Module 3: Methodologies and Application Domain Issues

Topics

1. Software Analysis Methodologies
2. Software Design Methodologies
3. Application Domain Issues

The transition from analysis to design can be difficult. Typically, there are multiple implementations for the same requirement. Which is the best implementation? How will the separate implementations for each of the software requirements interact with each other? How will the overall system behave? These are difficult questions for the software engineer to answer. Some of them cannot be fully understood until the software and the system are complete.

Models used during the analysis phase build the foundation for transitioning elements to the design model. In this module we will begin our discussion by reviewing the various elements of the analysis model that are used in structural and object-oriented methodologies.

To assist the software engineer in the task of software design, design principles have been established as guideposts. These design principles remain true regardless of the software development process used (e.g., linear sequential or spiral) or software methodology adopted (i.e., structured design versus object-oriented design). The design principles have stood the test of time and should be understood thoroughly by all practicing software engineers.

The design principles can be partitioned into two basic groups: those relating to individual module or object design and those relating to software architecture design.

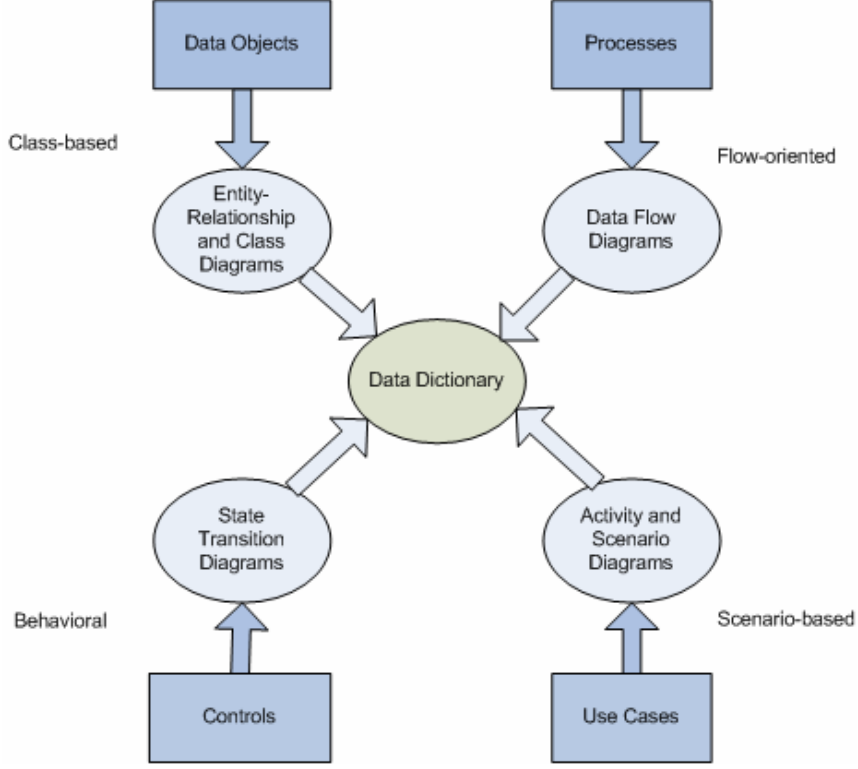
The overarching software design principle is simplicity. Complexity of software design should be avoided. A good rule of thumb is that the software design should not be more complex than the problem that it is intended to solve.

As previously described in module 1, the software engineering process is a critical part of software development. By conscientiously following a sound and proven process, mistakes can be minimized, in terms of both their number and their severity. This effect is known as **fault avoidance.** Developers who ignore the collective experience of industry and pursue what is called an *ad hoc* process (that is, driven by the spur of the moment) usually encounter problems later in the software development life cycle. Finding and removing defects in design and code is very expensive when done later in the life cycle.

1. Software Analysis Methodologies

In module 2 we demonstrated the elements (class-based, flow-oriented, behavioral, and scenario-based) that make up the analysis model. We represent these elements and their relationships in figure 3.1 below.

Figure 3.1
Elements of the Analysis Model



The analysis model is designed to achieve three primary objectives, which are outlined by Pressman (2005) as:

- to describe the customer requirements
- to establish the basis for software design
- to define a set of requirements that can be validated once the software is built

In this section, we further examine the processes that are used for software requirements analysis. In industry, these processes are called **methodologies.** As you will see, there are two major flavors: **structured methodologies** and **object-oriented methodologies.** These methodologies will provide the foundation for our discussion of design and the architecture to integrate the various components into a cohesive whole.

The elements of the analysis model as proposed by Pressman (2005) in figure 3.1 encompass all the features for both the structured and object-oriented methodologies. However, the software engineering team must choose which methodology is the best fit for the project and then exclude the elements in the analysis model that are not applicable to that methodology. The software engineer selects a methodology that best models the software requirements to ease the transition from analysis to design. In the following sections we will demonstrate the various elements presented in the analysis model and show how they can be used in each of the two methodologies.

Structured Analysis

The structured methodology for software requirements analysis has been practiced in the industry for many years. As such, this methodology is the most entrenched and the best known. **Structured analysis** was popularized by Tom DeMarco in 1979 in his book entitled *Structured Analysis and System Specification.*

With structured analysis, the functional behavior of the software is modeled using a set of standard notations. There are three key aspects to functional modeling: **data modeling, process modeling,** and **control modeling.** In the next three subsections we will discuss the functional modeling techniques.

Data Modeling

The purpose of data modeling is to construct a representation of the primary data objects that are relevant to the system. As pointed out by Pressman (2005), there are three related parts to the data model:

1. The **data object**—specific items (or entities) of interest to the system that are uniquely identifiable
2. Its **attributes**—properties of the data objects, in one of three forms:
    1. Name of the data object—how it is referred to
    2. Description of the object—descriptive characteristics
    3. Reference to other objects—specific instances of relationships between objects
3. The **relationships**between different data objects—how different objects are related in the context of the system's required behavior. There are three critical parts to any relationship:

1. The name of the relationship—what it is to be called
2. Its cardinality—the number of occurrences of the object as related to the number of occurrences of the other objects in the relationship
3. Its modality—whether the relationship is optional or not (that is, whether the objects need to be related)

When modeling data, notations like the entity-relationship diagram can be used. In particular, be sure you understand the differences between the objects and their attributes. The concepts of the reference attribute and the data model relationship can be a bit confusing. Just remember that the reference attribute provides the specifics about the relationship.

Consider table 3.1, which shows the attributes for the three types of cards that can be used for the payment transaction in our self-service checkout system.

Table 3.1
Table of Attributes for Cards

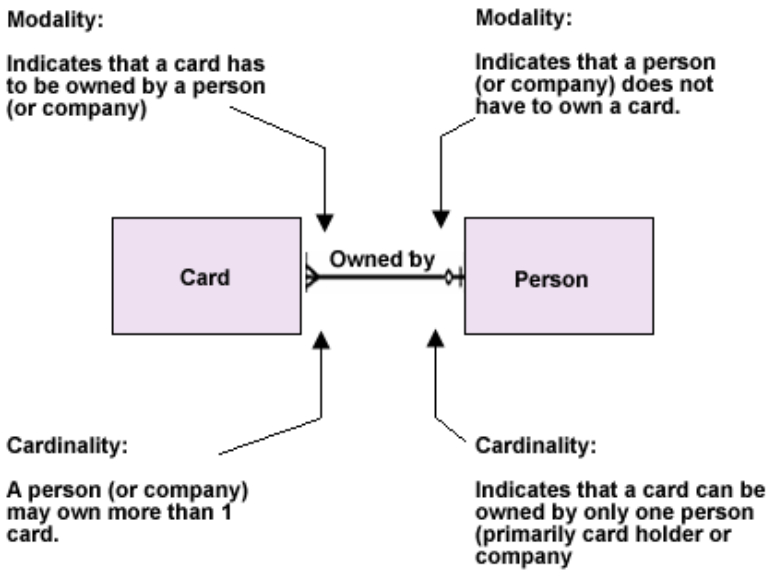
Type	Name	ID#	Code	Owner's Name
Debit	Visa	2354 4567 2476 9937	4532	John Thompson
Credit	Mastercard	5467 8945 9382 1293	234	Albert Smith
Shopper	Discount	2139847		Maggie Martin

The three types of attributes defined by the data model are:

1. The naming attributes: constituted by *name, ID#* (where *ID#* is the identifier)
2. descriptive attributes: constituted by *type and code*
3. referential attribute: constituted by *owner's name*

The referential attributes place a specific object ID in the place of the general relationship *owned by,* which is indicated in the ERD shown in figure 3.2.

