.1 Module Interfaces

> Describes the interaction among the packages

5.1.1 Interface to the EncounterGame Package

The interface to the *EncounterGame* package is provided by the the *EncounterGame* object of the *Encounter Game* facade class. It consists of the following:

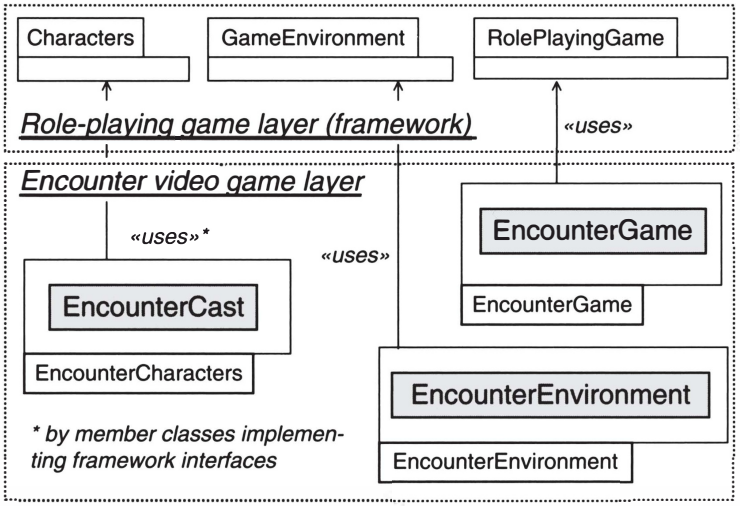


Figure 18.35 Framework-/-application dependencies

1. EncounterGame getTheEncounterGame()//gets the only instance
2. GameState getState()//current state of the *EncounterGame* instance
3. void setState(GameState)//– of the *EncounterGame* instance
4. //Any event affecting the single *EncounterGame* instance:
5. void handleEvent(AWTEvent)

5.1.2 Interface to the EncounterCharacters Package

The interface to the *EncounterCharacters* package is provided by the *theEncounterCast* object of the *EncounterCast* facade class. It consists of the following.

1. EncounterCast getTheEncounterCast()//gets the single instance
2. GameCharacter getThePlayerCharacter()//i.e., the unique character
3. GameCharacter getTheForeignCharacter()//the unique character
4. //Exchange quality values specific to the game area
5. void engagePlayerWithForeignCharacter (GameArea)

5.1.3 Interface to EncounterEnvironment Package

The interface to the *EncounterEnvironment* package is provided by the *EncounterEnvironment* object of *Encounter Environment Facade* class. It consists of the following:

1. EncounterEnvironment getTheEn-counter Environment()//gets the Facade object
2. GameArea getArea(String)
3. GameAreaConnection getAreaConnection(String)
4. void movePlayerTo(Area)
5. void moveForeignCharacterTo(Area) throws AreaNotAdjacentException
6. Image getNeighborhoodAreas(Area)//gets *Area* and areas one or two connections distant

5.2 Process Interface

We stated in Section 3.2 that there are two processes involved in Encounter. There is a significant design decision to be made in regard to the interface to the foreign character movement process, and we describe the result here. One option is to have the foreign character a thread, controlling itself. This has advantages, but requires this character either to know the environment—a disadvantage in terms of changing and expanding the game—or to be able to find out about the environment dynamically, which would be an elegant design but too ambitious for the scope of this case study. The architecture opts for another alternative, which is stated here.

5.2.1 Player Character Movement Process

The interfaces to the process that moves the player's character about the game consist of the graphical user interfaces specified in the SRS. The process reacts to events described in Section 3.4, which are handled by the *EncounterGame* package in accordance with its specifications, described later in this document.

5.2.2 Foreign Character Movement Process

The process of moving the foreign character is a separate process associated with and controlled by the *EncounterGame* singleton object. This process is controlled by the methods inherited from *java.lang. Thread*.

18.10 CASE STUDY: ARCHITECTURE OF ECLIPSE

Note to the Student:
The description that follows describes the architecture of Eclipse in a top-down fashion.

