Software Engineering Design

Software Design Fundamentals

Once a software product is designed and specified, plans must be made to actually implement it. **Software engineering design** is the activity of specifying programs, sub-systems, and their constituent parts and workings to meet software product specifications. Software engineering design is the counterpart to software product design in that product design specifies the "external" features and behavior of a product while engineering design specifies the "internal" features and behavior.

When an architect designs a building, different drawings are prepared showing the walls, windows, doors, and so forth (often from different points of view), the heating and cooling system, the lighting and electrical system, and the plumbing system. This is because a single drawing showing all this detail would be much too hard to read. In like fashion, when software engineers design a software system, they often make models showing different *views* of the system. For example, some diagrams show the static decomposition of the software into sub-systems, packages, classes, and so forth. Others might show the state-based behavior of the system. Others might show the hardware nodes in the system and the software components that run on them. And there can be other views as well. Software engineers must be familiar with notations for making these views and the conventions and good practices in these design domains.

As discussed in the chapter on software quality assurance, defects can arise at any step in the software process, and it is important to try to prevent and eliminate them at each step of the process. Software design defects can be prevented in several ways, among them the following.

Design principles—A **design principle** is a statement that certain characteristics of a design make it better. Design principles guide choices among design alternatives by providing criteria to evaluate them. We discuss design principles below.

Design notations—Knowing design notations and how to use them helps engineering designers make good designs that are complete and consistent. The most important design notations are architectural box-and-line diagrams (showing components and their relationships), UML class diagrams (showing classes, attributes, operations, inheritance, and relationships), UML package diagrams (showing how the code for a system is grouped), UML state diagrams (showing system states and transitions between them), and UML sequence diagrams (showing messages passed among collaborating objects). We will learn about box-and-line diagrams and class diagrams below.

Design processes—There are many ways to make designs, but some approaches have proven over time to be more effective. We consider good design processes below.

Design patterns—A **pattern** is a model proposed for imitation. Architectural styles and design patterns are models of good solutions to common design problems. Familiarity with important design patterns results in better solutions, and makes design communication and documentation more concise. We discuss examples of architectural styles and design patterns below. In addition to preventing defects, patterns have other benefits as well: they make design practice easier and faster, and they promote code reuse.

The main technique for detecting and removing design defects is reviewing design specifications and correcting any defects. A particularly effective design review technique is active reviews. An **active review** is an examination of parts of a design by experts who answer questions about the design in their areas of expertise. Active reviews are effective because they employ experts who focus effort on careful and detailed study of a design. The active review process has the following phases.

Preparation—The designers choose the parts of their design that they are least happy with and that they think may contain the most important defects. Experts competent to evaluate these portions of the design are chosen, and their consent to perform the review obtained. The design team prepares questions for the reviewers about the design. These questions should force the reviewers to examine the design carefully and consider its merits critically.

Performance—The reviewers receive the design and the questions prepared for them and write answers to the questions. Although given deadlines, each reviewer is free to work on their task when and how they see fit. Reviewers are also free to work with the design team to obtain additional material or to get help understanding the design.

Completion—Once the reviewers deliver their results, the design team studies them to understand defects, problems, and suggestions from the reviewers. The designers then modify their design in response. The designers can interact with the reviewers to clarify the reviewers' comments or get elaboration about them.

Active reviews are expensive and time consuming, but they often yield valuable results. Remember that a design with major defects that are not detected until far into the implementation or testing phases will cost a lot more to fix than one whose defects are found during design.

Design Principles

As noted, a **design principle** is statement that certain characteristics make a design better. Thus, design principles can be used as quality criteria in generating and evaluating designs. Many of these principles are about modules. A **module** is some sort of program unit. Modules can be small and simple, like blocks in a control structure, or large and complex, like a sub-system of a big program. The things we usually refer to as modules are functions or sub-programs, operations, classes, packages, and source code files.

Simplicity

Simplicity is a fundamental principle in every design discipline. The principle is simply that

Simpler designs are better.

This probably seems reasonable, but the question arises, what makes a design simple? Clearly size has some effect on simplicity: designs with fewer elements in them tend to be simpler. But other factors affect simplicity as well. For example, designs with many connections between their parts tend to be more complicated. Designs with long computations involving several different kinds of objects tend to be complicated. And of course, designs with single threads of execution are usually simpler than designs with concurrent threads or processes. The ultimate test is whether a design is hard to understand. In comparing several design alternatives, if one is easier to understand than the others, then it is simpler, and by this principle better than the others.

Small Modules

The principle of small modules is that

Designs with small modules are better.

Small modules are easier to write, read, understand, and test. Many projects have guidelines about how large modules should be. For example, some guidelines recommend that classes contain no more than a dozen operations and be no more than 500 lines long, and that operations be no more than 50 lines long.

Information Hiding

Information hiding is shielding the internal details of a module's structure and processing from other modules. The principle of information hiding is that

Each module should shield the internal details of its processing from other modules.

The usual way to hide information is to declare program elements to have the smallest scope possible, and in object-oriented programming, to use visibility modifiers to restrict the accessibility of program elements as much as possible.

An instance of the first technique is the rule that variables should be declared in the innermost block possible. For example, if a variable is only used inside a while-loop body, then it should be declared in the body. An instance of the second rule is that class attributes should have private visibility in most situations, protected visibility if they are needed in descendent classes, package visibility on rare occasions, and almost never have public visibility.

Information hiding has many advantages, including the following.