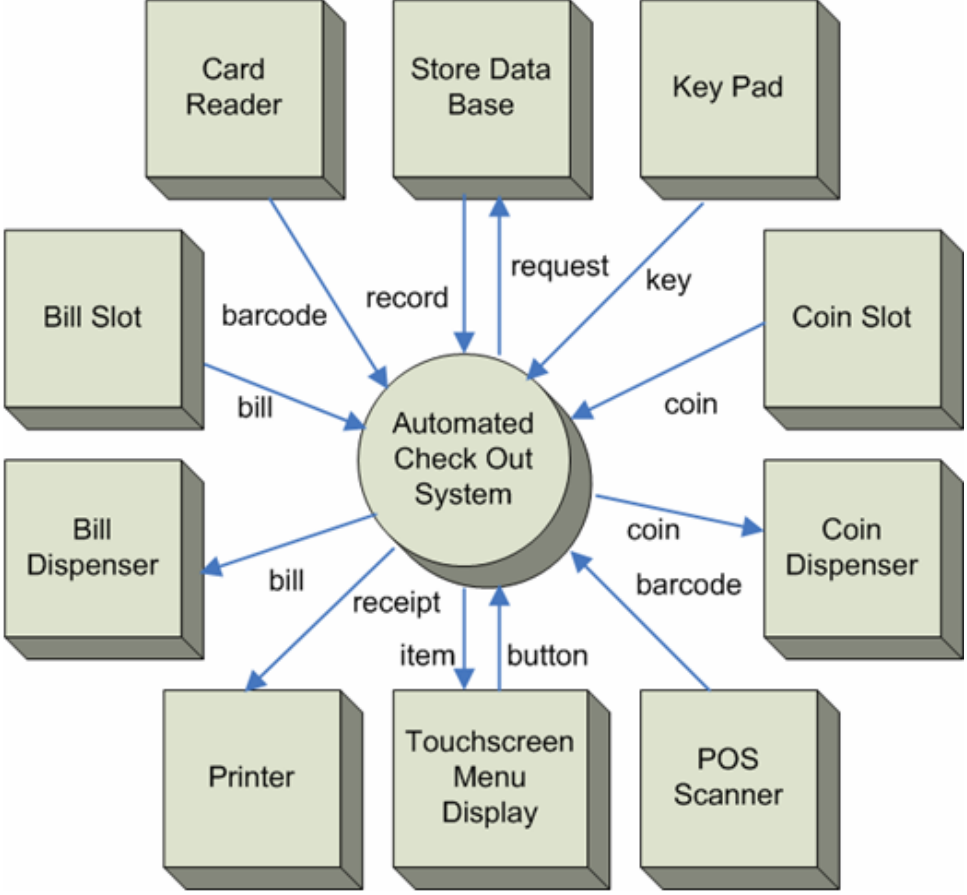
Figure 3.2
Relationship of ERD to Attributes



Process Modeling

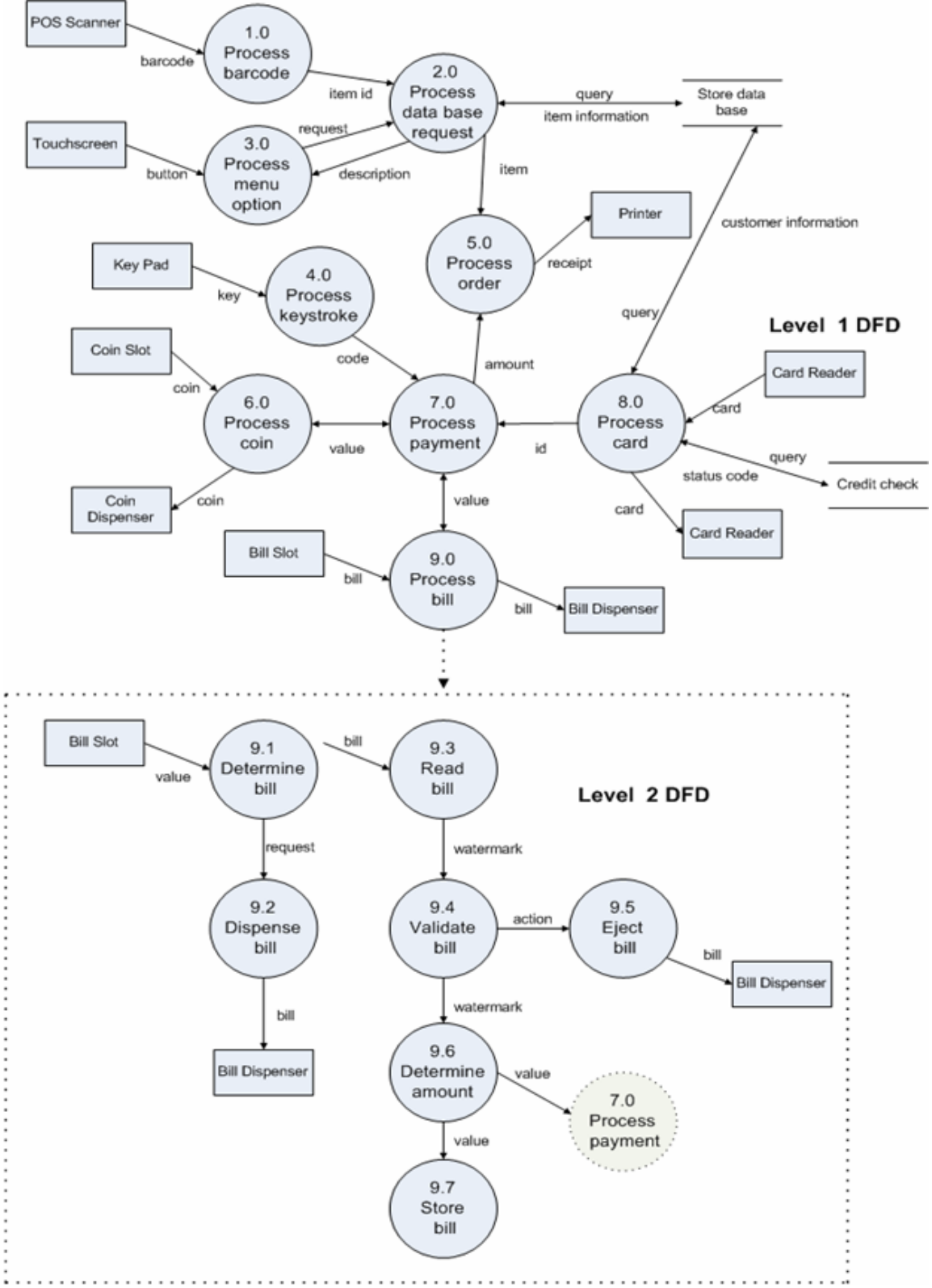
Process modeling is generally performed using data flow diagrams. A **data flow diagram** (DFD) is a graphical notation that depicts how data moves through a system. A DFD describes the various elements that, when combined, form these diagrams. Note that these diagrams may be drawn at different levels of abstraction. At the highest level, they are called context models or Level 0 DFDs. Notice how the system is represented as a single process in the Level 0 DFD represented in figure 3.3.

Figure 3.3
Level 0 DFD or Context Diagram for Self-Service Checkout System



As the refinement progresses, each process is expanded into "lower-level" diagrams, each showing more and more detail about the processes. Study figure 3.4, which illustrates an information flow model and basic DFD notation for the next two levels of the system process.

Figure 3.4
Level 1 and Level 2 DFDs for Self-Service Checkout System



The level 1 DFD depicts the major processes to implement the system and the level 2 DFD depicts all the processes corresponding to the refinement of a major process. However, depending on the function to be implemented in the process, not all processes may need refinement to a lower level.

When drawing DFDs, you should adhere to the following guidelines:

- Each process should have at least one input and one output data flow.
- Each process should transform the input data flow into a new output data flow.
- Each data store should have at least one data flow.
- Each external entity should have at least one data flow.
- Each process should be identifiable with a unique number at each level.
- Each data flow should be associated with at least one process.

As we move from analysis to design, we are creating a more detailed view of what is required by the software. It may look as if this process pushes us toward design, even though we are still in the requirements analysis phase. The key is that we are creating a model of

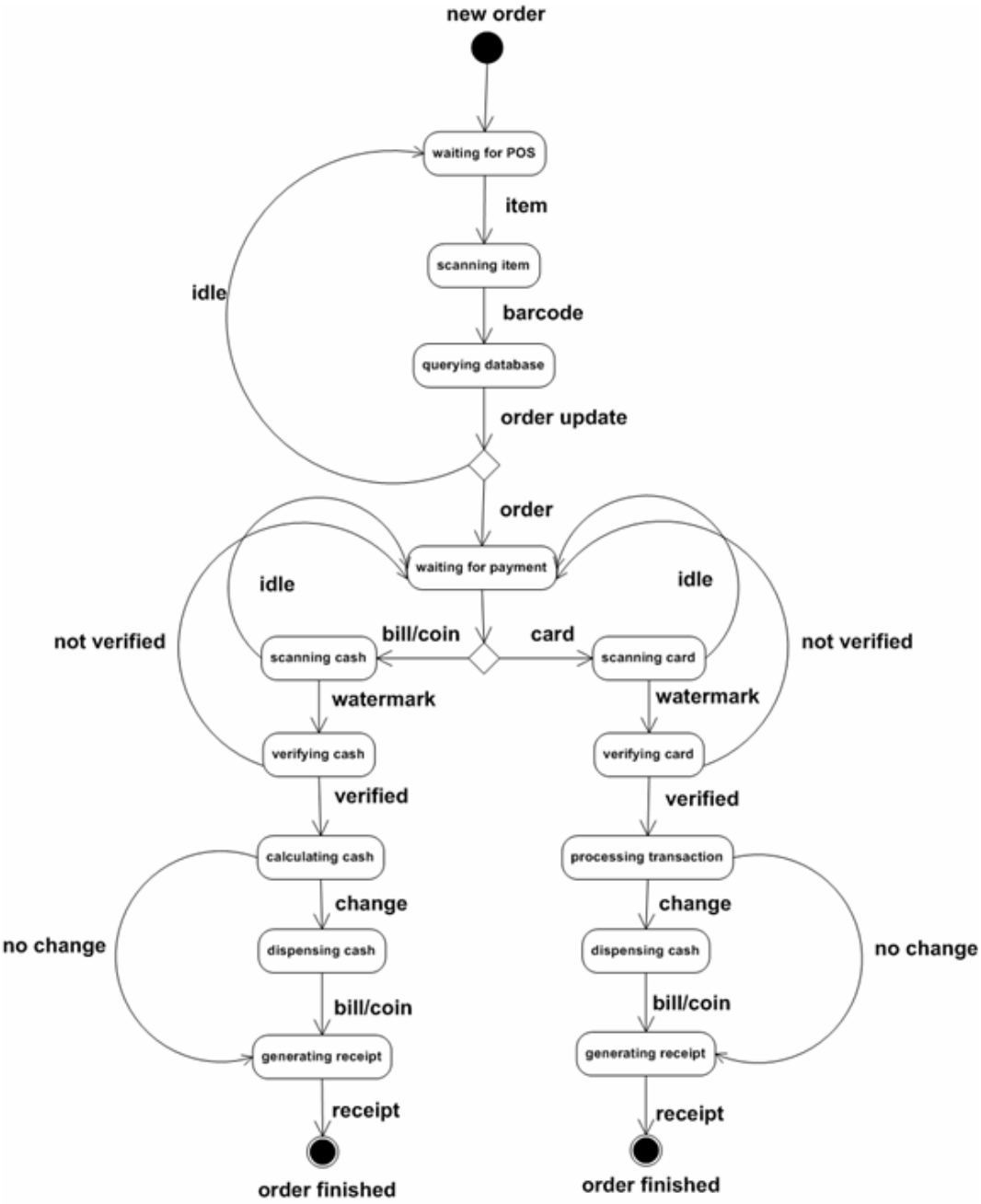
what is expected in the software to accomplish the functions at the modular level. The refinement approach provides a smooth transition from analysis to design—when we have completed defining our model we can proceed into design and use the DFDs to map the processes into software architecture.

