Module 5

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

It is recommended that you prioritize the readings: studying the “primary” ones first and then looking at as many of the “secondary” ones as you can. Unless otherwise noted, readings are from Systems Analysis and Design: An Object-Oriented Approach with UML, by Dennis, Wixom, & Tegarden (6th Edition, 2021). In the readings listed below, you are given a page range for the reading, but you are only required to read the subsections that are itemized. If no subsections are mentioned, you are required to read the entire page range.

Module 5 Study Guide and Deliverables

Readings:

Primary Reading for Module 5:

The following readings should be completed after reading the module parts that they pertain to.

- Pages 263-270: Packages and Package Diagrams
- Pages 270-274: Design Criteria (read "Coupling," "Cohesion")

Secondary Reading for Module 5

The following readings are not required, however they provide additional depth and examples for concepts in this module.

- Pages 289-300: Object Design Activities
- Pages 414-421: Physical Architecture Layer Design
- Pages 422-428: Cloud Computing, Ubiquitous Computing and the Internet of Things, Green IT

The *Encounter* video game case study will be referenced in this module.

- Requirements for the *Encounter* Video Game (opens PDF)
- Design of the *Encounter* Video Game (opens PDF)

Assignments:

- Draft Assignment 5: Part 3 of Term Project due Sunday, February 18 at 6:00 am ET
- Assignment 5: Part 3 of Term Project due Thursday, February 22 at 6:00 am ET

Live Classroom:	• Tuesday, February 13 from 8:00-10:00 pm ET - Class Lecture • Wednesday, February 14 from 8:00-9:00 pm ET - Assignment Preview • Live Office: Saturday, February 17 from 1:00-2:00 pm ET

Module 5 Objectives

By reading the lectures and completing the assignments, you will be able to:

- Understand the goals of software architecture
- Understand the meaning of “frameworks”
- Integrate diverse UML models
- Describe a system architecture using design models

Part I: System Design

Introduction

A **system design** is a set of documents on which an application can be fully programmed. In other words, a complete system design should be so explicit that a programmer could code the application from it without the need for any other documents.

Design documents describe what parts the finished software will consist of, how these parts are related, and what the job of each part is.

System designs are like the blueprints of a building prepared by an architect that are sufficient for a contractor to build the required building. System design can be understood in two parts: High-level design is often referred to as “system architecture” and all other design as “detailed design.” It can be beneficial to make designs very detailed, short of being actual code.

This section will cover the following topics:

- The goals of architectures and designs
- The use of data flow diagrams and state diagrams to represent designs
- Integrating use case models, data flow models, and state models
- The meaning and the use of frameworks
- Comparing architecture alternatives

The Goals of System Architectures and Designs

The first goal of a system design is to be sufficient for satisfying the requirements. Usually, system designs must also anticipate changes in the requirements, and so a second goal is flexibility. Another goal of system design is robustness: the ability of the product to anticipate a broad variety of input. These and other goals are summarized below.

Goals of System Design

- **Sufficiency:** Handles the requirements
- **Flexibility:** Can be readily modified to handle changes in requirements
- **Robustness:** Can deal with a wide variety of correct and incorrect input
- **Reusability:** Can use parts of the design and implementation in other applications
- **Reliability:** Acceptable failure rate
- **Security:** Includes components to assure appropriate protection from security threats
- **Efficiency:** Executes within acceptable limits of time, space, and any other applicable resources

These goals sometimes contradict one another. For example, to make a design efficient it may be necessary to combine modules in ways that limit flexibility. In fact, we trade off goals against each other in ways that depend on the project's priorities.

Sufficiency of Designs

This section discusses the first goal of system designs: to provide a list of the components and the set of responsibilities of each component for an implementation that satisfies the systems requirements. In modern approach to design, the responsibilities are implemented as methods that belong to specific classes. To perform their job, the methods access the values of class data fields (attributes) and parameters that are passed to the method from the caller (another method, most commonly in another class).

As a trivial example, consider the method setName(String fname, String lname) of the Customer class. This method implements the responsibility of setting customer name. It uses parameter values passed from the calling method and sets the values of the data fields of a Customer object.

To assess the sufficiency of a high-level design, one needs to be able to understand it. This fact is obvious, but it has profound consequences. It can be difficult to create an understandable design for applications due to the large number of options that are typically available. *OpenOffice*, for example, is a very confusing application when viewed in complete detail. Yet, *OpenOffice* is rather straightforward when viewed as consisting of a few sub-applications such as: word processing, spreadsheets, presentations, and a database application.

Thus, we see that modularity is key to understandability. Understandability is usually achieved by organizing the design as a progression of design decisions from a high-level with a manageable number of parts (with responsibilities specified for each part), then increasing the detail on the design of each part in an iterative fashion. In summary, design sufficiency is closely related to correctness, modularity, understandability, and readability.

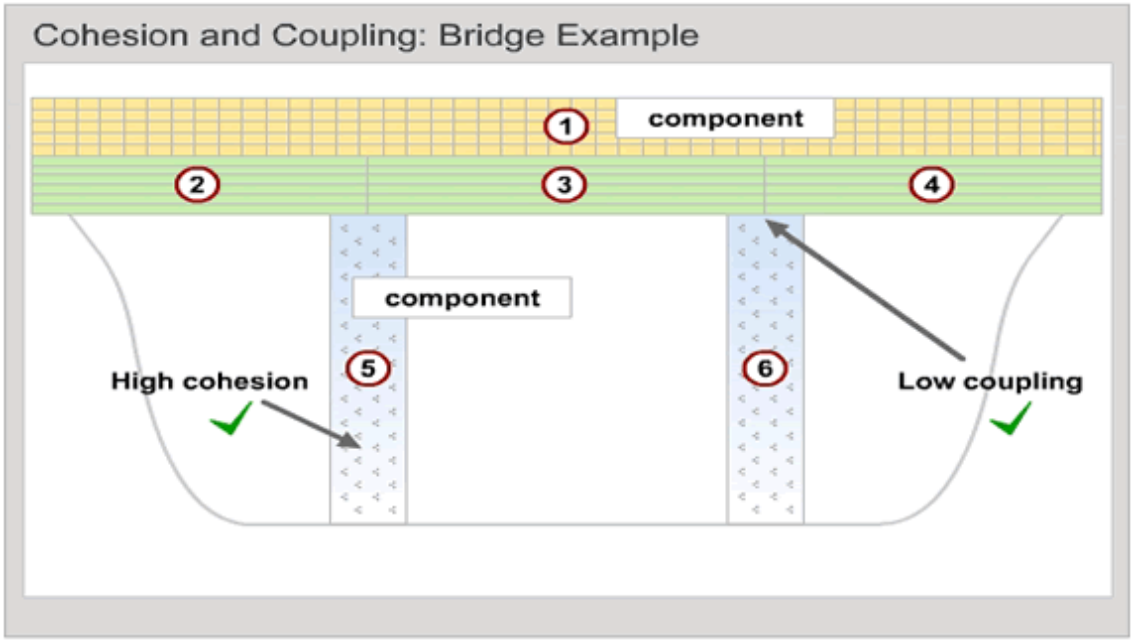
A good system architect and designer forms a clear mental model of how the application components will work at a high-level, then develops a decomposition to match this mental model. He first asks the key modularity questions: what five or six modules should we use to decompose a personal finance application and what should be the responsibility of each module? What four or five modules neatly encompass a word processing application and what does each module do? After deciding this, he turns to decomposing the components into subcomponents, defining the job of each one, etc. This process is sometimes called *recursive design* because it repeats the design process upon smaller and smaller design components at successive levels of detail. At each step, system decomposition involves consideration of cohesion and coupling.

Cohesion within a module is the degree to which the module's components belong together. Coupling describes the degree to which modules communicate with other modules. To modularize effectively, we maximize cohesion and minimize coupling.