18.10.1 Scope

[This paragraph makes specific the scope of this document (the title "Architecture of Eclipse" seems clear enough until we read in this paragraph that it is qualified).]

As of April 2004, there were three Eclipse releases. We will provide an overall description of the architecture of release 3.

18.10.2 Overview

The Eclipse architecture has been described by Gamma and Beck [9] as shown in Figure 18.36.

> Figure 18.36 has just three parts—a manageable number to comprehend. Eclipse is a large and very complex product, however. Together with the brief explanations below, this decomposition is helpful. One has to search for a while to find a description like this by looking through http://www.eclipse.org.

- The *Platform* is the infrastructure of Eclipse and is independent of languages with which Eclipse can be used and independent of all plug-ins.
- The *Java Development Tools* utilize the *Platform* and is class model for a Java interactive development environment.
- The *Plug-In Development Environment* allows developers to create and add plug-ins. It uses the Java development tools.

18.10.3 Platform Module

The platform decomposes as shown in Figure 18.37.

The modules in Figure 18.37 are as follows:

- *Runtime* handles the available plug-ins at runtime.
- *Workspace* manages projects, which consist of files and folders.
- *Standard Widget Toolset* (SWT) provides graphics elements
- *JFace* is a set of UI frameworks using SWT, used for common UIs
- *Workbench* "defines the Eclipse UI paradigm. This involves editors, views, and perspectives."[1]

18.10.4 Java Development Tools

The design philosophy of Eclipse is to e able to build any language environment on top of the *Platform*. The Java Development Tools (JDT) module is the Java environment. It consists of the *Java Core* module. This is shown in Figure 18.38.

> We won't pursue this module or further parts of the Eclipse architecture in this case study.

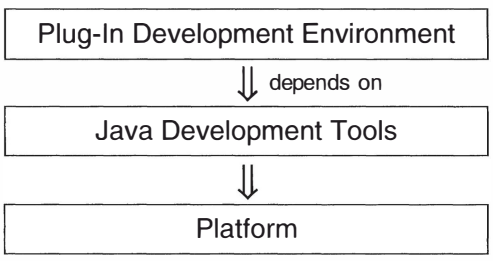


Figure 18.36 Very high-level architecture of Eclipse

Source: Adapted from Gamma, Erich, and Back, Kent. "Contributing to Eclipse: Principles, Patterns, Plug-Ins," Addison-Wesley, 2003, p.5.

[1] Gamma and Beck [9], p. 6

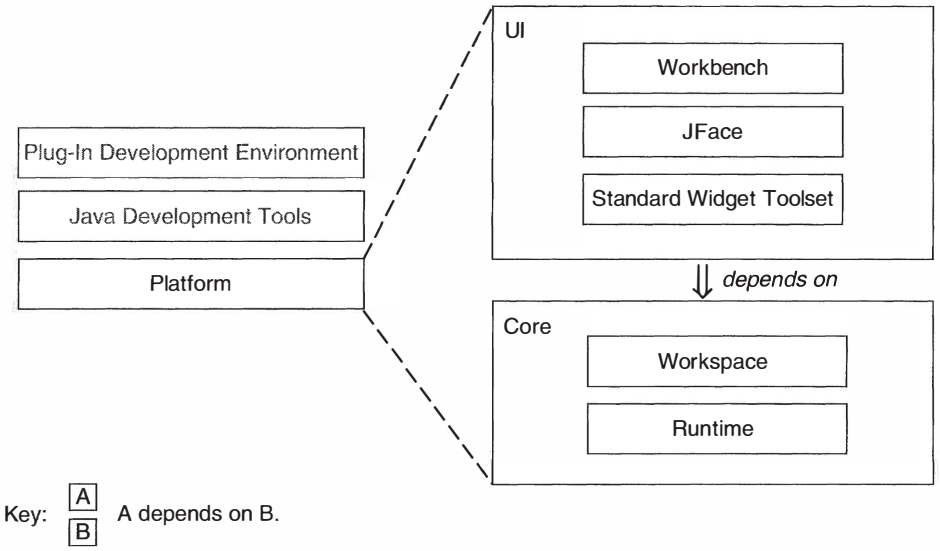


Figure 18.37 Architecture of Eclipse—platform

Source: Adapted from Gamma, Erich, and Back, Kent. "Contributing to Eclipse: Principles, Patterns, Plug-Ins," Addison-Wesley, 2003, p. 283.