Control Modeling

Control models are prepared using several notations, including state-transition diagrams (STDs), control flow diagrams, and control specifications (CSPECS). State-transition diagrams provide a different way of describing the behavior of the system. With an STD we can describe the sequencing of behavior. Every piece of software has a state, that is, a mode of behavior that can be observed. The software behaves differently for given inputs for each state. With an STD we capture these modes and specify, for each input and for each potential state, how the system transitions from one state to another and what it does with each transition.

Carefully examine the STD for the self-service checkout system shown in figure 3.5 and then read the description following the STD.

Figure 3.5
State Transition Diagram for Self-Service Checkout System



Description of Self-Service Checkout STD

In figure 3.5, each **state** is represented by a rounded box; a black circle represents an initial or final state. Each **state transition** is represented by an arc or straight line that exits from a box and proceeds to another box. Refer to the following explanation on notations used in state transition diagrams:

- **event:** specifies a particular input that will cause some action to take place (e.g., "item")
- **action:** occurs when the event takes place (e.g., "order update")
- **guard:** boolean expression that, if true, will enable an event to trigger a state transition (e.g., "verified")

The self-service checkout system that is described has two major states when a new order is initiated. Each major state has several minor states associated with it as described below:

1. **Waiting for point-of-sale:** In this state, the self-service checkout system is waiting for a user to scan an item for purchase. There is only one transition from this state. If the system detects an item over the scanner, a trigger will be issued that indicates the item is present. The transition to the next state will be invoked by the event to scan the item and read the barcode. The diagram omits if there is a system error detected when the item has been placed on the scanner. It also omits what happens if the barcode is missing from the item. Once the barcode is detected, a transition to the next state will occur, "querying database." Once the database query is detected, an "order update" trigger is generated. The system will return to the "waiting for POS" state until a signal is detected to transition to the next major state or a specified time limit has passed. When all order updates are complete by detecting the "order" trigger, the system will transition to the next major state, "waiting for payment."
2. **Waiting for payment:** In this state, the self-service checkout system is waiting for the user to either:
    - enter coins into the coin slot device
    - enter bills into the bill slot device
    - enter a credit card into the card reader

Once the transitioning of states for the scanning and verifying of all coins, bills, or credit cards is complete, the next state ("calculating cash" or "processing transaction") is entered to determine if any bills or coins need to be dispensed ("dispensing cash"). The final trigger is generated to transition to the "generating receipt" and "finish the order" states.

As you can see, by creating a model like this, you can follow the logic of the state transitions to detect when the specified behavior doesn't make sense. At this stage you need to make the appropriate corrections and reevaluate the model.

Underlying all of these types of models is the data dictionary, which retains and organizes the information relating to the data objects that are to be processed by the system. The data dictionary is at the heart of the structural analysis model. The data dictionary stores the descriptions of all the elements of the model, including the data objects, as well as representations for the function and control elements.

There are many variations in how data dictionaries are formatted, the discussion of which goes beyond the scope of this course. However, there are a few essential elements of a data dictionary, as outlined by Pressman (2001). We have summarized them in the following list as:

- **Name** – descriptive name of the data or control item, the data store, or external entity
- **Alias** – any other name used to describe the data
- **Where and how it is used** – listing of all processes that use the data and how it is used in the process
- **Content description** – notation for representing the content
- **Supplementary information** – information describing the data types, preset values, restrictions, or limitations

A CASE tool can be used to construct a data dictionary and to reduce the human effort and manage the complexity. Such tools, however, are useful for more than simple management of the data objects. If a formalized notation is used to describe the data, the tool can perform automated analyses of the data to ensure consistency across the system. Such analyses would be quite difficult if developers had to perform them by themselves.

Object-Oriented Analysis

The second major methodology for analysis is the object-oriented approach. Although you will see many similarities between object-oriented (OO) and structured analysis, you will also notice many differences.

In the OO approach, systems are viewed as collections of interacting objects, where objects are entities that combine data structures and operations. In the real world, most objects we encounter have structure and have a certain limited number of operations that we can perform on them. For example, consider automobiles and television sets; not only do they have different structures, we also use them differently. When following the OO approach, this phenomenon is explicitly recognized and modeled. In addition, although structured analysis tends to be developed in a linear fashion (requirements-design-implementation-test), OO developments frequently follow an incremental approach, where a series of software builds are released, each with more capabilities than the previous build.

There are some important concepts and terms that we deal with in OO. Study the following terms and definitions.

- **Objects**—items that have identity, such as a table, a car, or a TV set. Objects have two essential parts:
    - **Attributes**—the features that typify the object and that make it unique to the system, e.g., the *color* of a car.

    - **Operations**—the things that can be done to the objects, e.g., you can *drive* a car. By applying an operation to an object, one or more of the attributes are either referenced or modified.
- **Classes**—a collection of objects, all of which share common characteristics and operations, e.g., *motor vehicles* is a class. A class can be viewed as a template with which new objects can be built (instantiated).
- **Inheritance**—the passing of attributes and operations from the class to the object.
- **Encapsulation**—the isolation and management of all of the information relating to an object or class into a single concept, namely, the class itself.
- **Subclass**—a collection of specialized objects that belong to the superclass, e.g., *trucks* are a subclass of the superclass *motor vehicles.*
- **Messages**—objects communicate and interact via messages.
- **Polymorphism**—a characteristic of OO systems whereby an object can take on more than one form, a different behavior, depending on the data types that are passed to the object as arguments. The choice of which operation (from which class) to apply to the argument is made based on the class to which the object belongs.
- **Abstraction**—the act of representing essential features without including the background details or explanations.

As with structured analysis, the software engineer develops models for the proposed system. In structured analysis, we stressed the development of functional, data, and control models; however, in OO analysis we focus on the object and class models. We provided an example of a class diagram in module 2, figure 2.8, and we will provide a more detailed class diagram when we discuss object-oriented design in the next section.

When applying these processes, all of the individual elements of each of the models need to be identified (i.e., the objects, the classes, the inheritance relationships, and their attributes and operations). As Pressman (2005) points out, a variety of OO analysis methodologies can be followed, including the Booch method, the Coad and Yourdon Method, the Jacobson Method, the Rumbaugh Method (also known as the Object Modeling Technique, or OMT), and the Wirfs-Brock Method. Jacobson and Rumbaugh joined the Rational Corporation and began to develop a successor to OMT called Unified Modeling Language (UML). UML is the most commonly used notation for OO modeling in the industry today; we will focus on UML diagrams in our discussion.

Some of the traditional techniques require the use of functional models, also known as data flow diagrams and state transition diagrams. By their presence, we can see that OO is really an evolution of the functional techniques that we examined in the first part of this section, and has built on their foundation.

There are generally five steps in developing OO analysis models, several of which we already discussed in module 2. These steps (Pressman, 2005) are performed interactively until the analysis model has been completed:

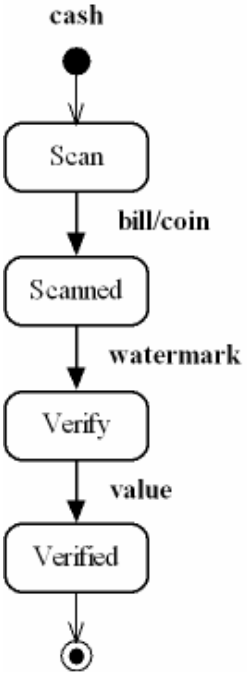
- Identify the basic user requirements and develop a set of use cases that capture the potential scenarios that explain how the users will use the system.
- Identify the classes needed for the application domain and specify the attributes and operations for each class.
- Specify a class hierarchy by identifying the superclasses and subclasses and to determine the levels of inheritance that exist between related classes.
- Determine the relationships that exist between the objects.
- Model the behavior of the objects.

When the system has been modeled, the complete behavior of the system should be unambiguously defined. The power of the OO techniques is that the OO models are relatively easy to understand and form a solid basis from which the software design can proceed.

State Transition Diagram Revisited

In our discussion of structured analysis, we introduced the state transition diagram as a means to model the overall behavior of the software. We demonstrated how to create a representation of the states and the events that trigger the transitions between all the possible states. In OO analysis, the state transition diagram can be used to model the behavior of individual objects. However, the STD is not used for modeling the collaboration between the objects. For example, in figure 3.7 we can trace the states and the events that cause a state transition for the cash object that was identified for the self-service checkout system.