- Modules that hide their internal structure and processing can be changed without affecting the remainder of a program.
- Modules that hide information are easier to understand, test, and reuse because they stand on their own.
- Modules that hide information are less likely to be affected by errors elsewhere in the program, and less susceptible to security breaches.

Information hiding is a fundamental software engineering design principle and should always be considered in design and implementation.

Module Coupling

Module coupling is the degree of connection between two modules. There are many ways that modules may be coupled, several of which are listed below.

- One class is an ancestor of another class
- One class has an attribute whose type is another class
- One class has an operation with a parameter whose type is another class
- One operation invokes another operation

The more of these sorts of connections there are between modules, and the more often they occur, the more strongly coupled two modules are.

When two modules are strongly coupled, it is difficult to understand, test, and reuse one of them without the other. Modifying one may cause problems in the other, so it is harder to change strongly coupled modules. These considerations lead to the following principle.

Module coupling should be minimized.

This principle suggests that in the best case, modules would not be coupled at all. But of course, modules must be coupled to some extent: there would be no point for modules to be in the same program if none were connected to the others in any way. Some module coupling is necessary, but we should try to couple modules as little as possible.

There are several ways to minimize module coupling. Hiding information usually helps, as does making modules cohesive (cohesion is discussed next). Another technique is **delegation**, a tactic in which one module trusts another with some responsibility. For example, suppose module A must read data from some input resource controlled by module B, and then place this data into a collection. Module A could use a loop to read data items one by one from module B until the input was exhausted, or module A could just delegate the job to module B and ask it to provide a collection filled with data from its input resource. The latter approach lessens the coupling between A and B. The use of interfaces can also dramatically decrease coupling.

Module Cohesion

Module cohesion is the degree to which the elements in a module are related to one another. Modules are cohesive when the data that they hold belong together, and the operations in the module process mainly the data held by the module. For example, suppose a class represents maintenance requests at a hospital. This class will hold data about the type of the request, the requestor, the date and time of the request, its severity, and so forth. What sort of data about the requestor should be stored? There needs to be enough data to identify the requestor (such as a name or identifier), but should the maintenance request class store the phone numbers, address, gender, title, and other information about the requestor? Clearly, this data does not belong in such a module, and including it lowers the module's cohesion. Similar considerations apply to other sorts of modules, such as functions and operations, packages, and so forth.

The principle of cohesion is stated as follows.

Module cohesion should be maximized.

Highly cohesive modules are easier to write, read, understand, test, and reuse. When modules are loosely coupled, they are usually also cohesive. Simple modules and modules that hide information well also tend to be cohesive. The main way to insure good module cohesion is to examine each of its elements and consider whether it really belongs with the rest. In general, modules that are difficult to name or that have vacuous names (like "utilities" or "miscellaneous") are not very cohesive.

Design Processes

Design is best thought of as a kind of problem solving. Like any sort of problem solving, the design process begins with **analysis**, which is breaking a problem into parts to understand it. Design analysis often employs various models. Making models for a problem often reveals details, clarifies relationships between parts of the problem, and suggests solutions.

Once a problem is understood, then various solution candidates can be proposed and investigated. Solution ideas can come from many places, including solutions to similar problems, standard solutions to classes of problems (such as design patterns and architectural styles, discussed below), and the results of various brainstorming and idea generation activities. Early solution candidates are rarely perfect; they must be evaluated and the bad ones thrown out while the best are kept. Often a promising but inadequate solution can be improved by combining it with other solutions or modifying in some other ways. In this way solutions are gradually refined and improved until a good design is produced. This is a trial and error process that repeatedly refines and evaluates candidate solutions until the design problem is solved.

The design process ends when the final solution is documented and checked. The diagram in Figure 1 illustrates this general process

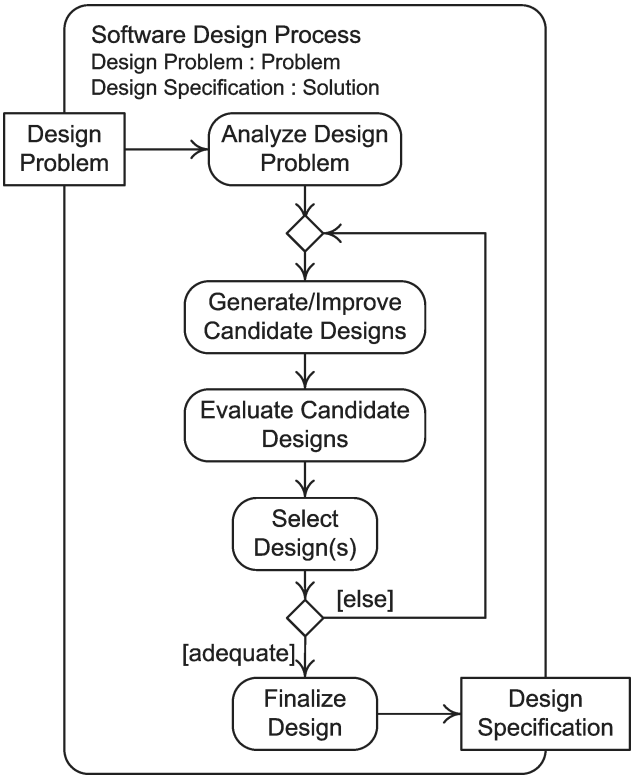

Figure 1: The General Software Design Process

Different phases or activities in software design are often distinguished at different levels of abstraction. **Architectural design** is the activity of specifying a program's major parts, including their responsibilities, properties, and interfaces, and the relationships and interactions among them. For example, a network-based product might have a server component running on a server computer, and a client component running on users' computers. The two major components might themselves be divided into several components. Specifying these components, how they communicate with one another, how quickly each has to respond to the other, whether and how they preserve state between interactions, and so forth, would be an architectural design activity.