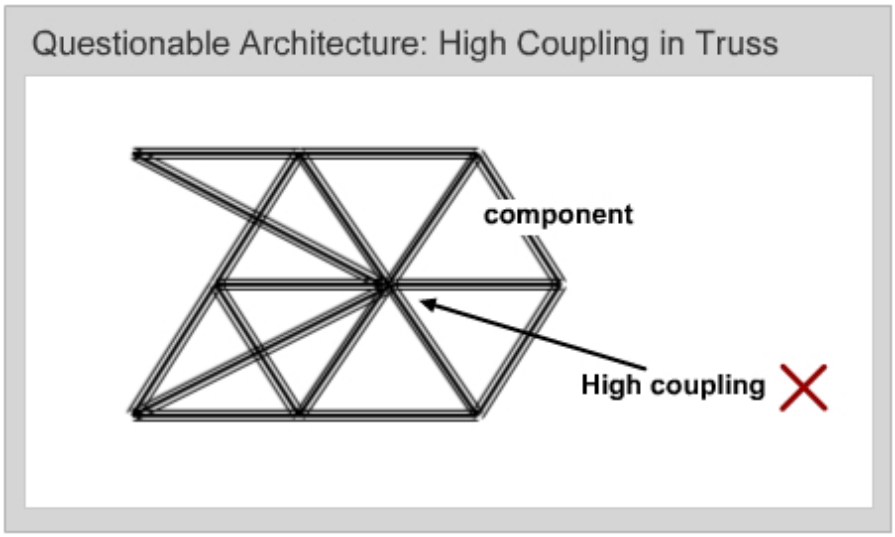
For a simple example, consider related classes Project and Employee. Normally, you would define the first name and last name of an employee as class Employee attributes. These attributes belong together with other Employee attributes, such as employee number, date of hire, department, etc. The programmer who writes a segment of program that needs the employee name to perform its job turns to the Employee object and calls the method *getName()*, which returns a string with the employee name.

Now consider what happens if the designer defines the employee first and last names as Project attributes. Logically, they do not belong together with other Project attributes such as the budget, start date, etc. Technically, the designer can define these attributes wherever he wants. As the result, when the program needs the employee name and turns to the Employee object to call its *getName()* method, the Employee object has to turn to the Project object and ask for the name. We see that the design decision that results in poor cohesion also increases coupling, in this case between the Project and Employee objects. This design also impedes system maintainability. When a maintenance programmer has to change the data type of the employee name and the name is not defined in the Employee class, it takes a greater effort to figure out in what class this data is implemented.

These principles help to decompose complex tasks into simpler ones. The example below suggests coupling/cohesion goals by showing an idealized architecture for a bridge, in which each of the six components has a great deal of cohesion and where the coupling between them is low. The parts of each bridge component are mutually dependent (e.g., concrete with embedded metal reinforcing the columns). This is high cohesion. On the other hand, each component depends on just a few other components—two or three, in fact. This is low coupling.



The steel truss shown below, on the other hand, shows many components depending on each other at one place. We want to avoid this high degree of coupling.



Low coupling/high cohesion are particularly important for system design because we often need to modify applications. Compare the life cycle of a typical system application with that of the bridge in the coupling and cohesion example above. The likelihood that the system will require modification is many times greater than for the bridge. Low coupled/high cohesion architectures are far easier to modify, since by isolating units of functionality we tend to minimize the effects of changes.

The number of top-level packages in an architecture should be small so that people can comprehend the result. A range of “7 ± 2” is a useful guideline, although specific projects can vary greatly from this range for special reasons. The difference between small- and large-scale projects is the amount of nesting of modules or packages. Projects typically decompose each top-level package into sub-packages, which are then decomposed into sub-sub-packages, etc. The “7 ± 2” guideline applies to each of these decompositions.

Instead of the term sufficiency, we can also use the term **correctness**, although this term is usually reserved for detailed designs and code. The term correctness implies that when the designer decomposes a component into its subcomponents, the designer allocates responsibilities (data attributes and methods, or messages) to each subcomponent so that the subcomponents correctly implement the responsibilities of the component that the designer is decomposing.

The Other Design Goals

This section discusses goals of design other than sufficiency (correctness).

Robustness

A design or implementation is robust if it is able to handle unusual conditions. These include bad data, user error, programmer error, and environmental conditions. A design for the Video Library application is robust if it deals with attempts to enter DVDs with wrong or inconsistent information, customers who don't exist or who have unusually long names, or late rental situations of all kinds. Robustness is an important consideration for applications that must handle communication.

Flexibility

The requirements of an application can change in many ways, as shown below. This requires **flexible** designs.

Aspects of Flexibility

- ***By isolating units of functionality we obtain more or less of what's already present***
  Example: handle more kinds of accounts by adding more code without needing to change the existing design or code
- ***Adding new kinds of functionality***
  Example: add *withdraw* to existing *deposit* function
- ***Changing functionality***
  Example: allow withdrawals to create an overdraft

A set of previously used designs is a useful resource for flexible designs. We design so that parts of our applications can be reused by others and ourselves on similar or even on different projects. The list below shows artifacts that can often be reused.

Types of Reuse

We can reuse...

- **Object code** (or equivalent)
  Example: sharing DLLs between word processor and spreadsheet
- **Classes** - in source code form
  Example: Customer class extended and used by several applications
  Thus, we write general code whenever possible to allow for specialized extension of classes. An example might be CreditCustomer or CashCustomer
- **Assemblies of Related Classes**
  Example: the java.awt package contains classes for implementing GUI with standard GUI controls (butons, text fields, etc.)
- **Design Patterns** of Class Assemblies

  Example: the Strategy Pattern, the Façade Pattern, etc. can be used in a variety of applications

For another example of class reuse, consider the class *DVDRental* that associates the DVD and the customer renting it. Such a class is reusable only in another application dealing with the rental of DVDs—which is limited. If, however, we had designed a *Rental* class dealing with the rental of an *Item* to a *Customer,* and if we were to have inherited *DVDRental* from *Rental*, then we would be able to reuse Rental for other applications due to its generality. Design patterns facilitate the reuse of assemblies of related classes.

Efficiency

Efficiency refers to the use of available machine cycles, memory, communication bandwidth, storage, and other resources. We create designs and implementations that are as fast as required and which make use of no more than the required amount of RAM and disk space. Efficient designs are often achieved in stages. First, a design is conceived without regard to efficiency; efficiency bottlenecks are then identified, and finally the original design is modified. Suppose, for example, that we wanted DVD rentals to be fast. We might first design a *DVDRental/DVD/VideoLibraryCustomer*combination, then perhaps identify the *rent()* method in *DVDRental* as a performance bottleneck that is requiring speedup. This would fan out into an investigation of the methods that *rent()* depends on, that is, the methods that *rent()* calls in the course of executing its code. The result will be identifying focused bottlenecks, such as disk fetches. These bottlenecks would then be eased with techniques such as pre-fetching movies that the customer is likely to rent to decrease the time spent on waiting for the results of disk accesses.

Reliability

It is unrealistic to expect that a real-world application be 100% defect-free. An application is reliable if it is relatively defect-free. Metrics make this quantifiable, such as the average time between failures. Although reliability is related to robustness, it is different. Robustness is mostly the result of a good design. On the other

hand, reliability is mostly a process issue, requiring thorough inspection and testing of artifacts. Design affects reliability in that a good design that is well understood will make it easier for developers to produce error-free applications.