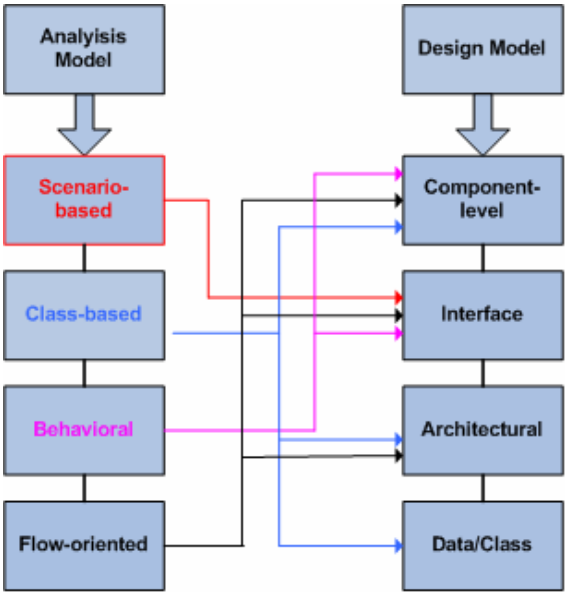
Figure 3.7
State Transition Diagram for Cash Object



2. Software Design Methodologies

Once the analysis phase is complete, the next task for the software engineer is to represent the software requirements by providing the details of the data structures, architecture, human-computer interface, and all the components necessary to implement a working system. The analysis model provides the foundation for the design model. The scenario-based, flow-oriented, class-based, and behavioral elements are translated into the data and class architecture, interface, and component-level designs that comprise the elements of the design model. The relationship of the analysis model to the design model (Pressman, 2005) is illustrated in figure 3.8. Study this figure until you understand the relationship.

Figure 3.8
Transition From Analysis Model to Design Model



As we mentioned in our earlier discussion, the design model encompasses all the features for both the structured and object-oriented methodologies. The software engineering team chooses which methodology is the best fit for the project and then excludes the elements in the design model that are not applicable to that methodology. In the following sections we will discuss the elements that apply to each design methodology.

Design Concepts

Throughout the history of software engineering, a set of fundamental design concepts has evolved and been used in the industry for development of improved design models. Pressman (2005) summarizes these design concepts in the following list:

- **Abstraction:** named collection of data that describes a data object

- **Architecture:** structure or organization of the components and data of a program, and the manner in which they interact
- **Patterns:** a design structure used to solve a particular problem within a specific context
- **Modularity:** separation of software logic into smaller, named components
- **Information hiding:** information such as the algorithm and data are hidden or inaccessible from other components
- **Functional independence:** each module can act independently of other modules and perform a function
- **Refinement**: levels of procedural detail can be achieved by stepwise refinement following a top-down strategy
- **Refactoring:** reorganizing the design without changing the function (Fowler, 2000)
- **Design classes:** develop classes according to the user interface, business domain, process, persistence, and the system

Structured Design

In conventional methodologies, design can be viewed as a transformation of the requirements analysis. There are four basic design activities that are conducted as a part of software engineering:

- **Data Design** — during which specific logical representations (data structures) for the data objects are prepared
- **Architectural Design** — during which the overall program structure is developed and documented
- **Procedural Design** — during which the specific and detailed processing steps are defined
- **Interface Design** — during which the program interfaces are defined

In the following subsections we will provide a brief discussion on each of these designs. Please note that we have limited the scope of our discussion because each design could entail a module of its own.

Data Design

Data design involves:

- defining specific logical representations for the data objects that were identified during requirements analysis
- identifying the program elements (modules) that need to reference or operate on each data element so that the scope of the data can be determined

The foundation of software design is the data. The software engineer designs data structures to best fit the representation of the data that was modeled during analysis. Data structures are designed at the software component level and database architectures are designed at the application level. The software engineer must design the data structures and databases to allow for efficient and reliable processing of data. To assist the software engineer in data design, Wasserman (1980) defines seven principles to follow for an efficient data design:

1. Consider the data design equally important as the functional design.
2. Determine the operations that will be performed on each data item.
3. Use the data dictionary developed during analysis as the foundation for data design.
4. Save the lower-level design details until later in the process.
5. Limit the details of the data structures on a "need to know" level in the modules.
6. Create libraries of data structures, and reference and use them when possible for new software.
7. Select and use an appropriate programming language to represent the data structures.

Architectural Design

In architectural design, the overall program structure is defined. This definition incorporates the data structures into the control structures. Pressman (2005) outlines several steps to guide the software engineer in the architectural design process:

1. Make a decision on which type of data flow is to be used based on the appearance (the flow boundaries) of the analysis data flow diagrams that were created during requirements analysis. Remember that many flow types are possible, but there are two important ones: **transform flow** and **transaction flow.**
2. Map the data flow diagram into the program structure using **transform mapping** or **transaction mapping** and define the control hierarchy by using the **factoring technique.**
3. Refine the resulting design until the ultimate architecture design is achieved that best represents the program structure.

As you can tell, this is not an exact science. Different software engineers will end up with different results (designs) even if they follow the same process. These are really just guidelines that help the software engineer to organize information into a cohesive whole.

Procedural Design

In procedural design, we use a set of logical constructs to form the processing steps for the software. You should be familiar with the set of logical constructs from earlier programming classes you have taken. We have listed them here for reference:

- **Sequence:** a series of processing steps to implement an algorithm
- **Repetition:** a repetition of processing steps, looping
- **Condition:** a selection of processing based on some logical occurrence

Each construct has a predictable logical structure that is entered at the top and exited at the bottom. By using the set of predefined constructs, the reader can easily follow the procedural steps that are needed to implement the logic in the software. These "logical chunks" mark the procedural elements of a module (Pressman, 2001) in the structured approach.

Interface Design

In interface design, we define the interfaces necessary to implement the software. In structured methodology, the three types of interfaces that need to be considered are:

- between software modules within the system
- between the software and external entities that interact with the system
- between the system and human

We will discuss design principles of human-computer interaction in great detail later in this module.

Object-Oriented Design

In requirements analysis we applied processes following the object-oriented approach. Now we can extend these processes in the design phase. Such a relationship is to be expected because OO development, typically, is iterative and involves many cycles around the requirements-design-code-test loop. In OO design, the analysis models that were created are evolved into design models.

During OO design, the various elements of the analysis model are converted into specific design elements by performing the following steps (Pressman, 2005):

- Identify any concurrency that exists in the system.
- Identify subsystems within the analysis model.
- Assign the subsystems to physical processors (computers) and software tasks.
- Decide on the approach to be taken for data management.
- Determine how global resources will be accessed.
- Determine how control of the system will be exercised.
- Identify boundary conditions and how they need to be handled.
- Resolve trade-off issues.

After reviewing these eight steps, consider how these steps differ from those described for structured analysis and design. Notice how, as the models are refined into designs, the progression of detail proceeds smoothly, showing that the various levels of models are compatible.

Classes and Objects Revisited

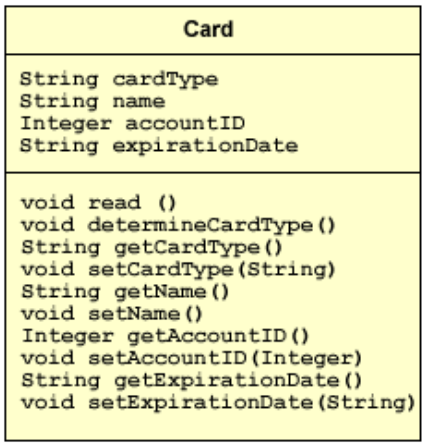
In OO design the software engineer refines the classes and objects that were defined in the class diagrams during the analysis phase. It may be necessary for the software engineer to add more detail to the classes and create new classes to provide the desired functionality. Once all the classes are defined, the relationship between the classes is modeled to show the hierarchy structure and to identify the superclasses and subclasses.

Identifying the set of classes for instantiating all the objects needed in a software design is not an easy task. Considerable research has been conducted on how to identify object classes. One researcher, Sommerville (2004), has summarized the best practices for identifying object classes as:

- Perform an analysis of the grammatical content of the textual descriptions (problem statements, use cases) for the proposed system. Identify the nouns in the descriptions that are good candidates for objects and attributes that can be used in defining classes.
- Identify the tangible entities and any data storage structure needed in the application domain.
- Review the behavioral diagrams that were created during analysis and identify any participant who plays an important role in initiating or participating in the behavior.
- Review the scenario diagrams created during analysis to identify data objects, attributes, and participants.

When designing a class, the software engineer needs to specify the interface, the signature, and semantics of the services provided by the object or groups of objects. These descriptions can be included in the class diagram used to model the design, as shown in figure 3.9.

