Module 4: System Modeling; Architectural Design

Objectives:

1. Understand the importance of architectural design.
2. Be familiar with some of the common techniques used in designing the architecture of software.
3. Be introduced to several common application architectures, and examples of software applications or systems that use those architectures.
4. Understand some of the basics of four major types of application architectures: data processing, transaction processing, event processing, and language processing, and be aware of some of the problems encountered with those architectures.

Assigned Reading:

1. *Sommerville*, chapters 5 & 6.
2. Optional: Web sections for chapters 5 & 6 (available at http://iansommerville.com/software-engineering-book/web/web-sections/):
    a. UML (Chapter 5)
    b. Data-flow Diagrams (Chapter 5)
    c. Executable UML (Chapter 5)
    d. Architecture Patterns (Chapter 6)
    e. Application Architecture (http://iansommerville.com/software-engineering-book/web/apparch/)(Chapter 6)
    f. Reference Architecture (Chapter 6)

Assignment:

No assignment for this module.

Quiz:

Take the quiz for Module 4.

Lecture:

System Models

In conjunction with requirements specification, as well as later during design, one or more models are typically produced for the software system being developed. There are many different types of models, and not every model needs to be used for every system, but they all serve to provide a graphical representation of one or more aspects of the software. A good analogy related to the physical world

would be a scale model of new industrial park, showing rough, but pretty accurate models of the buildings, parking lots, landscaping, etc. With software, system models focus on aspects such as:

1. context—what are the system boundaries?
2. interactions with users, hardware, and other software
3. structural details
4. behavior

Context Models

A context model describes the boundaries of the software. Boundaries may be physical, such as a description and locations of the hardware with which the software will interact; they may describe limitations on how the software will interact with other software systems; or they may determine the extent of the requested functionality that will be provided. In short, a context model shows how the software will fit in with its environment. Part of the scope statement assignment for the group project asked you to describe the boundaries of the project you intend to build. Most likely the extent of your context model involved decisions such as:

- Will the software be a standalone application or will it work across an internet connection?
- What platforms do you intend to support?
- Will the software need to interact with any other software (excluding operating systems)?

Interaction Models

Interaction models describe interactions between the software and the end users, interactions between individual components of the software itself, interactions with other software systems, and interactions with hardware or other physical devices. Two commonly employed techniques for modeling software interactions are the **use case diagram** and the **sequence diagram**, both of which are included as part of the Unified Modeling Language (UML). A use case diagram represents the interaction between the software and usually a single other entity, which can be an end user, another software application, or hardware. Use case diagrams are often used to help elicit requirements. Although use cases won't replace traditional requirement elicitation techniques, they can be of immense help in determining which requirements should be high priority. By allowing a developer to "walk through" a typical interaction, a use case can also help to uncover some non-functional requirements that might otherwise have been difficult to elucidate. A sequence diagram is designed to illustrate a precise sequence of events and actions that should happen when a particular task is carried out, as well as which components should be involved in the task. Sequence diagrams can be high-level, abstract representations, when used during requirements specification, or finely detailed, when used later on during design.

Structural Models

As their name implies, structural models are concerned with showing what the architecture of the software will look like. A structural model will typically include one or more diagrams that depict the components that will be developed. The UML class diagram is one example. The class diagram is geared towards object-oriented development, showing the various object classes that will comprise the overall software, but the basic principle can be used for non-object-oriented development, to show the modularity of the design.

Data Flow Models

Data flow diagrams (DFD's) are useful when you need to track how data are gathered, processed, and stored in a system. Normally, when one creates a data flow diagram, one begins with a very simple diagram that shows the system or subsystem as a whole, and then gradually shows more details in subsequent diagrams. The simplest data flow diagram is a Level 0, or context-level diagram. Figure 4.1 shows what the Level 0 data flow diagram for an order processing component might look like.