Usability

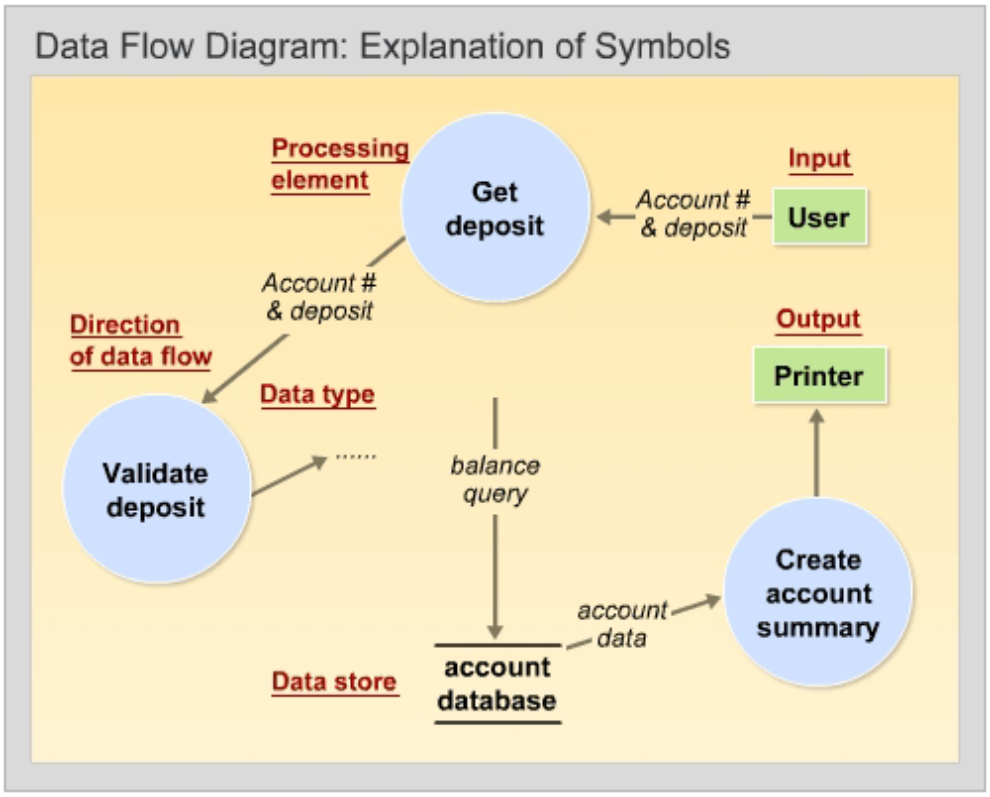
An application has high **`usability`** if users find it easy to use. Usability is attained through human interface design, discussed previously in this course.

The next section explains ways in which designs can be expressed. They augment our previous discussion of UML and are assembled as related “models.”

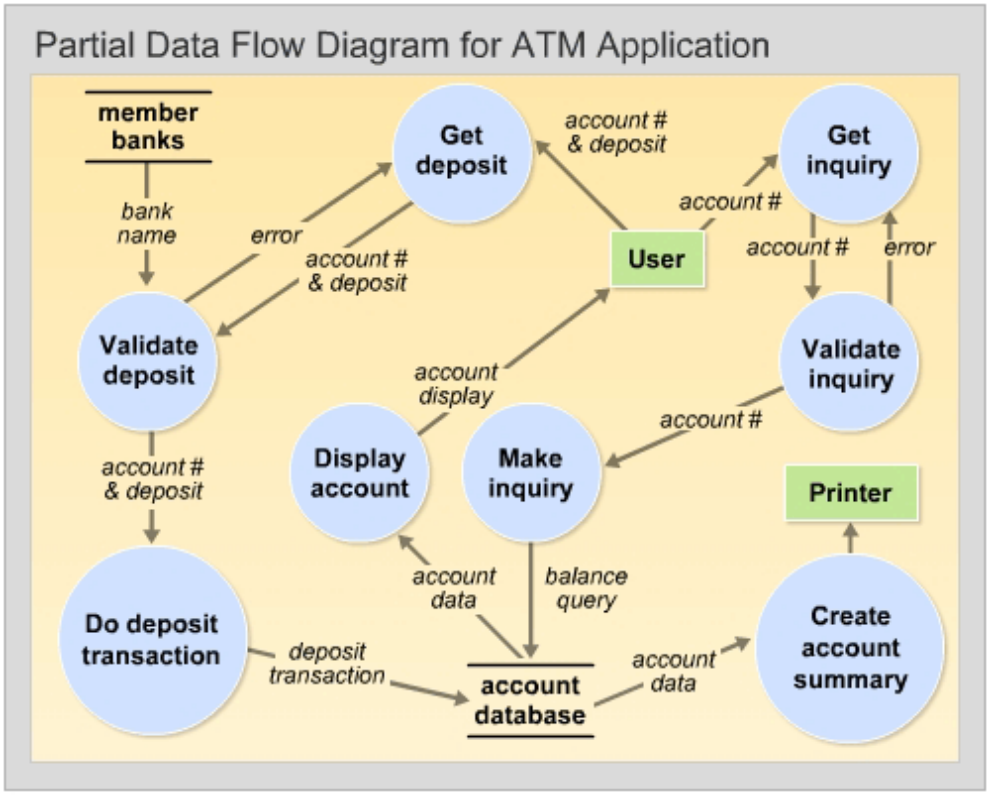
Data Flow Diagrams

There are two types of data flow diagrams: *logical* and *physical*. The operations in a logical DFD are carried out by the system. An example is “Credit Bank Account.” Note that there is no specification of where this operation is performed. (It may even be performed by several components acting together.) For a physical DFD, on the other hand, the operations are each associated with an actual physical entity such as a computer.

The data flow represents the type of data that can flow between functional processing. Data flow notation is shown below.

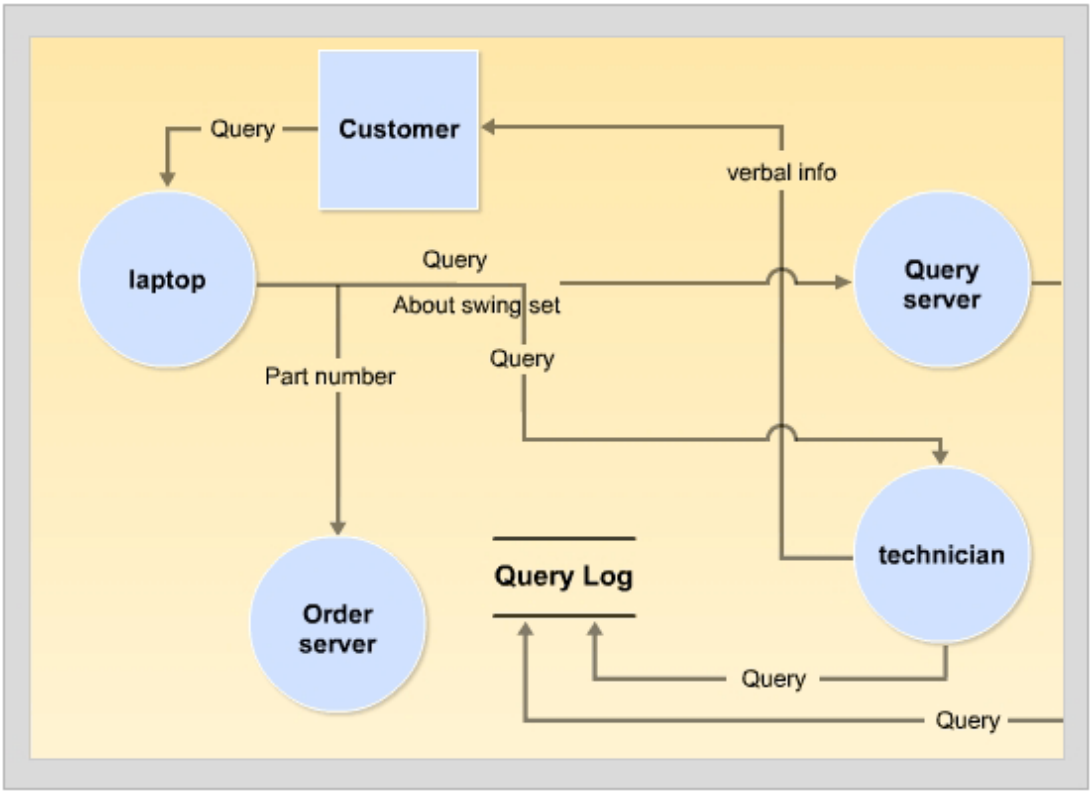


The example below shows an application of this notation to an ATM logical DFD design.