Figure 3.9
Example Class Design

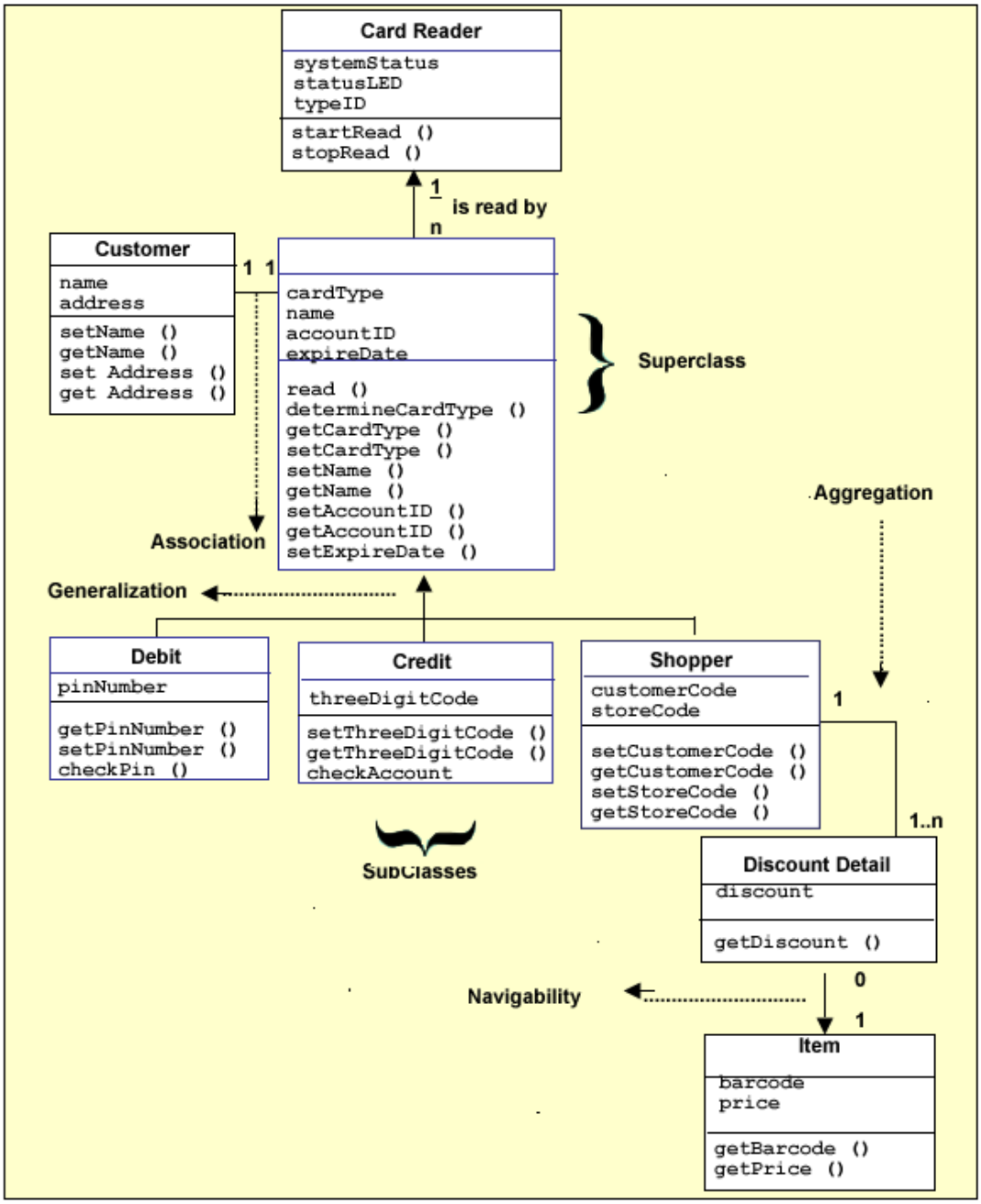


There are three types of relationships that can be modeled using a class diagram; we have listed them below:

- **Association:** depicts a necessary relationship with another class in order to perform its work
- **Aggregation:** links a class with a collection
- **Generalization:** links inheritance of a class with its superclass

In figure 3.10, we have extended the class diagram presented in module 2 for the self-service checkout system to show the possible relationships that can exist between classes.

Figure 3.10



Additionally, as seen in figure 3.10, associations can have a **navigability arrow** to indicate the direction and the ownership of the association. For example, a frequent shopper card can be associated with many discount detail objects. However, a discount detail can be queried for an item, but an item cannot query for a discount detail. In module 2 we discussed multiplicity of an association that shows the possible instances of the classes.

UML Design Diagrams

In OO design, there are different types of models that can be used in the design, **static models** and **dynamic models.** UML provides a variety of static and dynamic models that provide different diagrams for representing these models. We have already discussed a few of the more commonly used diagrams in previous discussions. The class diagram is an example of a static model and the state transition diagram (models software behavior, not object classes) is an example of a dynamic model. Additional diagrams that can be used for OO modeling are: the sequence, activity, component, communication, and deployment diagrams. You are encouraged to read more on each of these diagram types by performing a search on the Internet.

Programming Languages

One important consideration in design is the language chosen for implementation of OO concepts. Unless the programming language provides facilities for implementing OO concepts, the resulting code may be forced to make use of procedural constructs rather than OO structures. Care must be taken to ensure that the OO models are faithfully preserved so as to avoid erroneous functionality and inefficiencies.

Component-level Design

Once the data and architectural designs have been completed by the software engineer, the components of the software are delineated. You can think of components simply as building blocks that make up the operational pieces of the software. Let's take a look at the formal definition provided by the Object Management Group (OMG), which defines a component as a "modular unit with a well-defined interface that is replaceable in its environment" (2005, pp. 155-158). How does this definition apply to the methodology we choose to follow for analysis and design?

In structured methodology we view a component as the smallest program element, a module. A module is defined as an independent piece of code that implements processing logic and contains internal data structures. Each module has an interface, a signature, which enables the module to accept data passed to it from other modules (Pressman, 2005).

In object-oriented methodology we consider a component as a set of classes related to the problem domain (Pressman, 2005). The objects that are instantiated from the classes collaborate with each other through message passing. Each class is fully defined with a set of attributes, operations, and the necessary interfaces to pass messages between the collaborating objects.

Designing for Quality

As we discussed in module 1, high quality in software is a measurable and desirable characteristic for companies to remain marketable in the industry. Software quality can be measured and evaluated later in the development cycle by assessing a set of attributes that can be traced back to the design. To ensure that quality is built into the design, the software engineer should consider the attributes that satisfy the functionality, usability, reliability, performance, and supportability (FURPS) requirements during the design phase. We briefly discussed the FURPS requirements in module 2; now you should refer to table 3.2 for a list of attributes (Pressman, 2005) that need to be considered during design to satisfy each of the FURPS requirements.

Table 3.2
FURPS Quality Design Attributes

Requirement	Quality Attributes
Functionality	Feature set and capabilities, security
Usability	Human factor considerations, aesthetics, consistency, documentation
Reliability	Frequency and severity of failure, output results accuracy, mean-time-to-failure (MTTF), and failure recovery time
Performance	Processing speed, response time, resource consumption, throughput, and efficiency
Supportability	Ease of maintenance, testing, compatibility, configurability, and installation

To evaluate the quality of a design, Pressman (2005) has provided a list of technical criteria to access a software design. We have summarized these guidelines for achieving quality in design, if the design depicts:

- a design architecture style that is recognizable
- a logical partitioning of elements and subsystems
- a distinct representation of data, architecture, interfaces, and components
- data structures that are used as the foundation for classes and are evolved from noticeable data patterns
- independent components that convey functional characteristics
- simplicity in the interfaces to connect components with the external environment
- a representation using a notation that effectively communicates its meaning

In module 4 we will discuss how to verify quality in design. In module 5 we will discuss the role of project management in ensuring quality in the development of a software product.

IEEE Recommended Practice for Software Design Descriptions

The Institute of Electrical and Electronics Engineers has provided some guidelines for writing a Software Design Description (SDD) in the IEEE Recommended Practice for Software Design Descriptions, IEEE Std 1016-1998.

The SDD is referred to as a *build-to* specification and is the definitive statement on the implementation of the software components of the system.

In modules 1, 2, and 3, we have discussed several diagrams that can be used to graphically represent software design following either the structured or object-oriented methodology. These diagrams can be included in the SDD. The use of diagrams is highly recommended in technical documents. All diagrams should be supplemented with a textual description.

3. Application Domain Issues

In this section we bring together several computing-system topics that require the software engineer's special consideration. The topics are:

- client-server software engineering
- real-time system design
- human-computer interaction (HCI)

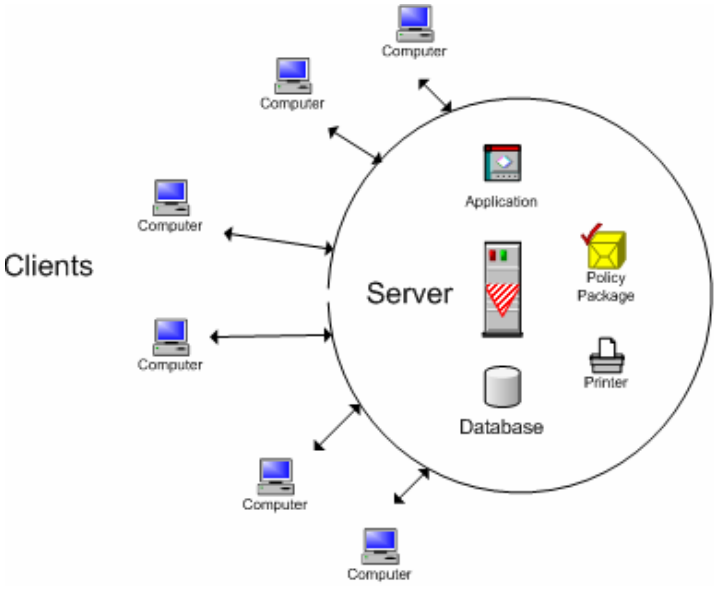
These topics need special consideration because they are tightly bound to the environment within which they operate. Although you should make sure you understand the operational environment of all computer systems, recognize that the computing systems listed have unique characteristics that must be addressed by the software engineer.

The concept of **application domain** is critical to the practice of software engineering; using this concept helps us to group or categorize computing systems by the type of problem each solves. By grouping computing systems into general categories, the software engineer can reuse former effective problem-solving approaches to solve future problems that are in the same category or have similar characteristics. Software reuse will permit more efficient software development by exploiting work done in the past, thus freeing the software engineer to focus on the unique aspects of the automated data processing problem.

Client-Server Software Engineering

The topic of client-server systems is really a subset of the larger topic of distributed systems. A client-server system is a specific distributed system architecture in which components of the system are assigned specific roles, i.e., a client role and a server role. Therefore, to implement client-server systems, you need to develop two software components, the client and the server. We have modeled the client-server architecture in figure 3.11.

Figure 3.11
Client-server Architecture



In addition, these role designations are "visible" to the logic of the software that implements the client and the server. In other words, the roles are hard-coded into the software application program. Hard-coding information into application programs is not bad in and of itself, but it results in software that is inflexible and difficult to change. Software should be designed to be abstract and modular with high **cohesion** and low **coupling.** In other words, the module or class has a focused responsibility with minimal collaboration with other modules or classes. These desirable attributes of design result in flexible code that minimizes the amount of hard-coded information that binds it to a particular computing environment or software architecture. But in client-sever systems, hard coding cannot be avoided and must be included in the design.

The software engineer must also make design decisions about the distributed system components of client-server systems. These components, which must all communicate with each other to perform their tasks, are the building blocks of client-server systems. They include the physical network cabling, the network protocol, the application communication protocol, the application program, and the integrating arbitrator or broker of network resources.