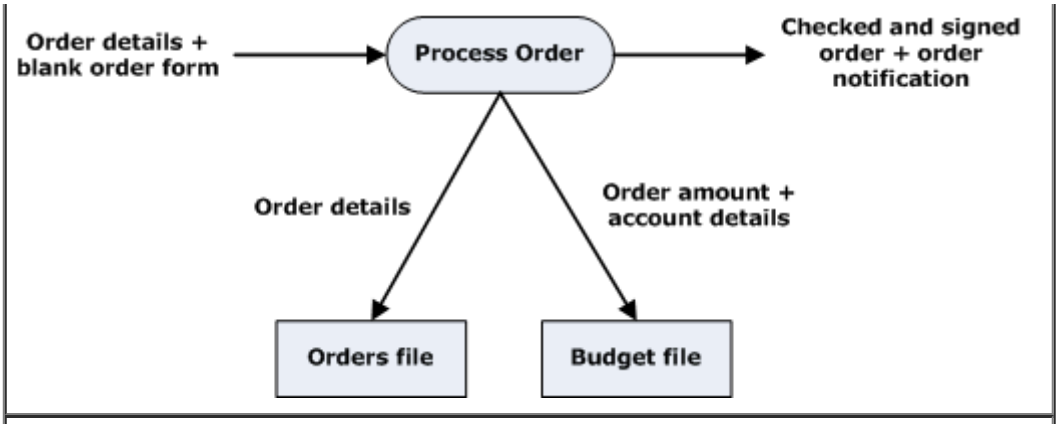

Figure 4.1 Level 0 data flow diagram for order processing system.

In a level 0 data flow diagram, the entire system is represented as a single bubble, along with the inputs and outputs. From here, the "Process Order" bubble is gradually broken down to illustrate more details of the system. Figure 4.2 shows what a Level 1 data flow diagram might look like. The "Process Order" bubble has been broken down into three bubbles which separate discrete functions of the overall system. One function is getting the details of the order. A second function is storing the order information. The third function is sending the order.

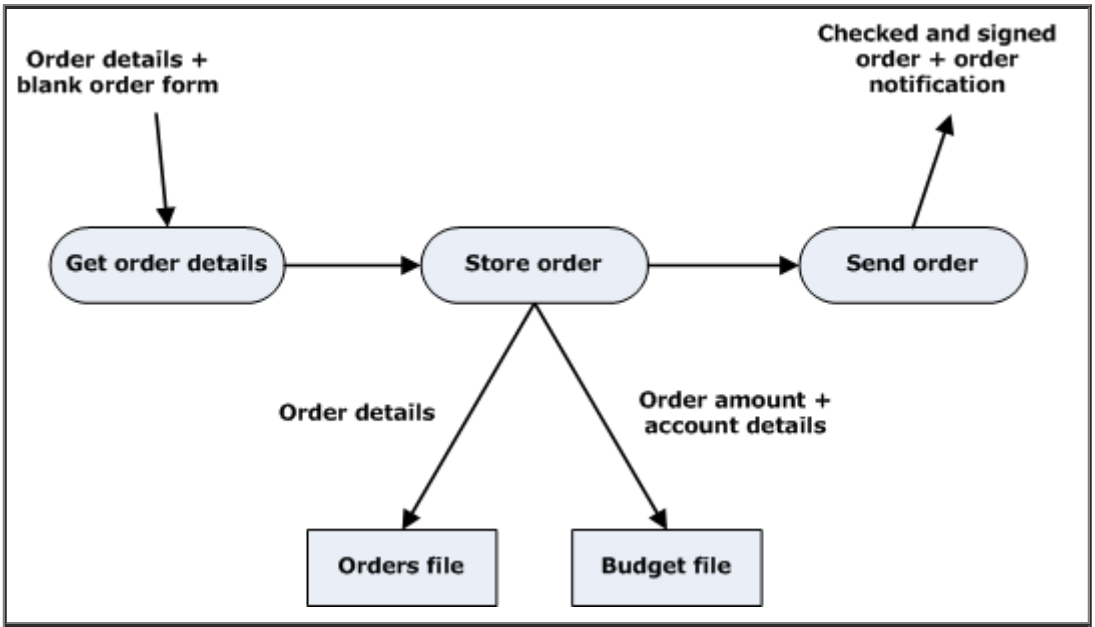

Figure 4.2 Level 1 data flow diagram for order processing system.

We can break down the Level 1 diagram to obtain finer granularity as follows:

1. Break down the "Get order details" bubble into the "Complete order form" and "Validate order" bubbles.

2. Break down the "Store order" bubble into the "Record order" and "Adjust available budget" bubbles.

The degree to which the system is broken down with each succeeding level of diagram is arbitrary. The main purpose is to make it clear how the data are processed by the system. In the end, you should be able to transform the data flow diagram to a corresponding architecture diagram. In other words, a diagram should be obtained in which each bubble corresponds logically to an individual unit (usually a method or class) that will contain the implementation for that particular function.