Detailed design is the activity of specifying the internal elements of all major program constituents, including their structures, relationships, processing, and often their algorithms and data structures. For examples, the detailed design of the client-server system alluded to above would include specifying the classes in each of the components of the product, along with the attributes and operations in the classes, the responsibilities of each class, the way classes collaborate to accomplish tasks, the processes and threads in the components and what runs in them, and so forth.

Top-down design begins with architectural design and moves on to detailed design. Parts of an architecture might be designed down to a low-level of detail before other parts of the architectural design are completed, or an architecture might be designed to include some existing low-level modules. This is how a design process can be partly bottom-up while being mainly top-down. We turn next to a closer look at architectural and detailed design.

Architectural Design

As noted above, **architectural design** is the activity of specifying a program's major parts, including their responsibilities, properties, and interfaces, and the relationships and interactions among them.

Several notations are used for this purpose, but perhaps the most widely used are simple **box-and-line diagrams** that show architectural components with arbitrary icons called *boxes*, and relationships and interactions between them with various connectors called *lines*. Because there is no standard for box-and-line drawings, it is important to make clear what the various boxes and lines in a diagram mean. This can be done by using common symbols (for example, using a drum icon for a data store), or by supplying a legend with the diagram. Examples of box-and-line diagrams appear later in this section.

A **pattern** is a model proposed for imitation. For example, a dress pattern is a paper template for the pieces of cloth cut out to be sewn together to make a garment. The paper templates are a model because they represent the shapes of cloth and how they must be sewn together. They are proposed for imitation by those who would like to make the garment.

Software design patterns are models proposed for imitation in solving a software design problem. Software design patterns occur at several levels of abstraction. At the architectural level, software design patterns are called **architectural styles**, which we can define as models of system components and their relationships. There are many common architectural styles; here we consider two of them.

Layered Style

The **layered architectural style** is a model in which components are arrayed linearly in ranks or levels with some "above" and some "below." These ranks or levels form the layers in the architecture. Each layer provides services to the layers above, and uses services of the layers below. The figure below shows a layered architecture in a box-and-line diagram. In this diagram (sometimes called a *wedding cake diagram*), rectangles represent layers and vertical adjacency represents the *uses* relationship between layers (that is, if layer *x* is above layer *y*, then *x* uses *y*).

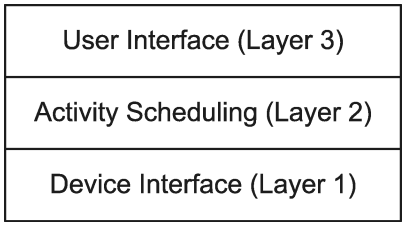

Figure 2: A Layered Architecture

In this architecture for a program that controls some hardware, such as an HVAC unit, a Device Interface layer has software that directly interacts with some hardware device(s). The Device Interface layer provides a simple, device-independent interface that hides details about interaction protocols, command formats, and so forth. This simplified interface is used by the Activity Scheduling layer. This layer is responsible for determining what the hardware should be doing and when it should be doing it. For example, a smart HVAC controller might adjust the HVAC thermostat depending on the day of the week and the time of the day—the Activity Scheduling layer keeps track of the time and adjusts the thermostat (through the Device Interface layer) accordingly. The top layer is responsible for interacting with human users. It displays information and collects user inputs to direct the Activity Scheduling layer. For example, the User Interface layer tells the Activity Scheduling layer the times and days to adjust the thermostat based on input from users.

The layered architectural style solves several problems:

- The layers in a layered architecture provide interfaces at different levels of abstraction, the highest at the top, and the lowest at the bottom. The layers thus provide a means of organizing a solution by levels of abstraction.
- It decomposes the program into parts, simplifying design and implementation.
- It (should) divide the program into independent modules (the layers) with well-defined interfaces, thus increasing cohesion and helping hide information.
- It constraints the communication paths among modules, thus decreasing coupling.
- It makes it easy to replace or modify parts of the program (the layers), thus increasing maintainability and encouraging reuse.

There are some minor drawbacks to using a layered style. Often collaborations must begin at the top layer, ripple down to the bottom, and then ripple back up, which may not be very efficient. Also, it may be hard to get the layers right: if too big, a layer can become too complicated. If too small, a layer may not be cohesive. Nevertheless, layered style architectures offer so many advantages that they are very common.

Pipe and Filter Style

In the **pipe and filter architectural style** the principal participants are modules that convert inputs into outputs (called **filters**) and the channels that connect the modules (called **pipes**). Box-and-line diagrams of pipe and filter architectures typically represent the filters as boxes and the pipes as lines or arrows. What distinguishes the pipe and filter style from the layered style is that there are no limits placed on communications—the output of any filter can be piped to any other filter, becoming its input. To make this possible, all filters must accept the same type of input and all filters must produce output of that type as well.[1]

For example, consider the following pipe and filter architecture for a billing system in which the inputs to (and outputs from) all of the filters are lines of ASCII characters (that is, each line is a sequence of ASCII characters terminated by a newline character).