For a physical DFD, we indicate the physical representations of logical elements in the diagram—processes, data stores, and data flows. For processes, we indicate what class is responsible for the operation performed by the process. Similarly, for data stores we indicate the class of objects that are stored in the store and the type of storage (such as a flat file, a database table, etc.). For data flow between two processes or between a process and an external actor, the name of data flow suffices. For data flows between a process and a data store, we indicate the operation performed (read, write, or update).

The figure below (constructed with Visio) is a partial physical data flow for a system that assists customers who need help maintaining an outdoor swing set. Note that *people* can be part of such a DFD. Ideas for the parts of such a data flow can be obtained by thinking about the physical components involved in the system and about the types of data flowing within them. It can help to actually go through documents (even marketing material), highlighting the words you encounter that are physical entities or types of data.



Dataflow diagrams are not helpful for all applications. For example, they do not appear to add much to the *Encounter* video game case study. They are often helpful in describing systems with many real-world inputs and outputs, independent bodies of data, and processes connecting the inputs, outputs, and data.

Relationship between States and GUIs

When an application is displaying a GUI, one can think of it as being in a particular state. Mouse or keyboard actions are **`events`** that affect this state. Although we may well want to define states that do not correspond to GUIs, and vice versa, a ***`State`*** ⟵ ⟶***`GUI`*** correspondence can be made for many applications. For the Video Library application, we actually showed transitions between GUIs instead of between states.

The idea of having *states* occurs commonly in systems with limited input options and different responses to those according to the system inputs entered previously. A typical ATM, with its fixed keypad of just a few keys, is one example:

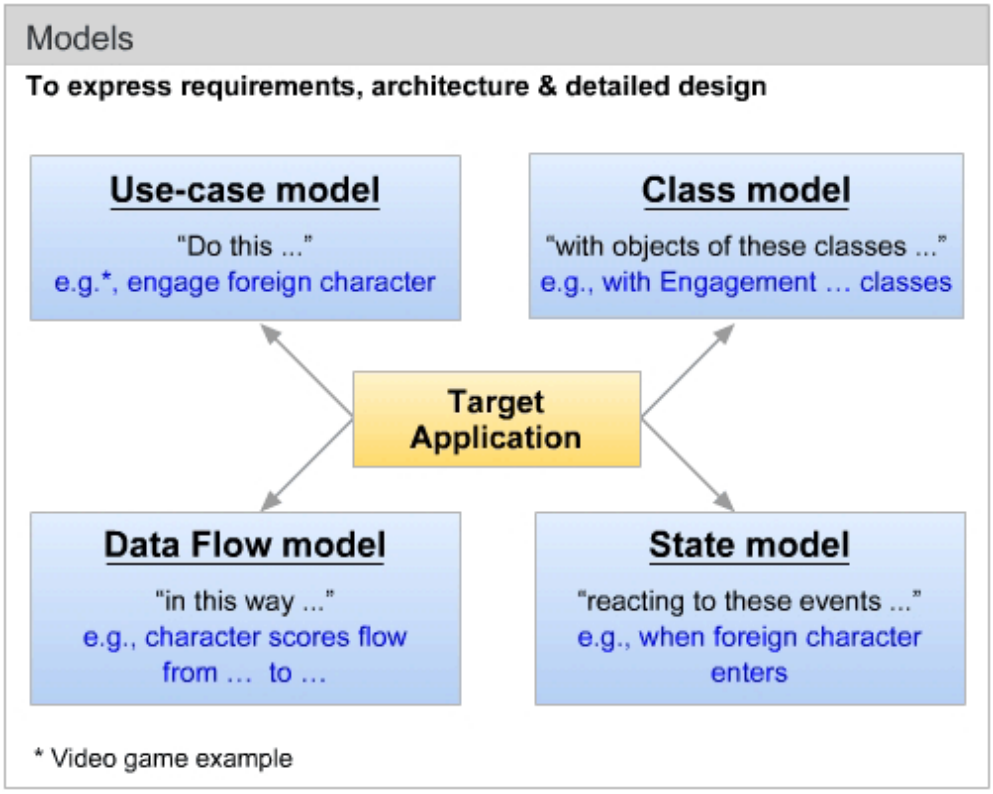
- After loading your ATM card, numbers on the keypad are treated as digits in your PIN.
- Once your PIN has been accepted, numbers on the keypad represent choices of deposit, withdrawal, balance query, or other transaction.
- Once you've chosen to make a deposit, numbers on the keypad represent the dollar amount of the deposit being made.

And so on. Each state (or *mode*) interprets the numeric keys in a specific way, but in a way different from how the other states use the keypad. Very roughly speaking, there is one GUI panel for each different state, but systems with no UI can also be described in terms of states.

Integrating Design Models

The architectural drawings of an office building comprise the front elevation, the side elevation, the electrical plan, the plumbing plan etc. In other words, several different views are required to express a building's architecture. Similarly, several different views are required to express a system architecture. They are called **`models`**. We have already used some of these models in previous modules.

The four important models are shown below. Several ideas here are taken from the Unified Software Development method of Booch, Jacobson, and Rumbaugh (1998). The following example refers to the *Encounter* video game example.

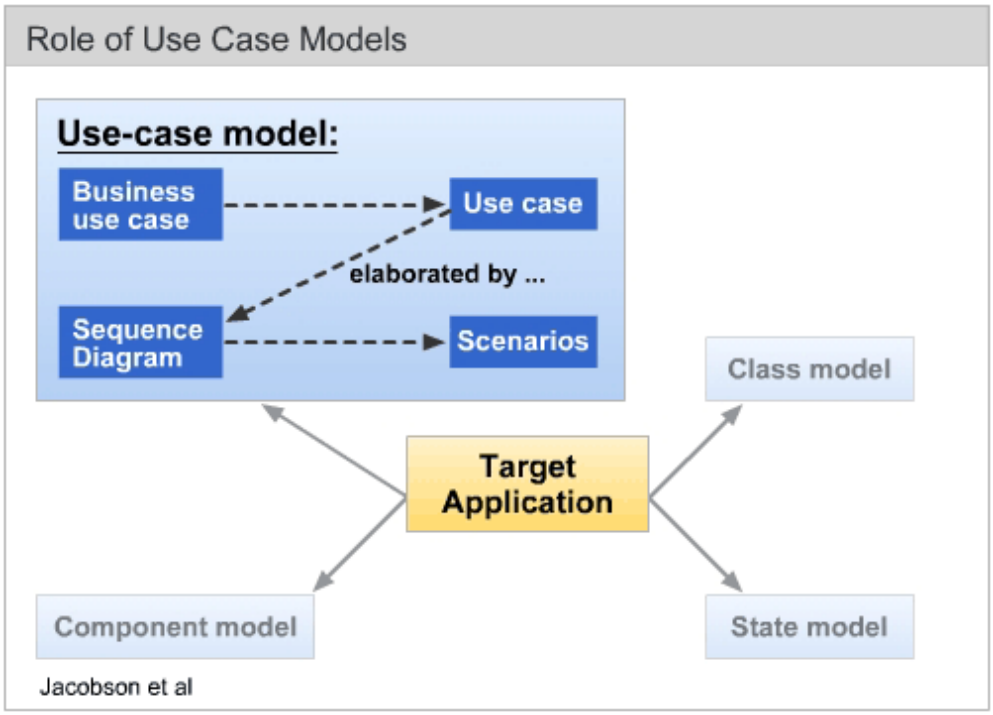


Use Case Model

This section describes several levels of use cases and related concepts. The **`use case model`** consists of the following four parts.