18.11 CASE STUDY: OPENOFFICE ARCHITECTURE

Note to the Student:
It is not straightforward to locate the description of the OpenOffice architecture in a single, identifiable place.

The following is a useful white paper that sets out the design methodology used for OpenOffice, and much of this section is adapted or quoted from [10].

The architecture of the *OpenOffice.org* suite is designed to be platform-independent. It consists of four layers, as described in Figure 18.39.

We will not discuss the *StarOffice* API here. As indicated in Figure 18.39, however, three layers depend on it. The author has made local improvements in the writing of the original. This architecture description is at a very high level.

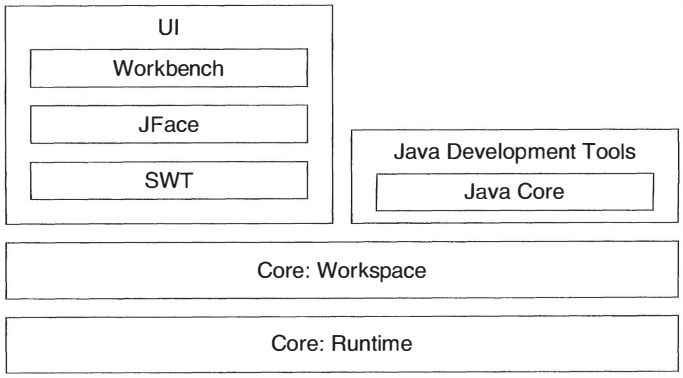


Figure 18.38 Architecture of Eclipse—the Java development tools module

Source: Adapted from Gamma, Erich, and Back, Kent. "Contributing to Eclipse: Principles, Patterns, and Plug-Ins," Addison-Wesley, 2003, p.282.

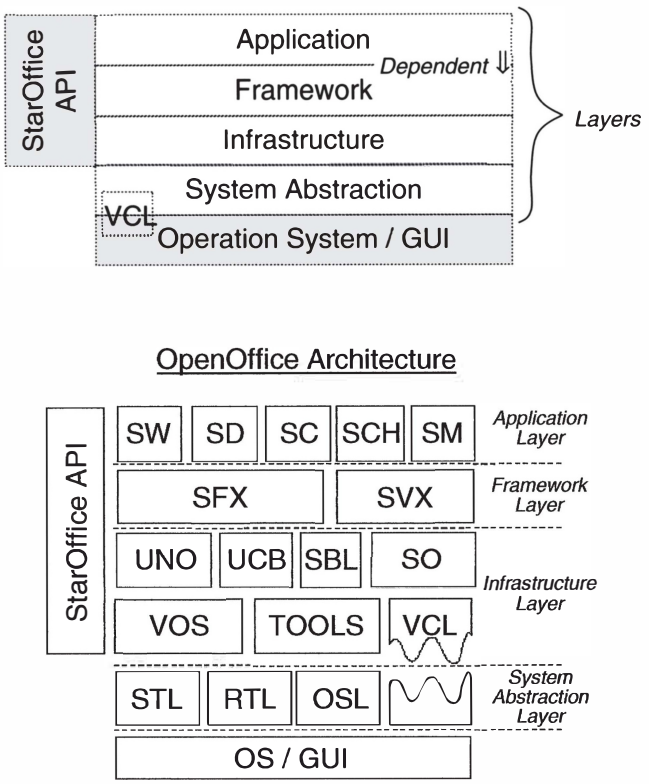


Figure 18.39 The architecture of OpenOffice

Source: Edited from OpenOffice, http://www.openoffice.org/white_papers/tech_overview/tech_overview.html#3.

The System Abstraction layer "encapsulates all system specific APIs and provides a consistent object-oriented API to access system resources in a platform independent manner." It provides a kind of single, virtual operating system on which to build OpenOffice.