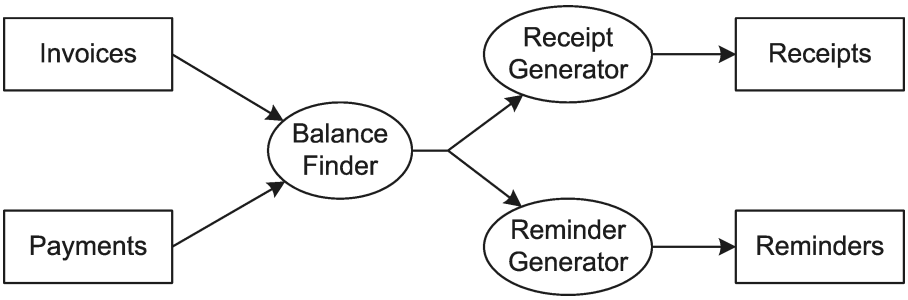

Figure 3: A Pipe and Filter Architecture

The Balance Finder is a filter that accepts a collection of account invoices and account payments, matches them (by account number), and determines which accounts have a remaining balance and which do not. The Receipt Generator is a filter that creates receipts for accounts that do not have a balance and the Reminder Generator is a filter that creates reminders for accounts that do have an

[1] In mathematics, a set is said to be *closed* under an operation if the result of that operation when performed on a member of the set is itself a member of that set. In a pipe and filter architecture, the set of inputs is closed under all filters.

outstanding balance. Invoices and Payments are files acting as data sources, and Receipts and Reminders are files acting as data sinks.

Pipe and filter architectures have been used successfully in a wide variety of areas. Operating systems based on Unix make extensive use of filters (the inputs to which are *streams*, which are sequences of bytes). For example, one can read a file, find all of the lines that contain the string "Overdue," and sort those lines by piping to the `cat`, `grep` and `sort` filters as follows.

```
cat in.txt | grep "Overdue" | sort > out.txt
```

The pipe and filter architectural style has two major advantages.

- It is simple and easy to understand.
- It promotes code reuse (i.e., it is easy to combine filters to provide new services).

The primary drawbacks of the pipe and filter architectural style are the following.

- It forces the lowest common denominator of data transmission.
- All data flows are unidirectional.
- There may be capacity constraints on the pipes.

There are many other architectural styles (see [1]); we have presented these two styles as widely-used examples that illustrate both a static model (a layered style architecture shows system components and relationships rather than behavior during execution), and a dynamic model (a pipe-and-filter architecture shows how data flows through components during execution).

Architectural styles are also often combined to form **hybrid architecture** displaying several styles at once. For example, the Figure 4 below shows two views of a compiler.

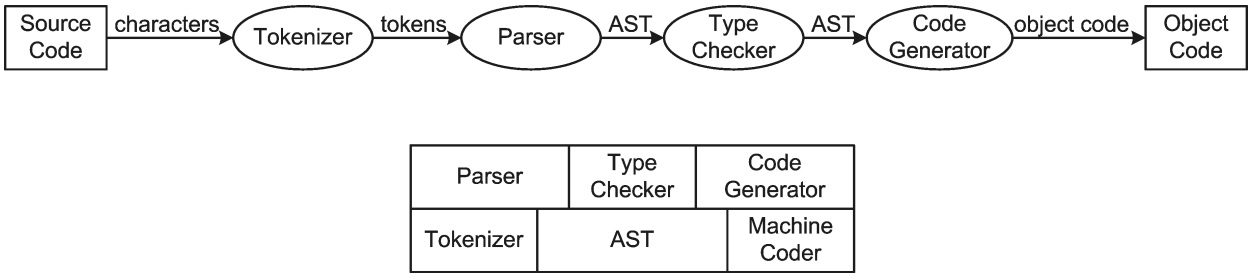

Figure 4: Two Views of a Compiler

The top portion of Figure 4 shows a pipe-and-filter view of a compiler. characters from a Source Code file stream into a Tokenizer that recognizes things like numbers, keywords, operators, and so on (tokens), and passes a stream of tokens to a Parser. The Parser recognizes statements, declarations, control structures, and so forth in the program and builds in internal representation called an *abstract syntax tree* (AST). The Parser passes this AST to a Type Checker that makes sure the program follows all the type rules of the language, adds automatic type conversions to the AST, and so forth. The AST is then passed along to the Code Generator that creates object code. The object code is written to an Object Code file.

An alternative layered view of the same program appears in the bottom portion of Figure 4. In this view, there are two layers in the compiler. The top layer consists of the Parser, Type Checker, and Code Generator. This layer makes use of the modules in the bottom layer, namely the Tokenizer, AST, and Machine Coder (which accepts commands to generate machine-level code and produces machine code for a particular computer). The bottom layer modules are accessible to modules that

physically touch it above. For example, the Parser access the Tokenizer and AST but not the Machine Coder.

Both of these views are useful to understand the design of the compiler and provide guidance in detailed design and coding. The compiler implements both a pipe-and-filter style and a layered style, so it has a hybrid architectural style.

Software Architecture and Non-Functional Requirements

You will recall that a non-functional requirement is a property that a software product must have. In general, non-functional requirements are constraints on the manner in which a product can behave or the way it is implemented. Typical non-functional requirements specify constraints on security, reliability, performance, efficiency, scalability, portability, or changeability. It turns out that most non-functional requirements can only be satisfied if a product has the right architecture, so architectural design focuses a lot on satisfying non-functional requirements.

To illustrate the connection between non-functional requirements and architectural design, we consider some design guidelines and practices for helping achieve two different non-functional requirements. More architectural styles can be found in several books, such as Taylor, Medvidović and Dashofy [1].

Achieving Performance Goals

The **performance** of a product is its ability to achieve computational goals without exceeding limits on time, memory, or energy. For example, a product might be required to compute an answer to a certain kind of problem within half a second, or to execute in a memory partition not exceeding one megabyte. There are several ways that architectural designers can meet performance requirements, including the following.