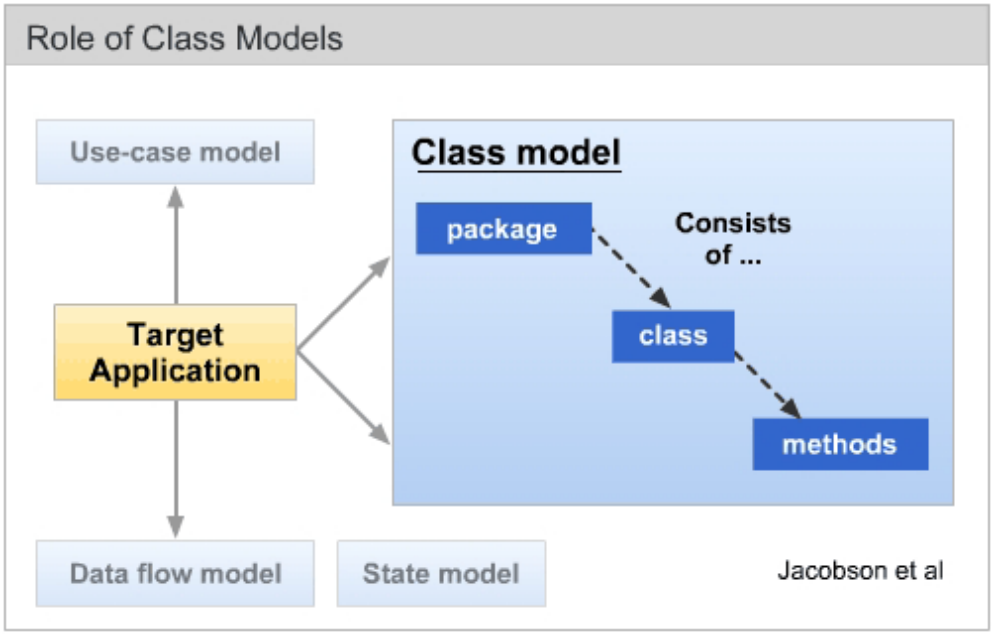
1. The **business use cases**—a narrative form, suitable for developer-to-customer communication, describing the required basic sequences of actions
2. Their refinement into regular **use cases** described in previous modules
3. Their transformation into **sequence diagrams**, which in turn can be in two successive stages of refinement:
    1. With informal functionality descriptions, at first lacking all details, then
    2. With specific function names and showing all details
4. **Scenarios**—instances of use cases which contain specifics, and which can be used for testing. For example, a scenario of the use case step:
   Customer chooses account
   would be something like:
   John Q. Smith chooses checking account 12345.

The use case model expresses what the application is supposed to do, as suggested below.



Class Models

Classes are the building blocks—more precisely, the types of building blocks—of designs. We have been dealing with class models throughout this course. Here we focus on the fact that class models may be decomposed into packages that decompose into smaller packages which decompose into classes, and these, in turn, decompose primarily into methods and data. A package is an architectural element since it exists at a higher level than a class. A package is a set of classes that belong together conceptually. This is illustrated below.



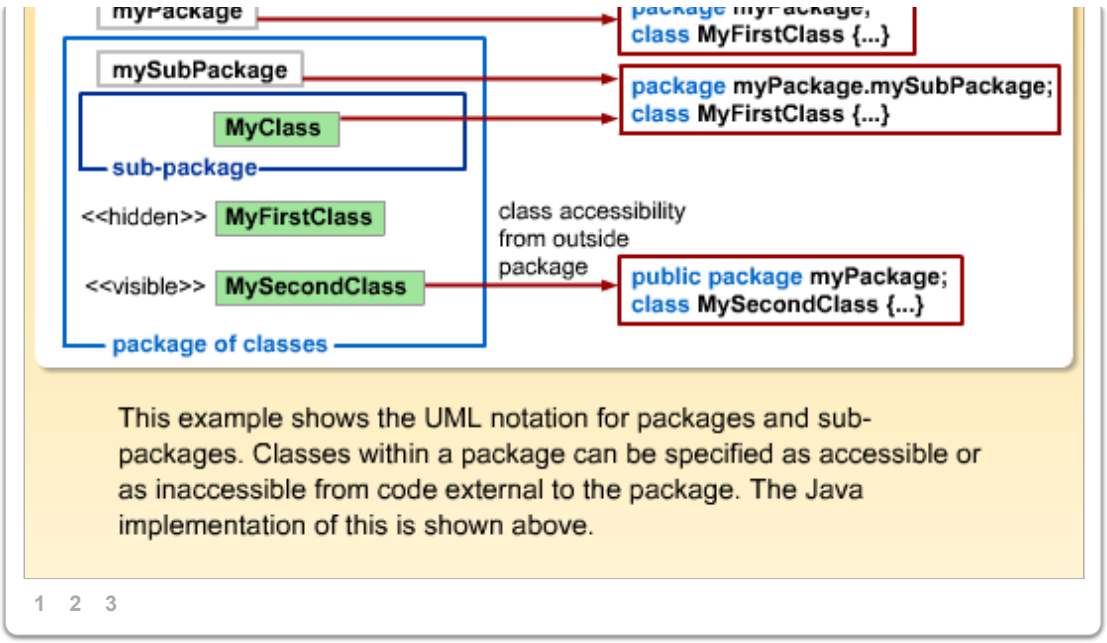
Packages

The Unified Modeling Language uses the term package for collecting design elements such as classes into assemblies that work together to achieve common goals. “Package” also happens to be the name of collections of Java classes (but not C++ classes; in C++ a similar concept is called namespace). Java packages translate into file directories on the computer; their sub-packages decompose into subdirectories etc.

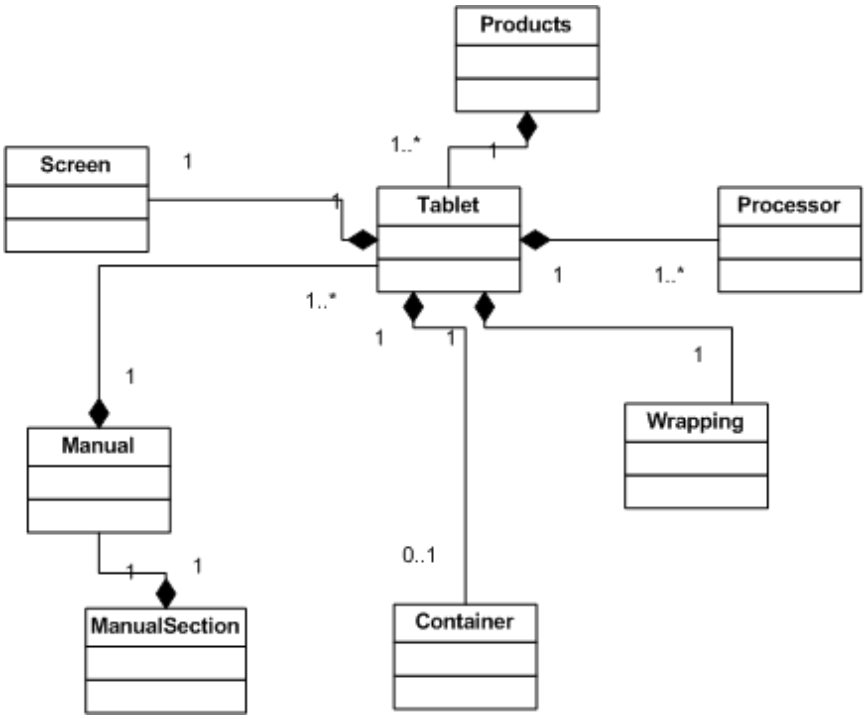
UML packages can contain any materials associated with an application, including source code, designs, documentation, etc. The example below shows the UML notation for packages and sub-packages. The package myPackage has three components; the package mySubPackage and two classes, MyFirstClass and MySecondClass. The package mySubPackage has only one component, MyClass.