The Infrastructure layer is a platform-independent environment for building applications, components and services.

The Framework layer: To allow reuse, this layer provides the environment for applications. It also provides all shared functionality such as common dialogs, file access, and configuration management.

The Application layer: "All *OpenOffice.org* applications are part of this layer. The way these applications interact is based on the lower layers."

The next sections describe these layers in more detail.

18.11.1 System Abstraction Layer

This section is reproduced from [11]. It references Figure 18.39.

Platform-depended implementations take place below the System Abstraction Layer (SAL) or are part of optional modules. "In an ideal world an implementation of the SAL-specific functionality and recompiling the upper layer module will allow you to run the applications. To provide the whole set of functionality, the optional platform specific modules, like telephony support or speech recognition, have to be ported, too." To reduce porting efforts, the SAL functionality is reduced to a minimal set, available on every platform. " . . . for some systems the layer includes some implementations to emulate some functionality or behavior. For example on systems where no native multithreading is supported, the layer can support so called 'user land' threads."

This description is not very clear. In fact, it is difficult to write meaningfully and clearly at a high level. What follows next, however, is indeed clear and meaningful.

As shown in Figure 18.40, the SAL consists of the Operating System Layer (OSL) the Runtime Library (RTL), the Standard Template Library (STL), and the platform-independent part of the Visual Class Library (VCL). These are described next.

18.11.1.1 Operating System Layer "The operating system layer (OSL) encapsulates all the operating system specific functionality for using and accessing system specific resources like files, memory, sockets, pipes, etc. The OSL is a very thin layer with an object-oriented API. In contrast to the upper layer this object-oriented API is a C-API." The reason for this is to allow easy porting to various platforms using different implementation languages. "For embedded systems or Internet appliances, for example, an assembler language can be used to realize the implementation."

18.11.1.2 Runtime Library "The runtime library provides all semi platform independent functionality. There is an implementation for string classes provided. Routines for conversion of strings to different character sets are implemented. The memory management functionality resides in this module."

The last sentence is useful and appropriate at this level. The meaning of "semi platform independent" is unclear. The second sentence is not clear but is presumably explained in the more detailed sections.

18.11.1.3 Standard Template Library "As a generic container library the standard template library is used. It supplies implementations for list, queues, stacks, maps, etc."

The relationship with the Standard Template Library that comes with C++ should be clarified.

18.11.1.4 Visual Class Library "The VCL encapsulates all access to the different underlying GUI systems. The implementation is separated into two major parts. One is platform-independent and includes an object-oriented 2D graphics API with metafiles, fonts, raster operations and the widget set use by the OpenOffice.org suite. This approach virtually guarantees that all widgets have the same behavior independently of the GUI system used. As a result, the look-and-feel and the functionality of the widgets on all platforms are the same."

This explains the squiggly lines in Figure 18.39 that separate the two parts of the VCL.

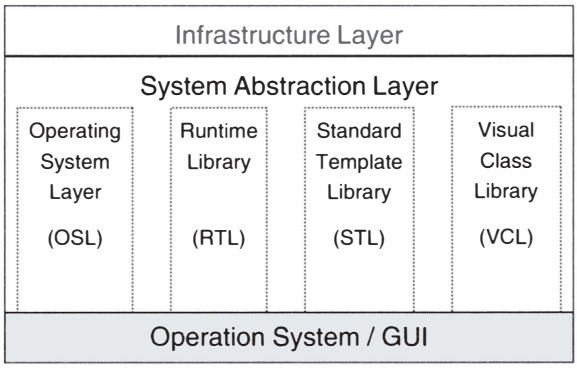


Figure 18.40 OpenOffice architecture—system abstraction layer

Source: Edited from OpenOffice, http://www.openoffice.org/white_papers/tech_overview/tech_overview.html#3.

The platform-dependent part implements a 2D-graphic drawing canvas that is used by the platform-independent parts. This canvas redirects functionality directly to the underlying GUI system. Currently, there exist implementations for the Win32, X-Windows, OS/2, and Mac. The access to the printing functionality, clipboard and drag-and-drop is also realized inside the VCL."