Use Budgets—Each component and connector in an architecture can be assigned a limit on the time, space, or energy it can use to do certain tasks. These limits are based on the total allotted to each function such that if all the components and connectors used to realize the function stay within their limits, then the entire product will meet its performance requirements.

Choose Appropriate Styles—Some styles are better for achieving performance goals than others. For example, if fast performance is a, then a layered style with many layers may not be a good choice.

Modify Styles—There are often modifications or elaborations of styles that can help meet constraints. In a layered style, special operations might be added to a low-level layer and made accessible to a high-level layer so that intermediate layers can be circumvented to speed up processing. Additionally, a cache might be added in a high-level layer to reduce interactions with lower levels to speed up processing.

Emphasize Simplicity—A highly configurable component that provides many operations to its clients, though easy to use (and reuse), may take up too much memory or be too slow to satisfy performance constraints. A simple version of the component with a stripped-down interface, though harder to use, will often be smaller and faster.

Minimize Synchronous Interactions—A **synchronous interaction** is one in which a component, after invoking an operation in another component or sending a message to another component, must stop its processing and wait for a reply. **Asynchronous interactions**, on the other hand, are those in which a component continues processing after invoking an operation in another component or sending a message to another component. If a component must complete its

work in a limited time, it is better for it to be able to continue its work while another component provides a service than to have to wait for the other component to finish.

Achieving Reliability Goals

The **reliability** of a product is the probability that it will behave as specified under normal operating conditions for a given period of time. For example, a product might have a non-functional reliability requirement that its probability of performing without failure under normal use for 24 hours is 99.9%. Among the ways that architects can make systems sufficiently reliable are the following.

Choose Appropriate Styles—Some styles are inherently more reliable than others. For example, a client-server style tends to be quite reliable because failure of all but one of the components does not cause failure of the entire system: any client can fail without consequences for the overall system. Unfortunately, failure of the server is catastrophic; however, steps can be taken to make the server sufficiently reliable.

Modify Styles—One step that can be taken in a client server style system is to modify the style to use redundant servers. Then failure of a redundant server component does not cause overall system failure.

Control Component Interactions—Interactions between architectural components should occur only through explicit interfaces. Otherwise, unexpected interactions (like side-effects or unexpected thread interactions) could cause system failure.

Handle Exceptions—Good exception handling interactions must be defined at the architectural level so that the system as a whole can (at least try to) recover from failures in its components. This guideline is connected to the last one: even if all exceptions are handled well, unexpected interactions that go around the exception handling mechanism can cause system failure.

Monitor System Health—If components run in different threads or processes, and some component becomes unresponsive, the system as a whole need not fail if a new version of the component can be started in a new thread or process. A health monitor component can constantly check on other components and restart those that fail.

Note that some of the practices and guidelines for increasing reliability conflict with those for achieving performance guidelines. For example, architectural styles that promote reliability often are not very efficient, making it harder to meet performance goals, and designs that make systems faster and smaller may be more prone to failure. This is part of what makes architectural design so much fun: the architect gets to make trade-offs between conflicting non-functional requirements.

Detailed Design

Recall that **detailed design** is the activity of specifying the internal elements of all major program constituents, including their structures, relationships, processing, and often their algorithms and data structures. Detailed design spans many levels of abstraction, from specifying several classes and their interactions to choosing data structures and algorithms. Here we focus mainly on the more abstract parts of detailed design having to do mainly with classes and their interactions.

UML Class Diagrams

Several UML notations are widely used for this sort of design.

Class diagrams represent classes, their attributes and operations, interfaces, inheritance relationships among classes, implementation relationships between classes and interfaces, and associations among class instances.

Sequences diagrams represent collaborations between class instances by depicting message passing between instances over time. They also show instance creation and destruction and the flow of control that provides the context for message passing.

Interaction diagrams represent collaborations between class instances by depicting referential relationships between instances and the message traffic supported by these referential links.

State diagrams depict the various states, or modes of being, that a class instance (or a system, subsystem, or other entity) may occupy, the transitions between states and the conditions and events governing transitions, and the actions that occur during transitions and while an entity is in a given state.

In this chapter we present class diagrams. The main symbol in a class diagram is a rectangular **class symbol** consisting of one or more **compartments**, shown as vertical divisions in the class symbol. The top compartment is required and contains the class name. The second compartment is optional and contains class attributes. A class attribute is specified by listing its name, possibly followed by a colon and its type. The third compartment is optional and contains class operations. An operation is specified by listing its name, possibly followed by a comma-separated list of argument names and types in parentheses, possibly followed by a colon and a comma-separated list of the types of values returned by the operation. Both attributes and operations can optionally be preceded by a **visibility modifier**: + for public, # for protected, ~ for package, and – for private. Static attributes and operations are underlined. Figure 5 shows examples of class symbols.

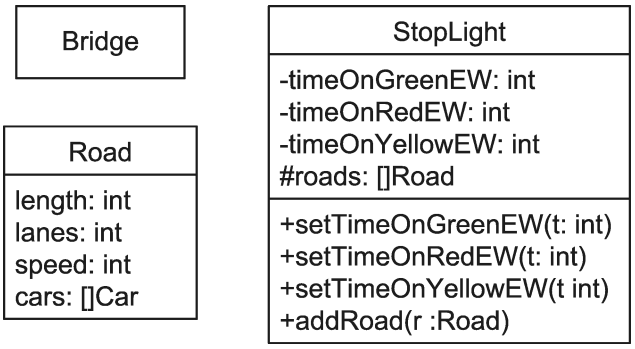

Figure 5: Examples of Class Symbols

In Figure 5, the Bridge class is shown with no attributes or operations. This does not mean that it has none, just that none are shown. The Road class is shown with (some of) its attributes, and the StopLight class with both attributes and operations. The first three StopLight attributes are marked as private and the fourth as protected. All StopLight operations are marked as public.