Notice the use of naming and capitalization conventions, especially for names that are composed from several component words—no spaces or special characters are used, all name components are written without spaces. For classes, each name component, including the first one, starts with a capital letter. For packages, for data attributes, for operations, for parameters and for actual arguments the first component starts with a lower case letter, and other name components start with a capital letter (so called camel notation—to someone with vivid imagination, the name looks like a camel with humps).

Classes within a package can be specified as accessible or as inaccessible from code external to the package. The Java implementation of this is shown below on the right hand side. Click *Play* in the animation below to see how these concepts are applied. In the UML diagram, the visibility of components is denoted by class access modifiers, the notation delimited by double angle brackets.



Here is an example of a class model (using Visio) and its packaging. The model is for a system that controls inventory of items for a factory that makes tablet computers.



We can assume that the number of classes will grow but the following is a reasonable decomposition of a collection of classes into packages. Often, the latter are denoted in lower case.

Package	Classes in the package
products	Product, Tablet

parts	Screen, Processor
packaging	Wrapping, Container
userManual	Manual, ManualSection

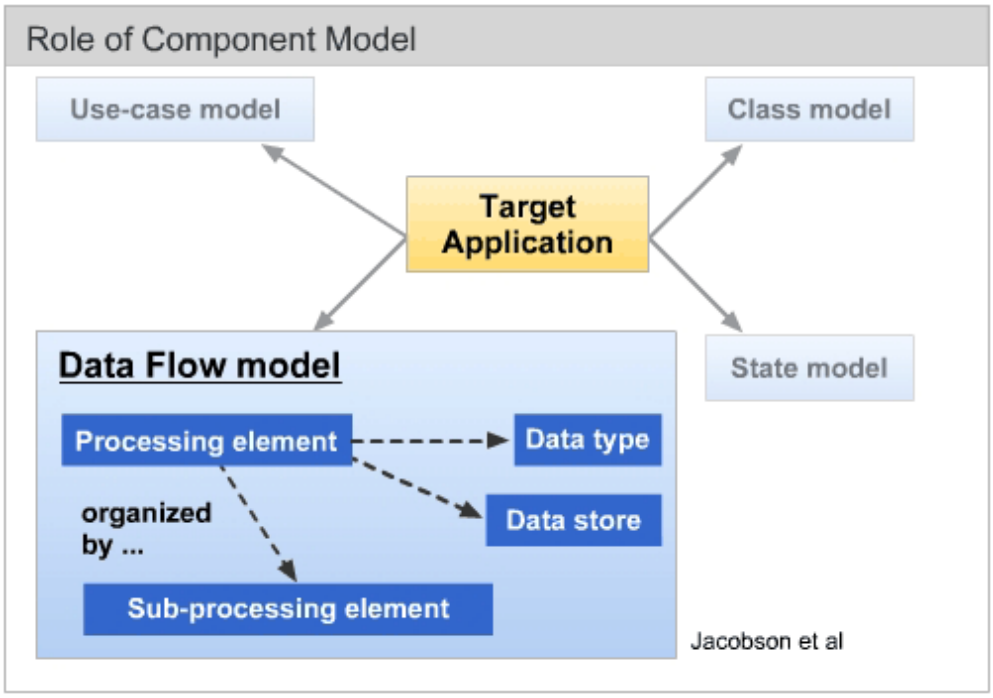
You can see in this decomposition the beginning of an architecture for the factory system: the products, the parts, the packaging, and the user documentation.

Data Flow Models

The class model describes the *kinds* of objects involved; it does not show the actual implementation of the different types of objects. The data flow model, on the other hand, shows specific processes and the types of data flowing between them. It is related to the class model because the processes involved must be allocated to specific classes in a class model and implemented as methods of these classes.

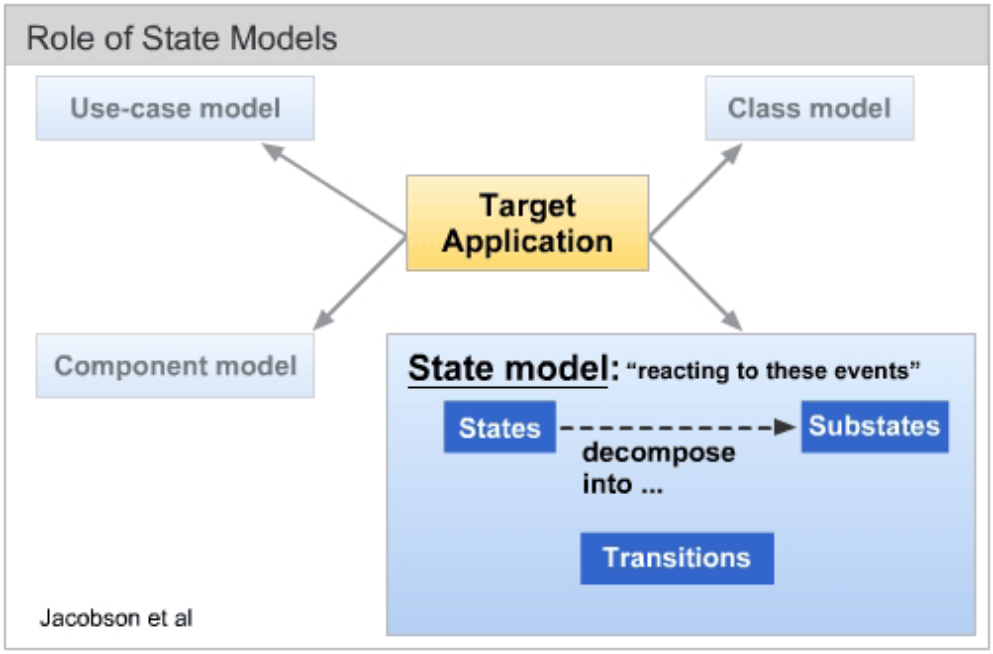
Data stores are implemented either as database tables or as class objects which serve as either sources of data to be processed or as sinks for data produced by the processes.

We discussed data flow diagrams above. The following example shows the parts of a data flow model.



State Models

State models reflect reactions to events. Events include mouse actions and changes in variable values. Events are not described in class models or component models, but they do occur in use case models. Their role is shown below.



Frameworks

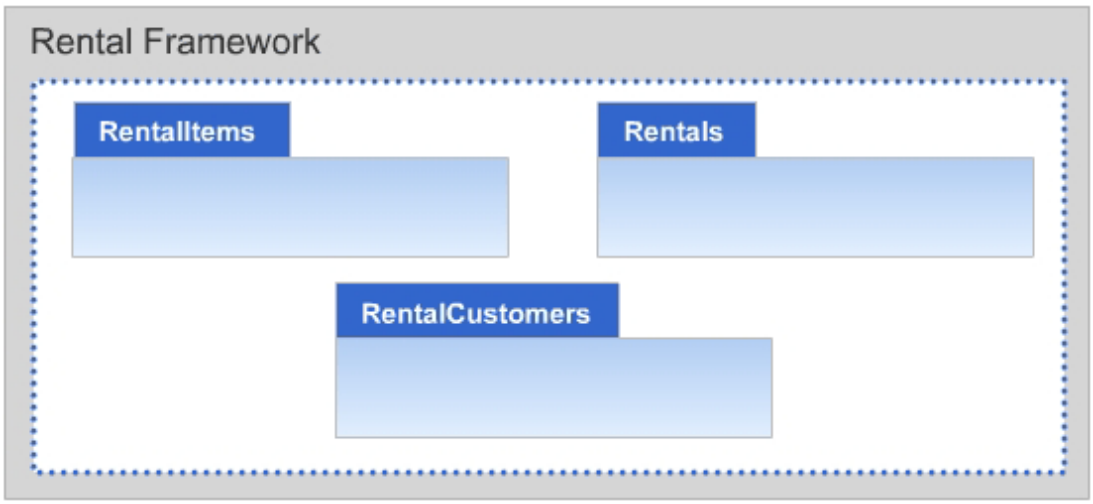
As we have seen, the reuse of components is a major goal in system development. If an organization can't leverage its current investment in software by reuse, competitors who do so will be faster to market with superior products. The parts of an application that are specific to this particular software and contain much of the systems business logic are referred to as Business Objects. Business Objects are essentially the domain classes discussed in previous modules. These are sometimes called entity classes.