In general, the purpose of client-server systems and distributed systems is to share computing resources. Ideally, to the individual users of the client-server system, these shared resources should appear as if they are local, i.e., the computing resources should appear to be resident or owned by the workstation with which the user is interacting.

A common misconception is that the term *computing resources* strictly means hardware, e.g., computer, printer, modem, scanner, and so on. Shareable computing resources, however, not only include the hardware components, but also the application programs, the data used by the application programs, and the graphic user interface.

Many experts argue that the data component is the most significant "design driver" for the client-server system. Data, in reality, *is* the fundamental asset of any organization. The questions that the software engineer must answer before final design decisions are made about the client-server system are:

- Who uses the data?
- Is the data specific to a particular group or geographic area?
- How critical is data integrity and data security?

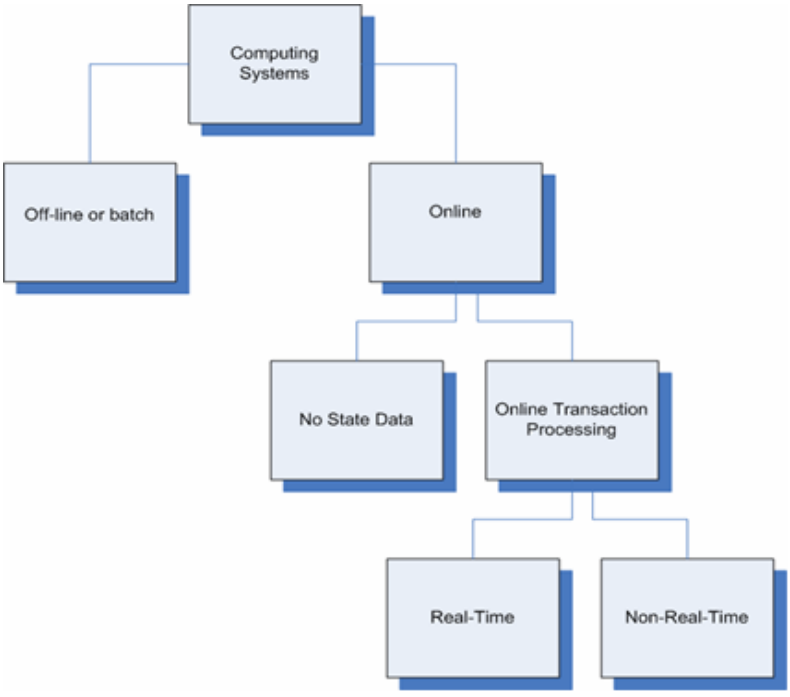
Ideally, the software engineer would strive to locate the data repositories as close physically to the user as possible. This ideal, however, frequently must be traded off against such factors as the need for data integrity and security.

Each time a client-server system is to be developed, the software engineer must design the client and server application components, the network, and communication protocols and middleware, and the computing platforms upon which the client and server applications will execute. The software engineer must also determine the location of the data and whether it is centralized, replicated, distributed, or staged.

Real-Time Systems Design

One way to understand real-time systems is to look at a high-level categorization or family tree of computing systems. In figure 3.12, we represent a very simple family tree of computing systems.

Figure 3.12
Family Tree of Computing Systems



More detailed trees can be drawn with multiple interpretations, but the above diagram will suffice to illustrate the basic categories of computing systems that you will consider in this course.

One of the first observations we can make from looking at this family tree is that real-time systems are online transaction processing systems, but online transaction processing systems are not necessarily real-time systems.

How do real-time systems differ from online transaction processing systems? The answer involves two system attributes: response time and data accuracy. Response time is the time interval between an input into the system and the corresponding system output. Data accuracy is just what the words imply—the closeness of the data value to its true value. These two system attributes behave

proportionally—in other words, for greater data accuracy, the system must take more time to process the data, so response time is greater.

Real-time systems, however, are often required to have very short response times. Data accuracy, therefore, sometimes must be compromised. Typically, a real-time system may have a response time requirement of several seconds or fractions of one second, or even less. For example, an air traffic control system must update each aircraft's position and direction of flight within six seconds or less.

Note that real-time systems are not characterized only by short response times, but instead by a need to respond within very specific, predefined time budgets. As mentioned, these times can be short, but it is not necessary that they be so. The main constraint is that such systems have to meet their time budgets—or else. In fact, there are two varieties of real-time systems: hard real-time systems, which must meet their time budgets or they are considered to have failed, and soft real-time systems, which must meet their time budgets on the average, but can occasionally miss a few and still not be considered to have failed.

Online transaction processing systems that are not real-time systems are those in which data accuracy is a more important requirement than is a specific, often short response time. For example, an ATM must determine an accurate account balance. Data accuracy must be ensured at the expense of the ATM customer's time.

Analysis and Simulation of Real-Time Systems

One of the hardest tasks of a software engineer is to prove *before the system is actually built* that his or her real-time system design will satisfy its performance requirements. System development is too costly to risk building a real-time system that doesn't satisfy its performance requirements. But how can the system's performance be proven before the system is actually built?

The only way around this dilemma is to model the real-time system while it still exists on paper and to study the model's behavior in an environment that mimics the environment within which the system will ultimately operate. This activity is referred to as simulation and is an effective means of verifying during the design stage the compliance of the real-time system with its performance requirements.

All simulation models fall within two broad fundamental categories: time-series simulation and discrete-event simulation.

The **time-series simulation model** is used to analyze the real-time system in its environment by "walking the clock," i.e., moving ahead in small increments of time and studying a "snapshot" of the state of the real-time system at each increment. Inputs to the system are generated artificially, but made as realistic as possible so that the system behavior is as close as possible to what can be expected in the real environment.

For example, a simulation of a patient-monitoring system that tracks vital statistics of patients in post-operative care would take snapshots of all vital sign data of a pseudo-patient at one-second increments to determine whether critical data is processed correctly.

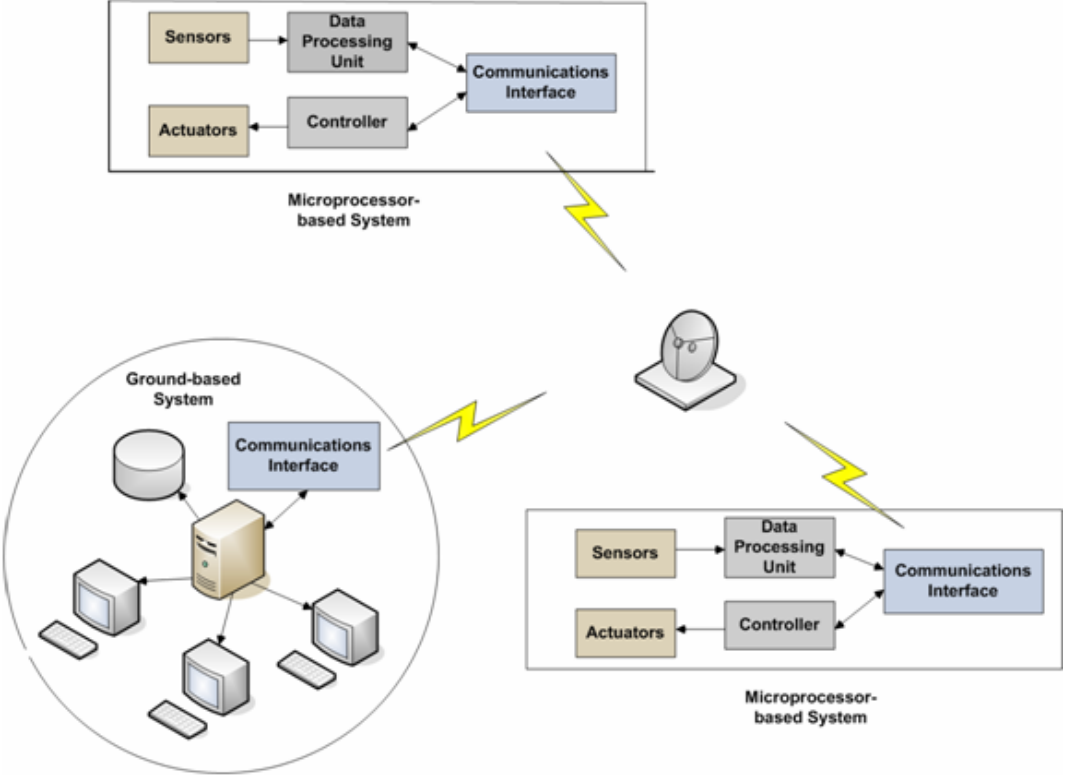
The **discrete-event simulation model** is used to analyze real-time systems by observing the state of the system after each input event. (An event is any discrete input to the system.) The inputs to the system are repeatedly generated artificially and are designed to arrive at rates that realistically parallel the real environment. For example, a simulation of an air traffic control system would study the state of the system immediately after the detection of an aircraft event.

Typically, the occurrence of events is random, like customers arriving at an ATM or a self-service checkout. The time interval between any two random events has been found to conform to certain mathematical models, e.g., an exponential distribution in statistical analysis.

Real-Time System Example

In figure 3.13 we have provided an example of a large scale real-time system, a telemetry control system that is widely used in the space, gas, oil, and electricity industries. These types of systems provide centralized control and monitoring of one or more remote sites from a single control room (Cooling, 2003). Each remote site contains a microprocessor-based system and communicates telemetry in real-time with a ground monitoring station. Our model depicts a real-time system with two remote sites and one ground-based station.

Figure 3.13
Real-Time Telemetry System



Human-Computer Interaction Design

In this section, we will discuss human-computer interaction (HCI) design issues, which are necessary for the effective operation of a software system.

HCI design requires a combination of software engineering and psychological skills. We have all already developed some of these HCI-related psychological skills simply because we are humans and we've interacted with computers. We have a feeling for what constitutes a good HCI design and what constitutes a bad one. Now we need to put our understanding of good HCI design into precise software engineering terminology.