18.11.2 Infrastructure Layer

The Infrastructure layer consists of the parts shown in Figure 18.41. These are each explained next.

> The figure implies that *Compound Objects* (for example) depends on the *SAL* and that the *Infrastructure* layer depends on it. It implies no dependence between *Compound Objects*, *UCB*, and *SBL*.

18.11.2.1 Virtual Operating System Layer The purpose of this layer is "to make the usage of system resources like files, threads, sockets, etc. more convenient the virtual operating system layer encapsulates all the functionality of the operating system layer into C++ classes. The C++ classes here offer an easy to use access to all system resources in an object-oriented way."

18.11.2.2 Tools Libraries "The tool functionality of OpenOffice consists of various small tool libraries. This includes a common implementation for handling date and time related data, an implementation of structured storages, a generic registry, typesafe management, and persistence of property data."

18.11.2.3 Universal Network Objects *Universal Network Objects* is the component technology used within *OpenOffice.org*. "It . . . is heavily based on multi-threading and network communication capabilities."

> This paragraph says something important about the architecture of OpenOffice.

"The system consists of several pieces. An IDL-Compiler, which generates out of the specified definition of an interface a binary representation and the associated C-Header or Java technology files. The binary representation is platform and language independent and is at runtime used to marshal argument for remote function calls or to generate code on the fly for a specific language to access the implementation provided by the interface. This technique reduced the amount of generated code for the different language binding tremendously. The drawback is that not only for every language binding a specific backend for the code generation is needed, it is that for every specific compiler a bridging module is needed at runtime."

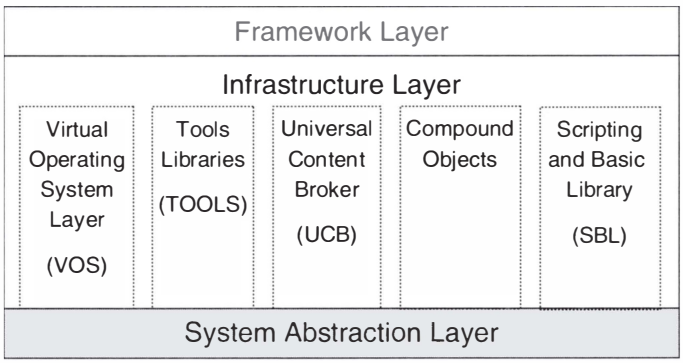


Figure 18.41 OpenOffice architecture—infrastructure layer

Source: Edited from OpenOffice, http://www.openoffice.org/white_papers/tech_overview/tech_overview.html#3.

This paragraph is not easy to understand but it is at a high level and becomes clearer as one reads more details. It mixes a description of the architecture with a rationale for it. Requirements documents frequently include pieces of rationale.

"Many parts of the UNO technology are implemented as UNO components. This facilitates flexibility and runtime extensions: e.g., providing new bridges or communication protocols. UNO provides transparent access to components locally or over the network. IIOP can be used for network communication. If components are realized as shared libraries, they can be loaded by UNO in to process memory of the application accessed by function calls. This does not require marshalling of arguments as required for remote function calls."

This paragraph seems to be heralding an ambitious design, consisting of objects available on the network.

18.11.2.4 Universal Content Broker

"The Universal Content Broker allows all upper layers to access different kinds of structure content transparently. The UCB consists of a core and several Universal Content Providers, which are used to integrate different access protocols. The current implementations provides content providers for the HTTP protocol, FTP protocol, WebDAV protocol, and access to the local file system.

The UCB does not only provide access to the content, it also provides the associated meta information to the content. Actually there is synchronous and asynchronous mode for operations supported."

18.11.2.5 OpenOffice.org Compound Objects

"The Compound Object implementation provide the functionality to build compound documents, where for example a spreadsheet is being embedded in a word-processor document." "The implementation provides a platform-independent implementation of this functionality for compound documents and for embedding visual controls such as multimedia players and various viewers. Storage is compatible with the OLE structure storage format. This allows access to OLE compound documents on every platform where OpenOffice.org is available. On Windows the implementation interacts with OLE services and so allows a tight integration of OLE-capable applications."