Where do we keep the classes slated for reuse? How do we organize them? Should we build in relationships among these classes? Do they control the application or does the application control them? The computing community has learned from experience that merely making a list of available functionality does not necessarily result in much reuse. We have learned, however, that arrangements like the Java API (coherent sets of classes) do indeed lend themselves to highly successful reuse. The Java API's (*3D, 2D, Swing*, etc.) are **`frameworks`**.

The Meaning and Usage of Frameworks

A framework is a collection of software artifacts usable by several different applications. These artifacts are typically classes, together with the additional software required to utilize them. In the simplest form, it is a library of classes that an application can use to accomplish its goal. In a more sophisticated case, a framework is a kind of common denominator for a family of applications. Progressive development organizations designate selected

classes as belonging to their framework. Typically, a framework begins to emerge by the time a development organization develops its second to fourth revision for a particular application. As an example, consider the *Rental* framework shown in the example below that our Video Library application could use.



The framework classes shown on this diagram implement functionality that is common to a family of similar applications, e.g., search for a rental item in the database. The individual classes in the framework specific to this particular application will be supplied later.

Classes within a framework may be related. They may be abstract or concrete. Applications may use them by means of inheritance, aggregation, or dependency. For example, if we want to search the database for specific items—DVDs in one application, books in another application—we might wind up writing separate methods for each application, even though the search algorithms are the same. To achieve reusability, the framework might contain an abstract class RentalItem and search methods written to handle RentalItem. For a DVD application, the DVD class extends RentalItem, and for a book application, the book class extends RentalItem, and both applications are able to reuse the framework search method without any modification.

The example below shows the relationship between framework classes, domain classes, and the remaining design classes. The complete design for an application contains all of these class types. Usually, we do not use all of the framework classes in an application. The design classes consist of those required for the architecture and those that are not.

In general, a design class is any class that we introduce into the model for convenience, to promote reusability, and to avoid the creation of large classes with diverse responsibilities. Any call that is not a domain class (entity class) is a design class. Special cases of design classes are GUI classes (interface classes), persistence classes (database interface classes), and control classes. For the Video Library application, the *RentalItem* class is a design class introduced for the convenience of extending handling DVD classes to other types of rental items. Another kind of design class is one that pulls together classes that need a source of aggregation. For example, a class model may have *Stock, Bond, RealEstate, and Cash* classes. It may then be recognized that a *Portfolio* class is needed (if not already present) that aggregates them.

The main benefit of using frameworks is to provide reusability of components common to the family of applications, such that other developers designing another application of the same family do not have to re-create the same types of components. Organizations will create their own internal frameworks that benefit the application developers for that organization only. Other organizations will develop frameworks for application developers outside their organization. Examples of commonly used frameworks are Adobe Flex or .NET.

Actually, the term *framework* could have a variety of different meanings. The common denominator of these meanings is the concept of a containing or supporting structure. When it is applied to software development, the containing structure concept is implemented as a set of classes that embody a design that could be used for a number of similar but still different problems. Usually, these classes are abstract because different problems require different implementation of similar features.

For example, you might want to develop a family of editors, e.g., a text editor, a stick figure editor, an apartment plan editor, etc. Obviously, combining so many diverse editors in one program is not practical. First, it will extend the time of development. Second, people will not want to pay for capabilities they do not use. However, different types of editors do have similarities—they all need to open a file, select some of its contents, cut and paste, save changes or discard them, and so on.

The skill of the designer here is to identify these common capabilities and implement them as a set of abstract framework classes and design algorithms for handling these classes. Then concrete classes can be developed for each kind of editor. These concrete classes will inherit from the framework abstract classes and reuse framework algorithms.

Developing, debugging, and testing a framework may require significant resources, but when it is completed, you can develop specific applications quickly and with high quality by extending the components of the framework.

An interesting example of a framework is the Zachman framework. The framework provides a conceptual structure for a generalized description of enterprise information technology. It refines the observations made with other methods and defines a table of many rows and columns. The table describes information technology concepts and artifacts independent of business enterprise specifics. For example, the intersection of the Business Knowledge column and the Systems Owners row defines the information scope and vision. The intersection of the Business Knowledge column and the System Designers row defines the physical database specification for any enterprise.

Is this framework useful? If you observed many information systems developed for many different enterprises, you will study the Zachman matrix and say, “Yes, it describes generalized IT characteristics correctly.” If you are only studying information technology, the Zachman matrix might be too general for you, and you might be more interested in specific, concrete examples.

Similarly, is the editor framework described above useful? Yes, if you are going to develop several editors. If you need only one editor, then spending time and effort building a framework might be overkill.

System Architectures

One reason we think at the architectural level is the same reason we think about “ranch” house architectures or “colonial” architectures when considering designing a house. Architecture concepts are reused repeatedly. In describing the architecture of a bridge, we use terms like *suspension bridge* or *cable-stayed bridge*. These terms allow us to envisage an intellectually manageable number of parts (3–12 perhaps, rather than hundreds). In other words, system architecture is primarily a psychological concept concerning acceptability to the human intellect. Similarly “architecture” for a system application is its high-level design enabling system analysts to gain understanding.

When an application is to be implemented by the object-oriented paradigm, the design is a collection of classes. If there are only 10 classes in the entire application, we may regard this decomposition as the architecture. More often, however, there will be hundreds of classes. Such a set of classes alone can no longer suffice as the architecture because it is too complex to comprehend. In UML, class collections are known as packages, so system architecture consists of a collection of packages.

Since the choice of architecture is so influential on the project, good designers create several alternatives from which to choose. We will list two alternatives for the Video Library application.

The requirements and domain classes help us to select an architecture. The following example summarizes a common way in which we arrive at a class model for an application.

The domain classes are obtained by performing requirements analysis, as outlined in previous modules (step 1). The classes in the architecture (step 2) are arrived at by evaluating your system as a whole and identifying the supporting classes required in addition to your domain classes. Typically, the framework classes already exist before we begin the current application; if not, your framework classes are obtained by collecting architecture classes for applications like the one under development. Finally, the rest of the classes are added to complete the design (step 3).

Trading Off Architecture Alternatives

Since the choice of a system architecture is important, it is wise to create more than one alternative for consideration. We will create and compare two architectures for the Video Library application as an example.

For the first candidate, we separate the application into three major parts. This is referred to as 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 system architectures. We may

want a logical (conceptual) view of an application to be entirely independent of the hardware platforms hosting it. Applying three-tier to *Video Library*, we could obtain the architecture as shown below.

One strength of this architecture is the fact that it is easy to understand. Another strength is that, since each logical tier is separate, changes to one have minimal impact on another tier. As long as the layer interfaces remain the same, then changes to the GUI will not have an impact on rental operations contained in the middle tier. One weakness is the strong 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 strong coupling between the classes in the *VSData* package and the classes in *VSOperations*.

A second architecture candidate is shown below.

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. Another option would be to group all displays in a separate package. A summary below shows one opinion of these architecture alternatives.

Comparing Architectures		
	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 Library rentals.	Yes: Easy to generalize to generic rentals
Easy to construct?	Perhaps	Yes: Clear potential to use façade

Selecting a Basic Architecture

This section is summarized below.