Modeling the HCI permits the software engineer to characterize how various users view the system. No system has only a single type of user. At the very least, there are operational users, maintenance users, management users, and training users. There may be other types of users as well, depending upon the system being built. Each type of user has a unique view of the system. Commands, displays, and help facilities may be different for each type of user. It is the responsibility of the software engineer to define these user views and design them into the system.

To completely design an HCI for a user type, a task analysis must be performed. A task analysis itemizes all the activities that each type of user will perform with the system. Each and every activity or task is defined down to the level where all possible system responses are defined for each user-command entry into the system. A task analysis is a very detailed study. This level of detail is especially critical for systems that involve the safety of humans, such as air traffic control systems.

A prototype, i.e., a dynamic interactive model, is a handy tool for resolving HCI design issues. Problems and their potential solutions seem to rise to the surface when they are prototyped in a mock-up of the system; the subtle interdependencies of features and system characteristics become more apparent for the software engineer and the user to study.

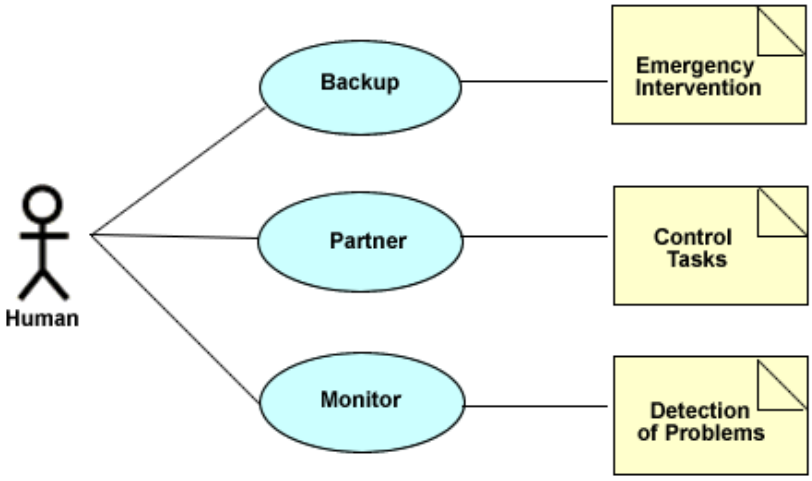
Interface Complexity

An example of a complex interface is the cockpit of a Boeing 777 aircraft, which contains many physical controls and displays. Obviously, highly trained humans and complex software are needed to operate this aircraft. Establishing the appropriate human-computer interaction design for the right interface for a complex system is instrumental for its safe operation.

Leveson (1995) explains that coping with the interface complexity requires humans to form an abstraction to allow them to concentrate on the relevant aspects of the problem.

Humans have the ability to form and change mental models depending on the role they are to play when interacting with a complex system. Figure 3.14 shows these three mental models and their functions.

Figure 3.14
Roles of the Human in Complex Systems



Source: Leveson, 1995

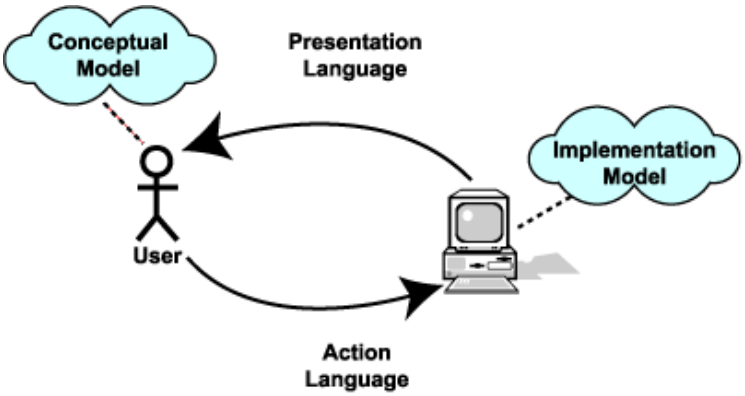
Design of the Human-Computer Interface

Designing the right interface for operating a system requires an understanding of the human and the basic principles of interface design. (The human element of a system is also called the *user* or *operator*.) In this section, we will discuss the user-interface model, the object-action model, the different types of users, and the ways to enhance the interface so that users can easily understand how to use it to perform specific tasks.

User-Interface Model

The definition of a user interface differs among programmers, analysts, human factor experts, and graphic designers. A model based on John Bennett's (1983) work describes the user interface using a holistic view. Refer to figure 3.15, which depicts the concepts of Bennett's user-interface model.

Figure 3.15
Model of the User Interface



The **conceptual model** represents the user's understanding of the task domain that is to be performed. (A task domain refers to the activities and the environment in which the task is to be performed.) This understanding includes how the system objects represent the real-world processes and the sequence of actions that must be taken to reach the intended goal.

The **presentation language** represents how the system visually displays the information to the user.

The **implementation model** represents the hardware and software configuration that is necessary to deliver the system's functional capabilities.

The **action language** is the mechanism that the designer has implemented to allow the user to interact with the system. The mechanism can consist of commands entered via the keyboard, mouse, or voice.

Interaction dynamics in the Bennett model are emphasized by the relationship of the computer to the human. The user initiates the interaction process with the computer. The goal is that each task is to be accomplished by using the application's interface. The user has a conceptual idea of the task domain and plans the sequence of actions that must be taken to reach the intended goal. The user forms a **scenario** to be carried out using the interface.

The actions in the scenario are transformed into steps that must be taken to successfully implement those actions through the use of the computer. The action language allows the user to execute these steps by initiating the functions in the software to perform the task. The software and hardware configuration permits the processing of each function and provides the mechanism for the user to provide the initiating input.

The presentation language provides the feedback necessary to show the results of the actions to the user. Feedback should provide the necessary information to guide the user on what to do next to complete the task.

Analyzing the Tasks To Be Accomplished

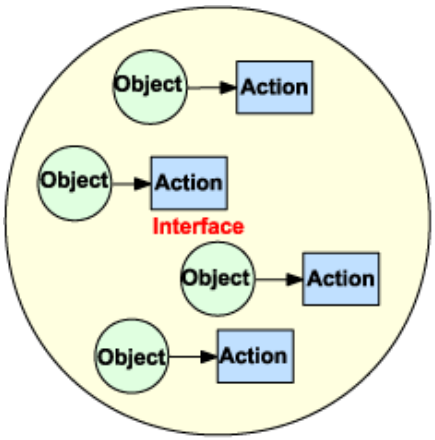
The first step in designing the user interface for any system is to identify the goals of the interface according to the requirements. One approach to designing the interface for complex systems is to create use cases and scenarios that mimic the actions of a potential user. We demonstrated how to write use cases and model scenarios in module 2.

To recap, user scenarios were introduced by Ivar Jacobson et al. (1992) as part of an object-oriented development methodology. Each scenario defines how the actor will interact with the system to achieve a specific goal by accomplishing a particular task. The software engineer focuses on essential use cases, which are stripped of implementation constraints and alternatives. As you saw in module 2, this approach enables the software requirements to be derived so the actor can achieve the objective or desired output in each scenario. The use cases modeled during analysis are used to direct the human-computer interface design model.

Object-Action Interaction Model

To fully understand how to design a user interface, Ben Shneiderman (1998) discusses the object-action interface (OAI) model in his textbook, *Designing the User Interface.* The OAI stresses the direct manipulation of an interface by humans using a customized set of objects and actions to carry out the usage scenarios. The basic OAI model is represented in figure 3.16 as a set of objects and actions that make up a complete human-computer interface.

Figure 3.16
Object-Action Interaction Model



The software engineer conceptualizes the real-world objects that can be manipulated to accomplish the application's intentions and the actions to be initiated by the operator. The software engineer defines these objects to optimize the operator's understanding in performing the tasks. The goal of OAI is to define a hierarchy of these objects and actions to mimic the tasks that are necessary to invoke the required software functions of the application.

Designing the interface following the OAI model requires the software engineer to have a thorough understanding of the specific tasks that must be accomplished by the software. The larger, high-level tasks are refined into smaller, lower-level tasks that reveal all the tasks necessary to fulfill the objectives of the interface.

The steps necessary to accomplish each task are written as computer-independent descriptions. Shneiderman (1998) suggests asking a series of questions to help identify these tasks.

- What information is needed before executing the task?
- How much of this information relates to the objects and actions needed on the interface?
- How much of this information is related to the system operations?

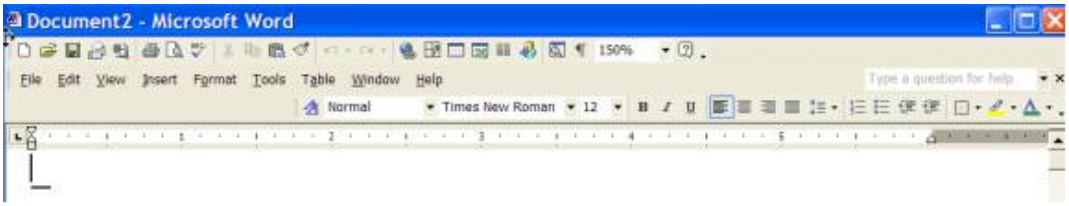
Shneiderman (1998) describes **interface objects** as visual representations using combinations of pixels on the screen that are strategically placed in the display space. In the next section we will examine the familiar Microsoft Word interface as an OAI example.

The OAI model can be used to represent items for complex processes that must occur for an operator to accomplish a task (Shneiderman, 1998). By establishing the relationship between the object and the action in similar applications, operators can easily be trained to comprehend the required tasks on complex interfaces.