State-Transition Diagrams

State-transition diagrams, or state machine diagrams, are useful primarily in embedded systems, or

systems with a very limited user interface, where a limited number of buttons or keys must be capable of controlling a relatively large number of functions. In order to use a state-transition diagram the system must be capable of existing in two or more distinct modes of operation, where different things can happen in each mode. The example I like to use to illustrate such a system is a command-line text editor, such as those found on UNIX systems. In such a system, pressing certain keys can have different effects depending on which mode the editor is in. Another good example is cell phones. Most cell phones used to have only 15-20 buttons on them, yet had to allow you to enter contact information using all 26 letters of the alphabet, all 10 digits, and possibly certain special characters. Even many modern cell phones, such as Apple's iPhone, do not allow a full keyboard to be seen all at once. Incidentally, touch-screen technology, combined with increased computing power employed by the iPhone and many other cell phone models, have essentially removed the limitations imposed by the need to have physical buttons present on handheld devices, but the small real estate of the screen means one can still only see a limited amount of controls at a time.

The Data Dictionary

Sommerville doesn't really say much about the data dictionary, but I feel it is important enough to mention. A data dictionary is a central repository of all the names, identifiers, constants, etc. used in the project. You might not bother creating a data dictionary for a personal project, since only you would need to know this information. But if a large team, or multiple teams, were working on a project, a data dictionary would be essential to ensure everyone was on the same page, so to speak. Using a data dictionary promotes consistency in the user interface. Imagine what would happen if you had ten people in ten different locations working on a website, for example, and each person was responsible for designing one section of the site. More likely than not, you would end up with a website that looks like it has been created by ten different people, rather than a website with a cohesive design.

Architectural Patterns

A pattern is a more or less standardized solution to a commonly encountered problem. In the context of software engineering, the term "pattern" is usually taken to refer to design patterns, but there are numerous patterns that can be applied to software architecture—far too many to discuss here, so I will briefly mention a few that Sommerville does not cover in the textbook.

Peer to Peer (P2P)

A peer to peer pattern uses a network of individual computers that are all linked through network connections, where each computer shares a portion of its resources, such as hard drive space, with the other computers (or peers) in the P2P network. P2P differs from the more traditional client-server architecture is in the absence of the dichotomy between the client and server; in P2P, each peer acts as both client and server. One of the most common uses of a P2P architecture is file sharing, with Napster and KaZaA being two of the early applications that popularized P2P for the mainstream. P2P is used for other applications, though, such as the telephony capability of Skype, and cloud computing being two examples.

Interpreters & Virtual Machines

An interpreter is an application that executes the instructions dictated by a programming language. Usually, the interpreter either executes a form of code that is intermediate between source code and machine code, as with Ruby; or executes code that has been compiled from this intermediate code (often referred to as bytecode), as with Java. In some cases, like Java, the code is executed within a virtual machine, which is essentially a self-contained environment within memory that is designed to either emulate an entire operating system, or allow the execution of a single process in a sandbox. The Java Virtual Machine (JVM) is an example of the latter, while the application that allows applications to run in Windows XP mode on the Windows 7 environment is an example of the former.

Blackboard

Not to be confused with the BlackBoard distance learning software, the blackboard pattern consists of a collection of specialized programs that work independently to solve a problem, and share common data. The programs, sometimes called agent programs to reflect their relationship to artificially intelligent agents, do not communicate with each other, nor do they operate in any kind of defined, synchronized manner. Each program operates according to its own set of rules. This type of architecture is most

often used for very complex problems where there is no clear cut solution, or well defined strategy for finding a solution. The agent programs, through their anonymous collaborative efforts, approximate a solution to a problem. An example of an application that uses the blackboard architecture is the Hearsay II speech recognition software.

Broker

The broker pattern is used with distributed systems to allow components residing on different computers to interact with each other transparently to provide some service. Transparency refers to the decoupling, or hiding of implementation details, of the individual broker components. Components can be written in different languages, and executed on different operating systems, without compromising the overall functionality of the system. CORBA (Common Object Request Broker Architecture) and Microsoft's OLE (Object Linking and Embedding) architecture are two examples of software that use the broker pattern.

Application Architectures

In the last module we discussed the architecture of software from the system standpoint. The various architectures we talked about were pretty generic, and we covered some design techniques that can be useful for all systems. However, many of the software applications developed today tend to fall into one of several categories, based on user needs. There are four broad categories that together cover most of the applications written today. The terms, system and application, are often used interchangeably, as you may have already noticed. In most cases the major distinction is that an application can reside within a system, but usually systems do not reside within applications.

Data Processing Applications

Virtually every business-related or IT application focuses on the processing of data—customer records, patient records, client records, etc. The application takes input data, processes the data, and either stores the output or generates a report. These applications always have a back end (a database), and usually have a front end (the user interface) for data entry. Some data processing applications are designed to run exclusively in the background to perform such tasks as backing up a database, or issuing notices to customers who are late with payments, for example. The databases used by data processing systems can be enormous, typically much larger than the system itself.