Interfaces in UML define sets of operations and are used to indicate that the set of operations is provided or must be provided. They are represented by class icons with the stereotype «interface» above the class name. A class that realizes or implements an interface (i.e., a class that provides all of the operations in the interface) is connected to the interface it realizes by a dashed line terminating at a solid triangle attached to the interface symbol. This connector is a variant of the *generalization* connector (UML's symbol for inheritance), which is a solid line from the child class terminating at a solid triangle attached to the parent class. Figure 6 illustrates interface implementation and inheritance relationships shown in UML class diagrams.

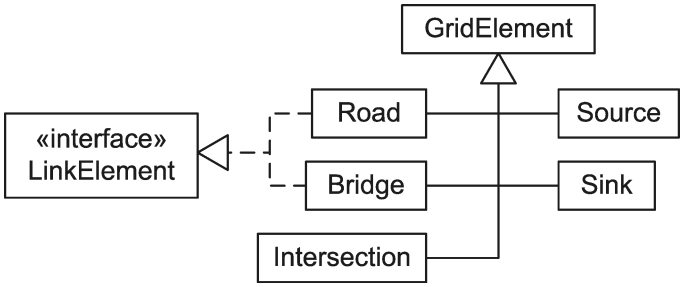

Figure 6: Interface Realization and Class Inheritance

This diagram shows that Road, Bridge, Intersection, Source, and Sink are all sub-classes of GridElement (i.e., they are all specializations of GridElement or, alternatively, GridElement is a generalization of each of them), and that Road and Bridge also implement the LinkElement interface.

Class diagrams may also show relationships between class instances, called **associations** in UML. These are indicated by a line between class symbols, usually labeled with the association name. Furthermore, the number of participating instances in each role of an association, called the **multiplicity**, can be shown on the diagram. Multiplicities are specified in comma-separated lists of ranges of the form $m..n$ where m and n are non-negative integers, $m \leq n$, and n may be an asterisk, meaning "arbitrarily many." If $m = n$ then the range may be abbreviated to just a single number. For example, 0..1 means zero or one, 1,3,5,7..10 means one, three, five, or between seven and 10, and * means zero or more. The association "is a biological parent of" between Parent and Child classes has a multiplicity of 2 on the Parent end (because everyone has exactly two biological parents), and 1..* on the Child end (because every Parent has one or more children). Figure 7 is a class diagram illustrating multiplicities.

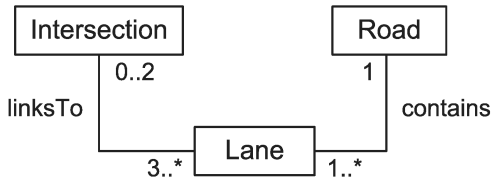

Figure 7: UML Association Multiplicities

Figure 7 depicts a few classes in a traffic simulation system. In this program, an Intersection is linked to three or more Lanes, and each Lane is linked to 0 to 2 Intersections (a Lane has 2 ends that both might be linked to Intersections, or one or both may be linked to other entities). A Road contains at least one Lane and perhaps more, but every Lane is contained in only a single Road.

In some situations, it is important to indicate that two (or more) classes exhibit a part-whole (or containment) relationship. In UML, this is accomplished using an **aggregation** relationship that is denoted by a line with a diamond head (rather than a triangle head). Figure 8 is a class diagram illustrating aggregation.

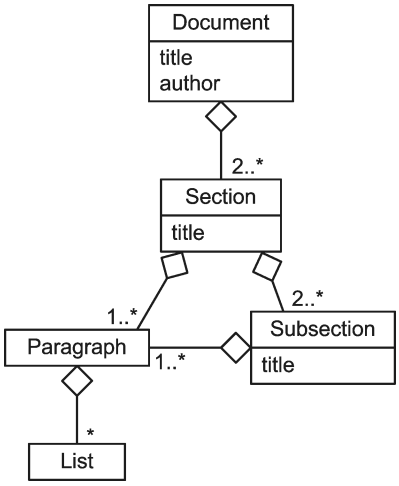

Figure 8: UML Aggregation

Figure 8 depicts a model of a Document that consists of two or more Sections, each of which can contain one or more Paragraphs or two or more Subsections (which themselves can contain one or more Paragraphs). Note that the use of aggregation relationships in this example provides additional information over and above the association relationships in the previous example. Specifically, the aggregations indicate that, for example, the Document isn't just associated with Sections but it contains Sections.

Design Patterns

Architectural styles are design patterns at the highest level of abstraction in engineering design. Patterns at the level of classes and modules are simply termed **design patterns**. We will now use UML class diagrams to consider three examples of design patterns, the Composite, Command and Decorator patterns. More patterns can be found in Gamma, Helm, Johnson, and Vlissides [2].

The Composite Pattern—Many things that we model in software have a hierarchical structure. For example, documents can have parts, which themselves can have parts, and so on; windows can have parts that themselves can have parts, and so on; and processes can have parts that themselves can have parts, and so on. When building hierarchies, it must be possible for a hierarchy to have other hierarchies as parts. Furthermore, though often everything in a hierarchy must be treated the same way, at other times the things at the bottom of a hierarchy need to be treated differently. The **Composite pattern** provides a way of conveniently handling hierarchies using classes. The static structure of the Composite pattern is shown in Figure 9 below.