18.11.2.6 OpenOffice.org Scripting and Basic Library

"Scripting" refers to the ability to write procedures that cause the application to execute application functions in a desired sequence. For example, *.bat* files in Windows and *.sh* files in Unix are scripting files. Design documents inevitably rely on jargon that the reader must be familiar with.

"The scripting functionality that comes with the OpenOffice.org suite is a BASIC dialect featuring an interpreter that parses the source statements and generates meta instructions. These instructions can be executed directly by the supplied meta-instructions processor or can be made persistent in modules or libraries for later access. All functionality supplied by the upper level application components is accessible via a scripting interface in the component technology. This will help to ensure that new components using the OpenOffice.org component technology can be fully scriptable without spending a huge amount of effort.

The scripting interfaces are also implemented as components that will allow an easy integration of other scripting languages." They provide functionality, such as core reflection and introspection, similar to Java platform functionality.

18.11.3 Framework Layer

The Framework Layer has the parts shown in Figure 18.42.

These parts are described next.

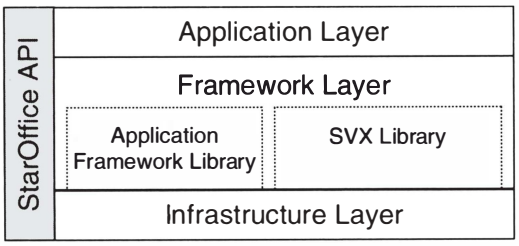


Figure 18.42 OpenOffice architecture—framework layer

Source: OpenOffice, http://www.openoffice.org/white_papers/tech_overview/tech_overview.html#3.

18.11.3.1 The Application Framework Library

See Figure 18.42.

"Functionality shared by all application and not provided by any other layer is realized here. For the framework, every visual application has to provide a shell and can provide several views. The library provides all basic functionality so only the application specific features have to be added."

"The Framework is also responsible for content detection and aggregation." The AFL provides template management and configuration management. It "is in some areas related to the compound documents, because of the functionality for merging or switching menus and toolbars. It also provides the capability for customization of applications."

18.11.3.2 SVX Library

"The SVX library provides shared functionality for all applications which is not related to a framework. So part of the library is a complete object-oriented drawing layer that is used by several applications for graphic editing and output; also a complete 3D-rendering systems is part of the drawing functionality. The common dialogs for font selection, color chooser, etc., are all part of this library. Also the whole database connectivity is realized here."

18.11.4 Application Layer

This layer consists of the actual applications such as the word processor, spreadsheet, presentation, charting, and so on. All these are realized as shared libraries, loaded by the application framework at runtime. The framework provides the environment for these applications and provides the functionality for interaction among them.

> Software architecture descriptions inform the reader, but in a general manner. They are necessarily imprecise about the meaning of the parts. Architectures of nontrival software applications have to be learned over time just as it takes time to learn any complex subject. This process is helped by delving into selected details as needed, perhaps even to the code level, and then re-reading needed architecture and detailed design descriptions. In this book, we omit the detailed design of OpenOffice.

18.12 SUMMARY

A software architecture describes the components of a software system and the way in which they interact with each other. There are many different ways a system can be architected. Garlan and Shaw have classified software architectures into categories such as dataflow, independent components, virtual machines, repository, and layered. Service-oriented architecture is another type of architecture in which various components are combined to provide a service.

Software designs are documented in a software design document (SDD). The IEEE publishes IEEE Std 1016-1998 for such a purpose. The SDD for the Encounter case study uses this as a document template.

For large software projects, it is important to modularize the software design. Modularization facilitates different groups of developers working on the different parts simultaneously. To make this work as efficiently as possible, the Facade design pattern can be used to provide a clean interface for each module. The Facade pattern is typically appropriate when developers are collocated. In distributed environments, Web services can often be used.