Transaction Processing Applications

The key term to understand here is **transaction**. A transaction is defined as a sequence of logically related operations, all of which must be correctly completed before the transaction can be considered complete. If any one of the operations fails, or produces an incorrect result, the transaction is incomplete. The canonical example of a transaction is transferring money from one bank account to another. The instant the money is removed from the first account, an intermediate state exists in which the account owner is "missing" a certain amount of money. This intermediate state is resolved once the money has been successfully added to the second account.

There are a lot of problems intrinsic to transaction processing applications. It is vital for these applications to be designed with some degree of failure recovery, since there are many ways in which a transaction can be disrupted. A few examples include:

1. a loss of power in the middle of a transaction
2. faulty code that rounds numeric data inconsistently
3. the computer containing a crucial database goes down and must be restarted or repaired

Many database systems provide rollback capabilities to restore the state of the data in the event a transaction fails. Commit capabilities are provided to make sure successful transactions are made permanent. Despite these precautions, it is still possible for the data to become corrupted. To add another layer of protection, multiple copies of the data can be stored, although this adds the new problem of making sure all the copies are consistent with each other. RAID (Redundant Array of Independent Devices) configurations are also sometimes employed, where the data are stored across

multiple devices (e.g., hard drives) rather than a single device. If one device fails, at least part of the data can still be accessed until the failed device is replaced.

Concurrency is also a major problem of transaction processing systems. When two or more customers (or more commonly, software processes) are allowed to modify the same data, care must be taken to ensure deadlocks do not arise.

Related to concurrency is the problem of synchronicity, or timing. This is particularly important in multi-threaded applications where separate threads can behave independently of each other. As an example, suppose two transaction threads are spawned to modify an account containing $1000. One thread is supposed to compute monthly interest (5 %) on the account and add it to the balance. Another thread is responsible for reconciling a pending debit of $1025. If the first thread's transaction is processed first, the final account balance will be $25. If the second thread's transaction is processed first, the final balance will be -$26.25 (or -$25, depending on how the system deals with negative balances). As this example shows, the timing of transactions can have disastrous results. One possible solution that would minimize the above problem would be to perform all credit transactions before performing any debit transactions.

Security problems can also plague transaction processing systems. The system must be resistant to intrusions, denial of service attacks, and the like. Sensitive data must be encrypted as a failsafe in the event the system is penetrated. Implementing system security is tricky, since security measures can be (and are) compromised, necessitating new security measures, which can then be broken, etc.

One final consideration I'll mention is storage. Depending on the usage, a transaction processing system can potentially handle millions of transactions per day. In addition, most organizations routinely archive transactions after a certain amount of time has passed. Eventually, this can consume enormous amounts of storage space. Such systems must be designed to ensure adequate storage capabilities exist, and that the appropriate type of media (hard disk, tape backup, CD-ROM, etc.) is used.

Event Processing Applications

Virtually every modern software application has a user interface aspect, and as such is an event-driven system. The application waits for the user to do something, then responds when the user presses a key or clicks a mouse button. There are two basic ways in which this can occur:

1. The application can periodically poll to see if a key or mouse button has been pressed. Essentially, an infinite loop executes until the user does something.

2. Interrupt handlers can be used to respond to hardware events. No loop is used here. Instead, hardware events are caught by the interrupt handlers, which redirect execution to a specific process.

Each method has advantages and disadvantages. Polling can result in the CPU spending a lot of time doing no real work, reducing its availability for other processes that may be running. Using interrupts avoids this, but introduces other problems. Only a limited number of interrupts are available, and conflicts can occur if two applications must respond to the same interrupt.

Language Processing Applications

These applications interpret and process data in the form of some language, be it a natural language used by humans or a programming language used by computers. Compilers, parsers, and translators all fall into this category. The design of a language processing application will almost surely have a grammar component, usually formally specified using Backus-Naur form or some other grammar notation. The choice of data structures is usually an important aspect of the design, with trees and hashtables commonly used.

The generic structure of a translator contains the following components[1]:

1. A lexical analyzer that takes input language tokens and converts them to an internal form.

2. A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.

3. A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.

4. A syntax tree, which is an internal structure representing the program being compiled.

5. A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.

6. A code generator that 'walks' the syntax tree and generates abstract machine

References

1. Software Engineering 9 companion website (http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/index.html), Web Chapter 28: Application Architectures, last accessed 09-12-2010.