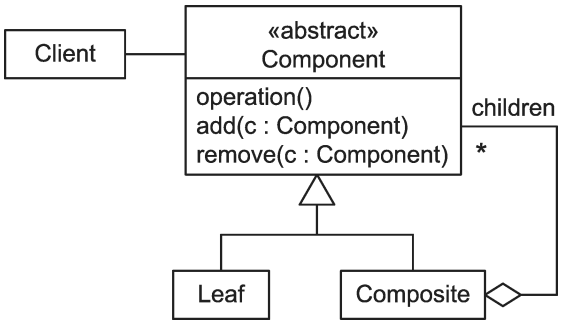

Figure 9: The Composite Pattern

The Component abstract class defines default operations for all components in a hierarchy. A Leaf object has no children while a Composite object is a container for children. Note that both a Composite and a Leaf can be a child of another Composite, so arbitrary hierarchies can be formed. Also, note that a Client object need not know whether a component that it has a reference for is a Leaf or a Composite because they both implement the same interface.

When a Client calls an operation() to manipulate the elements of the hierarchy, a Composite object usually just forwards the call to all its children, while Leaf objects at the bottom of the hierarchy respond directly to the method.

A good example of the use of the Composite pattern is graphical components in a GUI. All implement a Component interface with operations common to all graphical components. Some components, such as buttons and text fields, do not have any sub-components—these are leaf components. Others, such as windows, frames, or panels, are containers for other components—these are composite components. Windows, panels, and frames can be placed within other windows, panels and frames to construct complex GUI layouts. Of course, ground-level components like buttons and text fields are also placed in windows, panels, and frames.

The Composite pattern has several advantages.

- Arbitrary hierarchies can be constructed easily.
- Hierarchy details need not concern clients, which only have to deal with a Component interface.
- It is easy to add new Leaf and Composite classes into the structure without disrupting existing code.

Among the difficulties of using the Composite pattern is that Leaf components have methods that don't make sense for them, such as adding child components. In a variation of the pattern, such methods can be moved from the Component to the Composite class (where, strictly speaking, they belong), but doing so removes the uniform interface that all Components share. Some designers make one decision when using this pattern, and some make the other.

The Composite pattern occurs very frequently in a variety of applications, and it is a good pattern for designers to have in their toolkits.

The Command Pattern—In graphical user interfaces (GUIs), code must be executed when users click buttons, move scrollbars, select items from menus, and so forth. The code presenting GUI components and implementing their behavior is part of libraries or frameworks used by all programs with GUIs. How is the code implementing particular behavior associated with GUI components? The usual technique is to associate functions, called *listeners*, *handlers*, or *callbacks*, with GUI components. When some action, such as a button press, scrollbar motion, or menu selection occurs, the listener function associated with the button, scrollbar, or menu item is invoked. This is a general solution that nicely separates and decouples the general from the implementation-specific parts of user interface code, is easy to work with, and is efficient.

Encapsulating functionality and passing it to components or storing it for later use is a common idiom in many kinds of software. In many object-oriented languages, functions cannot be manipulated as pure values, so they must be implemented as operations in objects and then the objects are passed or stored. This is the basis of the **Command** pattern, whose static structures is pictured in Figure 10.

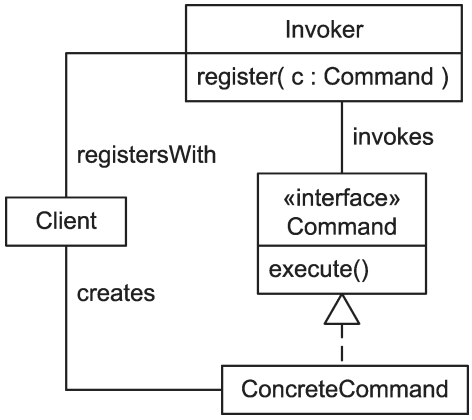

Figure 10: The Command Pattern

A ConcreteCommand class is written to encapsulate some desired functionality in its execute() operation. A Client class instance creates an instance of ConcreteCommand and passes it as the parameter of the register() operation of the Invoker class. When the time comes, the Invoker class instance invokes the execute() operation of the Command instance, putting into play the encapsulated functionality registered with it.

The Command pattern has the following advantages.

- The Client and the Invoker are loosely coupled. The Invoker knows nothing of the Client or the ConcreteCommand that it invokes, except that the latter implements the Command interface. Similarly, the Client does not depend on any aspects of the Invoker other than its guarantee to invoke the Command interface execute() operation under certain circumstances.
- The Client is decomposed and simplified because part of its functionality is encapsulated in the ConcreteCommand class.
- Command instances can be passed and stored as regular values, even if functions cannot be.
- ConcreteCommand classes can store all sorts of data and can support various operations, making them highly configurable for the Client.
- The Command interface can contain many operations to be called under a variety of circumstances, so this pattern can be generalized to work in more complex situations.
- The Invoker can support registration of several different Commands, several instances of the same Command, and Command deregistration.

The Command pattern is found throughout GUI implementations in object-oriented languages that do not allow functions as values (like Java, for example). It is also found in other event-driven contexts.