OAI Example

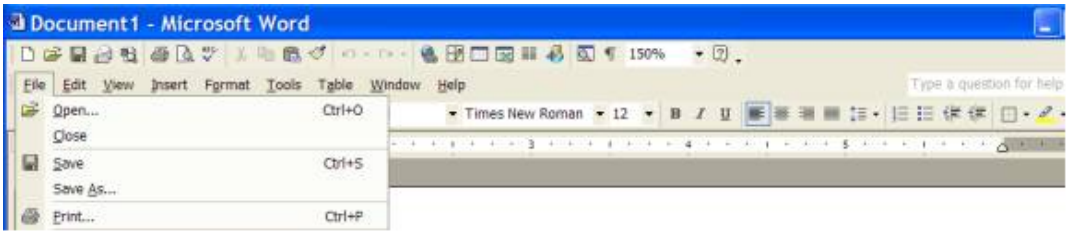
To demonstrate the OAI model, we can easily identify the objects in the Microsoft Word interface as shown in figure 3.17. The interface consists of a combination of menus, buttons, list boxes, and icons.

Figure 3.17
Microsoft Word User Interface



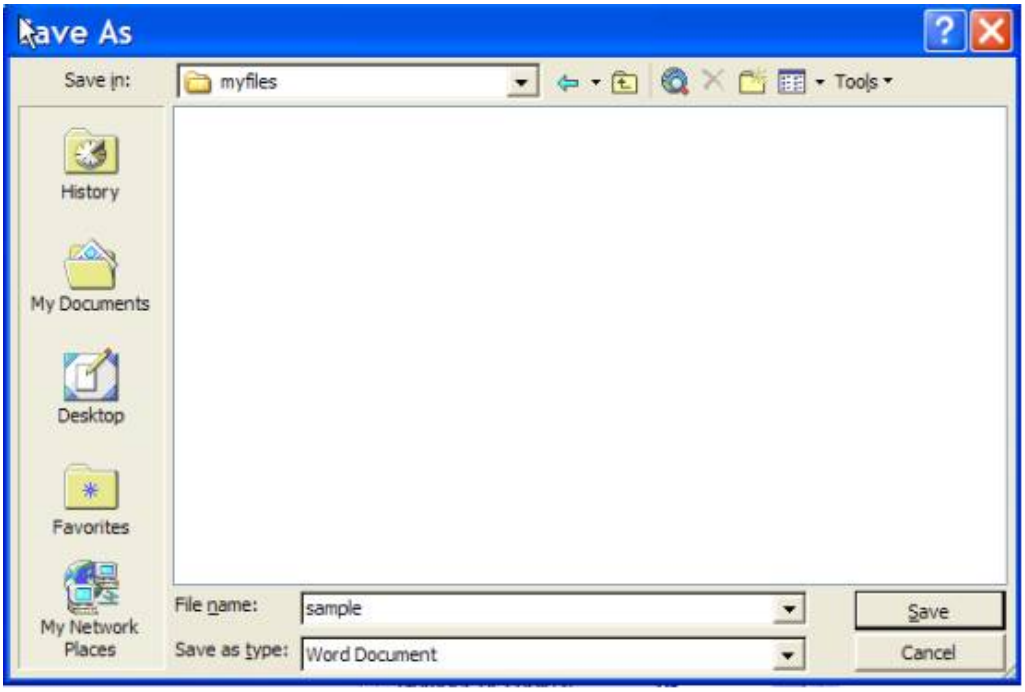
Now let's take a closer look at the Windows interface and identify some of the specific common objects that are used frequently in many applications. The menu bar at the top of the Windows application contains the familiar *File* item that brings up a submenu with a list of items that can be selected to invoke basic tasks. For example, let's review the *Save* and *Print* items. When you select these items, the expected behavior will occur in all applications that follow the Windows standards. Refer to figure 3.18 to see a snapshot of the *File* menu with its submenu and list of items.

Figure 3.18
File Menu and Submenu



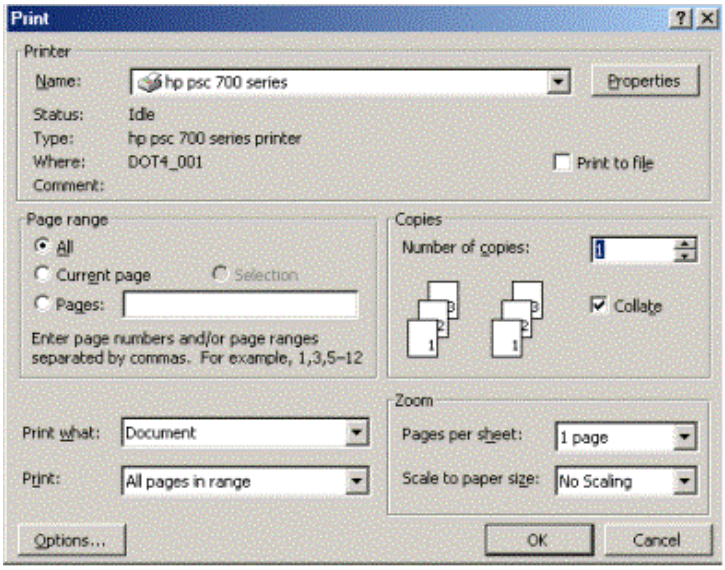
Selecting the *Save As* item from the *File* submenu will invoke the common file dialog box to be displayed. The file dialog prompts the user to select or specify a directory and file name. Refer to figure 3.19 to see a snapshot of the *Save As* dialog.

Figure 3.19
Microsoft *Save As* Dialog



In figure 3.18, selecting the *Print* item from the *File* submenu will initiate the generic *Printer* dialog, which will prompt you to select a printer device that is currently installed on your computer. Refer to figure 3.20 to see a snapshot of the *Print* dialog.

Figure 3.20
Microsoft *Print* Dialog



The **icon** is another form of an object that can represent certain actions. The meaning of an icon should be intuitive to the user. But software engineers must be careful when using icons because they are subject to different interpretations. The Windows interface uses the pictures of a 3-1/2" floppy disk and a printer device as icons to represent the *Save* and *Print* functions. These icons are used to direct the user to perform the same actions as represented in the *File* submenu.

In summary, as users, we have been trained to be familiar with the Windows interface by using the same objects with the same associated actions in many different applications. Users can perceive and act on these objects and then form a conceptual view of how they will behave together as a cohesive whole in a particular application.

Designing the Right Interface

Designing the right interface for the intended user requires some preliminary analysis. The basic types of users are categorized during analysis as being novice, intermittent, and expert. These categories take into consideration how frequently the user uses the system and his or her previous knowledge of the mechanics of the interface (Pressman, 2005; Shneiderman, 1998). Determining the user's level is instrumental in designing the objects.

Different approaches to interface design can be used for displaying the right amount of information and the type of features that are used in the design according to the intended user's category. For example, the novice user needs more directive information on the display, but in smaller amounts. The simple layered approach to control information flow and feedback would work well in the case of the novice user who uses the system infrequently. On the other side of the spectrum, the expert or frequent user may not need as much information to make informed decisions.

Usability studies have been conducted over the decades and suggest that the success of the technology used in the interface component for a complicated system is based on fundamental characteristics of the user, the tasks, and the environment.

Collins (1995) lists the characteristics of end users that must be considered in the design of the user interface:

- physiological
- sensory/motor
- cognitive
- cultural

Physiological, sensory/motor, and cognitive characteristics are a result of one's background and education and are beyond the control of technology. These characteristics are difficult to change but can be addressed by preliminary training and incorporating *help* features into the interface design.

The best approach for implementing a design of the user interface is to develop a prototype. A prototype allows the intended users to try out the system under the watchful eye of the designers, who can identify specific problem areas that are inherent to the characteristics of the users.

Basic Rules for Human-Computer Interface Design

Collins (1995), Leveson (1995), Pressman (2005), and Shneiderman (1998) have provided a comprehensive list of HCI design guidelines for systems. Some of these guidelines are highlighted in the following list.

- Create objects that represent the application domain.
- Sequence actions consistently.
- Organize the available display space.
- Provide directive and informative feedback for decision making.
- Produce meaningful error messages.
- Present and acquire information consistently.
- Prevent input error conditions.
- Use bold colors for highlighting critical information.
- Use sound to alert users in emergency situations.
- Guide the user through critical sequencing of actions.
- Display the appropriate amount of information necessary for decision making.
- Provide a means of reversal of actions in case of error.
- Reduce the short-term memory load.
- Place the user in control.

Final Words on Human-Computer Interface Design

More than any other characteristic, the user's proficiency in interacting with the system determines which set of guidelines the software engineer may choose in designing a specific system. To that end, the user's skill level determines the appropriate Human-Computer Interaction (HCI) design.

Using the air traffic control system as an example, we know that air traffic controllers do not need help screens and forms to assist them in command entry. There is simply no time to query the system about the proper format of a command. The controller is a highly skilled user of the system who requires rapid command-entry facilities and shortcuts (i.e., "fast-path" command entry) to support the time-critical nature of air traffic control.

References

Bennett, John. *Building decision support systems.* Reading, MA: Addison-Wesley, 1983.

Collins, Dave. *Designing object-oriented user interfaces.* Redwood City, CA: Benjamin/Cummings, 1995.

Cooling, Jim. Software Engineering Real-Time Systems. England: Addison-Wesley, 2003.

Fowler, M. et al. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.

Jacobson, Ivar; Magnus Christerson; Patrik Jonsson; and Gunnar Overgaard. *Object-oriented software engineering: A use case driven approach.* New York: Addison-Wesley Professional, 1992.

Leveson, Nancy. *Safeware: System safety and computers.* New York: Addison-Wesley, 1995.

Object Management Group. Unified Modeling Language: Superstructure, Version 2.0, 2005. Retrieved on September 5, 2006 from http://omg.org.

Pressman, Roger S. *Software engineering: A practitioner's approach* (fifth edition). New York: McGraw-Hill, 2001.

Pressman, Roger S. *Software engineering: A practitioner's approach* (sixth edition). New York: McGraw-Hill, 2005.

Shneiderman, Ben. *Designing the user interface: Strategies for effective human-computer interaction* (third edition). Reading, MA: Addison-Wesley Longman, 1998.

Sommerville, Ian. Software engineering (seventh edition). England: Pearson Education Addison Wesley, 2004.

Wasserman, A. Software Design techniques, third edition. IEEE: IEEE Computer Society Press, pp. 287-293, 1980.