Once an architecture is selected, the project schedule is updated to reflect the order in which the parts are to be developed.

18.13 EXERCISES

1. In a paragraph, explain the purpose of a software architecture and how it relates to design.
2. Suppose that you are designing a batch simulation of customers in a bank. Being a batch simulation, the characteristics of the simulation are first set, then the simulation is executed without intervention. How could you describe this as a data flow application? Use a simple skeleton consisting of four parts to the diagram. (Identify your four parts, and then look at how you could describe this application as a state-transition diagram.) Which perspective offers more value in describing the architecture?
3. When designing a client-server architecture, there are generally two alternatives: *thin* and *thick* clients. A thin client implies that client functionality is kept to a minimum; most of the processing is performed via the server. A thick client implies that much of the functionality is contained in the client; the functionality on the server is kept to a minimum. Discuss the one or two major advantages and disadvantages to each of these approaches.
4. Operating systems are frequently designed using a layered architecture. Research the Linux operating system on the Internet, and explain how it utilizes a layered architecture. What are the benefits of such an architecture?
5. Consider a word processing application with which you are familiar. Sketch the software architecture of that program using one of the architectures described in this chapter. Describe the purpose of each of the components of your architecture.
6. Select an alternative software architecture for the word processing application of Exercise 5. Compare both architectures you selected and describe their relative merits.
7. Some design patterns are particularly relevant at the architectural level. Name two of these and explain their relevance.
8. Which software architecture is the best candidate for each of the following applications?
   a. An application for reordering auto parts from hundreds of stores
   b. A real-time application that shows the health of an automobile
   c. An application that provides advice to stock customers. It uses a multi-platform design consisting of several Web sites. One site continually collects and stores prices and other information about stocks; a second site continually collects stock advice from analysts; a third recommends portfolio suggestions to users.
   d. A scientific instrument company builds equipment for analyzing molecular structures. The application you are to design analyzes the structure of DNA molecules, which are very large.

TEAM EXERCISE

Architecture

Develop the architecture for your project. Describe your architecture using the IEEE standard, as in the case study accompanying this chapter. Make it clear what type of architecture and design patterns are being applied. Show at least one other architecture that you considered, and explain why you chose the alternative described. Include the use of metrics. It is not required that you automatically choose the architectures via metrics alone.

Track the time you spend doing this exercise in increments of five minutes, and include a time sheet showing the time spent by individuals and by the team. Use or improved upon the form in Table 18.2 that records the time spent per person on each module. Give your opinion on whether your tracking of time paid off, and whether your time could have been better managed.

Table 18.2 Form showing time spent per module

		Module			
		1	2	3	4
Team member	Smith	10	4		
	Jones		5	12	
	Brown	2			14

BIBLIOGRAPHY

1. Shaw, M. G., and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline,"* Prentice Hall, 1996.
2. Dijkstra, E., *A Discipline of Programming*, Prentice Hall, 1976.
3. Lea, D., *Concurrent Programming in Java: Design Principles and Patterns (Java Series)*, Addison-Wesley, 1996.
4. Kaluzniacky, E. K., and V. Kanabar. *Xbase Programming for the True Beginner: An Introduction to the Xbase Language in the Context of dBase Iii+, Iv, 5, Foxpro, and Clipper*, McGraw Hill Professional, 1996.
5. Jagannathan, V., Rajendra Dodhiawala, and Lawrence S. Baum, editors. *Blackboard Architectures and Applications*. Academic Press, 1989.
6. Engelmore, Robert, and Anthony Morgan (Editors), *Blackboard Systems (The Insight Series in Artificial Intelligence)*, *Addison-Wesley*, 1988.
7. Erl, Thomas, *"Service-Oriented Architecture: Concepts, Technology, and Design,"* Prentice Hall, 2006.
8. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1999.
9. Gamma, Erich, and Beck, Kent. *"Contributing to Eclipse: Principles, Patterns, and Plug-Ins,"* Addison-Wesley, 2003.
10. OpenOffice Project, http://www.openoffice.org/white_papers/tech_overview/tech_overview.html#3 [accessed November 29, 2009].