To Begin Selecting a Basic Architecture

1. Develop a **mental model** of the application
   – As if it were a small application
   E.g., personal finance application...
   ..."works by receiving money or paying out money, in any order, controlled through a user interface"
2. **Decompose** into the required components
   – Look for high cohesion and low coupling
   E.g., personal finance application...
   ...decomposes into Assets, Sources, Suppliers, and Interface
3. **Repeat** this process for the components

ATAM Methodology in Architecture Selection

As we saw, a critical step in choosing a system architecture is defining and performing an architectural analysis. What is of paramount importance in the choice of system architecture is that the system architecture is meeting the functional and non-functional requirements of the system. There are several different ways in which system architecture can be analyzed and then built. One methodology to do it is called ATAM (Architectural Trade-off Analysis Method). It is based on performing an architectural trade-off analysis (Software Engineering Institute, http://www.sei.cmu.edu/library/abstracts/reports/00tr004.cfm).

The ATAM methodology focuses on how a system architecture and design will meet the quality attributes and non-functional requirements defined by the stakeholders of the system. ATAM provides a template for how to try to measure the need of those non-functional requirements based on user scenarios. Some user scenarios describe user interactions with the system similar to use cases. Others describe changes to the system load (the number of transactions processed) or more drastic changes, such as migration to a different operating system.

Each user scenario is analyzed then, depending upon the available development schedule and the potential limitations of the system to be developed, trade-offs are made. The main purpose of ATAM is to assess the consequences of decisions in light of quality attributes of the system. In essence, ATAM is a risk analysis identification methodology and a means of detecting potential risk within a software system.

The key term in ATAM is a concept of *quality attribute characterization.* To characterize the quality attribute (e.g., response time), the stakeholders describe the stimuli to which the architecture must respond (e.g., increase in system load), how to establish that the acceptable value of the attribute is indeed achieved (e.g., response time does not exceed a certain limit), and what architectural decisions might impact this attribute (e.g., introduction of additional security checks).

When applying the ATAM to risks, one is not trying to enumerate all of the risks within a system and counter them, but instead determine the quality attributes of interest to the stakeholders and understand how those

attributes will be affected by the architectural decisions. As a risk evaluation methodology, the ATAM should be applied early on in the software development lifecycle. Since at the early stages of development one cannot compute quality attributes of the future system exactly, the ATAM is limited to identifying only trends of acceptability for the quality attributes.

So, the ATAM process consists of three steps: (a) identify quality attributes that might affect the choice of architecture, (b) identify applicable architectures, and (c) evaluate whether each architecture satisfies the quality requirements.

Another feature of the ATAM approach is describing each quality attribute characterization in three separate sections: (a) external stimuli, (b) responses, and (c) architectural decisions.

The *external stimuli* are events that might cause the architecture to respond or change in some manner (for example, the external stimuli for performance communication messages or user keystrokes that start computations). These changes must be measurable and observable. They should be defined as the *responses* (for example, for performance, responses are latency and throughput, which are measurable). The *architectural decisions* are aspects of the architecture that have direct impact on satisfying the *responses* (for performance, architectural decisions include processors, threads, and scheduling mechanisms for processes).

To provide an example of stimuli, response, and architectural decision, let's take the following user-defined scenario, "The initial load time of the application must be less than three seconds." To define stimuli, response, and then architectural decisions, the ATAM methodology suggests that analysts would ask these kinds of questions:

- What happens if there are multiple users loading the application at the same time? Does the load ceiling of three seconds apply? (This defines a stimulus.)
- Can there be a range as to how quickly the application can load (e.g., 3–5 seconds)? (This defines a response.)
- Will the application reside on a single machine or be loaded from multiple machines? (This defines an architectural decision.)

These questions will help to further refine the quality attributes and will help the architectural team understand the stimuli, responses, and architectural trade-offs that will need to be made.

As mentioned earlier, ATAM distinguishes between three different types of user scenarios: (a) use case scenarios, which involve typical and expected uses of the system, (b) growth scenarios, which describe anticipated changes to the system, and (c) exploratory scenarios, which cover extreme changes that are expected to stress the system. These user scenarios should be focused on non-functional requirements and should be minimally in the realms of Performance, Scalability, Reliability, and Security quality attributes.

Each scenario should contain a stimuli and response from which an architectural decision should be derived. For example, for a use case scenario in which "the system should be able to handle 25 users at a time and keep an average system response rate of five seconds," the stimuli is defined as *"25 users at a time"* while the response

is to *"keep an average system response rate of five seconds."* In order to satisfy this scenario, the architectural team must make architectural decisions that will accommodate this request.

Another type of system architecture analysis method that is focused on quality criteria is a methodology called QFD or Quality Function Deployment. QFD is focused on the transformation of stakeholder requirements into ensuring that quality is engrained within the creation of the components or sub-systems being created. QFD is not only germane to software development but can also be applied to manufacturing processes (actually, it originated in manufacturing). It is highly focused and driven by the viewpoints of market segments, company, or technology-driven needs. Ultimately, QFD helps to ensure that the stakeholders' requirements are represented as engineering characteristics with methods for testing based on prioritization.

Components

Many definitions have been put forward for **`components`**. The definition we will use is *a part (software or hardware) usable without alteration*. Components can generate entities, typically instances, and these instances can be altered. Using an example of a builder working on a window, note that he may paint a particular window (instance). In other words, he sets the value of its "color" property. What he does not alter is the window (model) that he uses.

The Object Management Group's "Modeling Language Specification" (Revision 1.3) defines a component as "a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of a system's implementation, including software code (source, binary or executable) or equivalents, such as scripts or command files." This includes the kinds of software listed below.

Components Can Be...

- source code
  - Classes—one or more, possibly related
- executable code
  - Object code
  - Virtual object code
- other files
  - Images, text, indices, etc

Although it might appear from this list that anything could be a component, note the restriction we place upon components: that they be usable without alteration. If one does alter a component for the use in a specific application, the result is considered a different component.

The diagram below shows the UML notation for components and the interfaces that they may support. The component shown consists of two components and supports three interfaces. Here, the term “interface” denotes a set of methods of this component available for execution from the calling code (that is, from another component). An example of such a method is int sum(int, int). This means that a component that supports an interface with this description must have a method that adds two integers and returns their sum to the caller. One of the interfaces in the diagram is entirely supported by one of the two sub-components, two other interfaces are supported by both components.

The diagram above describes the logical components of the implementation code—classes, interfaces and packages—rather than physical components—executable files placed in different directories or different computers. Modern technology allows developers to combine executable files into modular components that could be reused in different contexts (for different systems) without modification. Application components are combined into physical files that are deployed on appropriate computers.

It is hard to overestimate the significance of this technological development. In older days, executable files were monolithic. When the application had to be modified, the developers produced “patches” that each installation had to apply individually. This was a labor-intensive and error-prone process. One incorrect step in applying the patch, and the application was rendered unusable. With modern approach, using, for example, Java, developers combine executable files of Java-based applications into so called modular “jar” files (jar stands for Java ARchive and is a standard extension for this kind of files). When the application has to be modified, the component is simply replaced with a different jar file (without modification). This jar file can be downloaded from the Internet, and this avoids operators' errors.

A similar technique for deploying physical files on servers packs executable code into so called “war” files (war stands for Web ARchive and is a standard extension for this kind of files.

These jar and war archive files collect software components into physical representations (physical files) deployed on different machines. The graphical representations of component distribution are called *“Deployment Diagrams”* and can be represented in UML using the logical components shown above. Deployment Diagrams can play a key role in determining how a system might scale or how a system would be deployed on a network.

The figure below shows a simple example of deployment of an online video rental store system. The nodes represent physical containers, either participating computers or programmatic containers (files). In this diagram there several nodes typical for a distributed application—the web server, the database and the client. The web server contains a component OnlineVideo war that encapsulates HTML files, servlets and related classes and supporting libraries. The OnlineVideoContainer contains component files searchVideo.jar, Checkoutvideo.jar, and shipvideo.jar. This container can be deployed either on the server machine or on a separate machine depending on geographic distribution of the application. The ClientDevice contains a web browser and is connected with the OnlineVideoContainer through the HTTPS protocol. The OnlilneVideoDB node contains the Oracle database system connected to the OnlineVideoContainer through the JDBC connection.

Boston University Metropolitan College