The Decorator Pattern—Sub-classes of a class add attributes and operations that will attach to all instances of the sub-class. Classes must be chosen when programs are written, so the extra attributes and operations of a sub-class cannot be added or removed at runtime (in many languages). If many different collections of attributes and operations are possible, then a complicated inheritance hierarchy must be created to accommodate all the possibilities. The **Decorator** pattern offers a way to add (and remove) attributes and operations to arbitrary class instances at runtime, and to add various collections of attributes and operations without creating a complex class hierarchy.

The structure of the Decorator pattern appears in Figure 11.

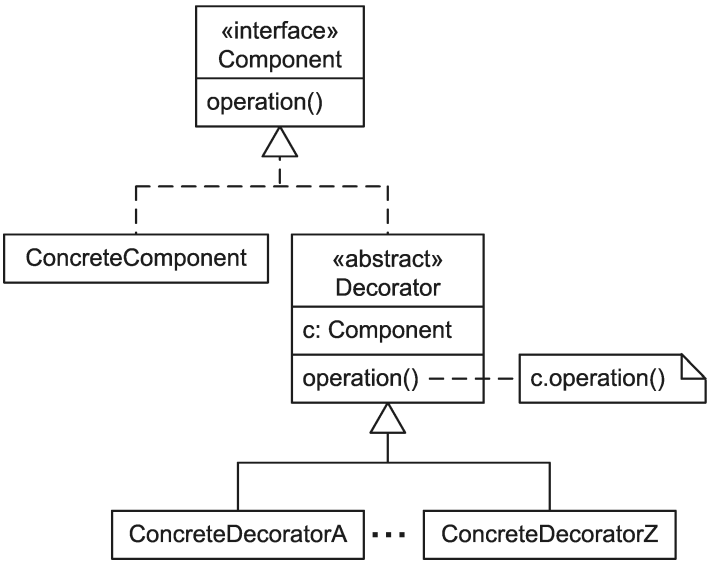

Figure 11: The Decorator Pattern

In Figure 11, the class whose instances are to be decorated is ConcreteComponent. It implements various operations signified by the operation() method in the Component interface. The Decorators all share the Component interface and record the decorated component in the attribute c. By default they simply call the operations of the component c that they decorate. However, they may also add their own attributes and operations, and override Component operations if they wish, adding processing before, after, or in place of the Component's operation processing. Furthermore, a Decorator may decorate another Decorator, so an entire chain of Decorators can be used to add various features in arbitrary ways.

To illustrate this pattern, suppose we are writing a program to support order and production of cakes at a bakery. Consider the design in Figure 12.

The BasicCake class implements the price() and invoice() methods without calling any other methods. Suppose a customer wants a cake with a happy birthday message and rosettes. We could create a BasicCake instance and a chain of Decorators to represent this order with code like the following.

```
Cake cake = new Message("Happy Birthday", new Rosettes(new BasicCake()))
```

As noted, the Decorator pattern has several advantages, including the ability to (apparently) add or remove class features at runtime and to simplify class hierarchies. Its disadvantages are that it adds lots of little classes and it may be difficult or confusing to use when there are a lot of basic components and decorators.

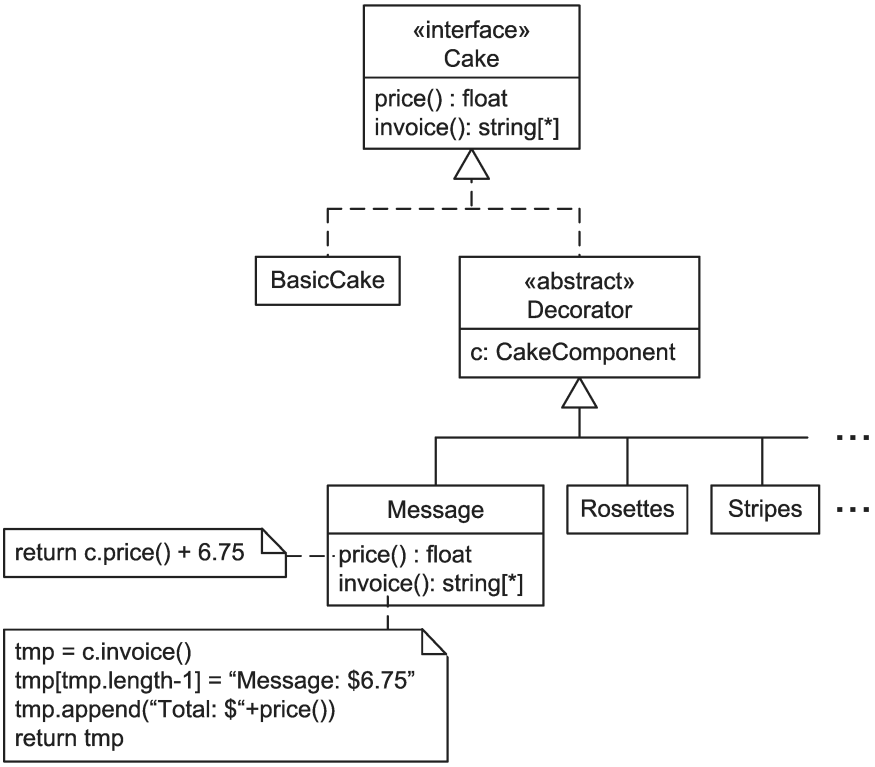

Figure 12: Cake and Decoration Classes

The Decorator pattern is widely used. For example, the Java IO library uses the Decorator pattern to add buffering and filtering, and to make data sources and sinks into IO streams.

References

1. Richard Taylor, Nenad Medvidović and Eric Dashofy. *Software Architecture: Foundations, Theory, Practice*. John Wiley and Sons, 2010.
2. Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.