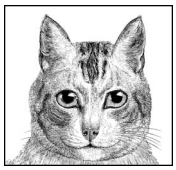

CHAPTER 3

Object Orientation

This chapter presents the object-oriented paradigm that forms the basis for the UML. After reading this chapter, you will understand the following:

- Real-world concepts; this is where problems are conceptualized.
- Implementation-world concepts; this is where solutions are realized.
- The function-driven, data-driven, and object-oriented paradigms and how they relate to one another.
- Fundamental object-oriented paradigm concepts.

This chapter introduces some of the key concepts and constructs of object orientation. Some of these concepts may be new to you if you have only become familiar with object orientation through an implementation language such as C++ or Java. I start by introducing the object-oriented view of the real world, and the fundamental concepts of objects and classes. I then introduce links and associations, which represent relationships among objects. I also introduce scenarios and interactions, which determine how objects interact and make use of one another.

Worlds

The world we all live in is impenetrably complex; to solve our problems we must reduce the important parts of each problem to elements we can manipulate. Our representational constructs are mental notions or ideas that represent something within a particular world or domain. Fundamentally, anything we are able to think about is a representational construct. Our way of simplifying the world so we can create and manipulate representational constructs is called a paradigm; in this chapter we examine the object-oriented paradigm and two others.

Solving a problem involves manipulating representational constructs from the problem domain and the solution domain to derive a representation of the desired solution. Realizing a solution involves mapping those representational constructs of the solution onto the solution world, that is, constructing the solution. The use

of representational constructs is a very natural process that often occurs subtly and sometimes unconsciously in problem solving.

A paradigm determines the possible types of representations utilized in problem-solving efforts. The actual representations are determined by the problem being solved and the solution being derived. A language allows us to express these representations. Once expressed, they can be communicated to others so that we can solve problems within teams. A paradigm ought to consist of all constructs required to address a diverse multitude of problems and to derive a solution that may be realized using a diverse multitude of mechanisms.

Consider the example in Chapter 2, in which an organization desires an information system to better utilize employees on projects. In solving the problem, representational constructs from the problem domain must be identified and elaborated; this includes representing employees, projects, and organizational groups (departments, teams). Representational constructs from the solution domain must also be identified so that they can be utilized to realize the resulting information system; this includes representing mechanisms for storing information (databases) and mechanisms for processing information (programs). Considering this problem, we must be able to represent "things," including the following:

- People (resources) or employees
- Projects
- Organizational groups (departments, teams)

We must be able to represent what composes these things, including the following:

- A person's name, social security number, employee number, etc.
- A project's tasks or activities, resources requirements and assignments, deadlines, etc.

We examine the concept of a thing and its composition in the section "Objects and Classes."

We must be able to represent relationships among things, including the following:

- People assigned to projects
- People belonging to organizational groups
- Projects managed by people (managers) within organizational groups

We examine relationships in the section "Links and Associations."

We must be able to represent interactions among things, including the following:

- People moving among projects
- People moving among organizational groups

We examine interactions in the section "Scenarios and Interactions."

Considering a solution to this problem, we must be able to represent a system and its components or elements as "things." We must be able to represent what constitutes these components, and how they are related to one another. We must be able to represent how these components cooperatively function or interact to

provide the desired functionality. These may be some of the many aspects that must be captured and manipulated to solve the problem. Furthermore, the information system must be tailored to address the needs of the organization.

To understand how the UML can be used to support problem solving, an understanding of the representational constructs provided by the UML is required. Such an understanding will enable you to deliberate on a problem and utilize the representational constructs provided by the UML to facilitate deriving the solution to the problem.

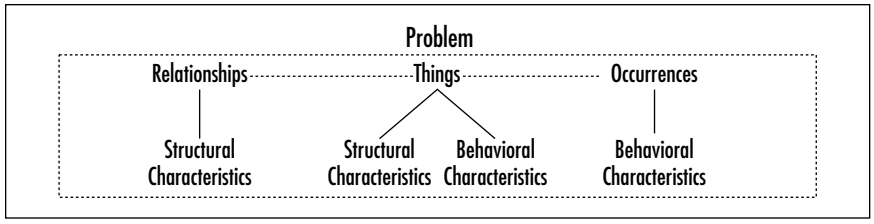

Figure 3-1: Real World

The real world (Figure 3-1) is the domain encompassing problems, solutions, and problem-solving efforts. The real world

- Is where problems must be conceptualized as input.
- Is where problem solving occurs.
- Is where solutions must be realized as output.
- Consists of things or entities that have a purpose or role within a problem or solution. Things can be classified as follows:
  - Concrete (real) entities have real existence in the world. They can be identified as existing independent of their perceiver.
  - Conceptual (abstract) entities do not have real existence in the world. They represent concrete entities, or can be attributed to concrete or other conceptual entities. They do not exist independent of their perceiver and are introduced into a situation to facilitate understanding.

  Things, whether concrete or conceptual, have the following types of characteristics or features:
  - Structural characteristics determine what a thing "knows" in order to sustain its purpose. These characteristics determine the possible states of an entity, that is, the conditions that an entity may attain at any time during its existence. Within a problem, these characteristics include datalike elements and express datalike requirements of a problem or a solution.
  - Behavioral characteristics determine the activities a thing "does" in order to achieve or sustain its purpose. These characteristics determine the possible behaviors of an entity, that is, the actions and reactions that an entity may perform at any time during its existence. Within a problem, these characteristics include algorithmlike elements and express processlike requirements of a problem or a solution.

- Consists of relationships among entities. *Relationships* are conceptual or concrete constructs that connect or associate two or more other entities. They may be treated as structural characteristics shared among multiple entities that have a relationship with one another.
- Consists of occurrences or dynamic characteristics within entities and among related and unrelated entities. *Occurrences* are conceptual or concrete constructs that involve something that happens within an entity or among two or more other entities. These events or incidents transpire as a result of entities sustaining their responsibilities. Dynamic characteristics determine how groups of entities interact and communicate based on their roles within domains. Occurrences may be treated as behavioral characteristics shared among multiple entities that participate or interact with one another.

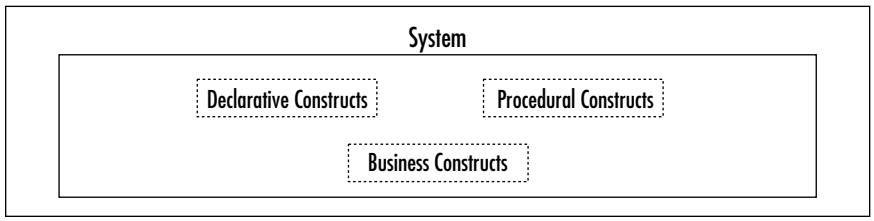

Figure 3-2: Solution Worlds

Solution worlds (Figure 3-2) are systems that solve problems and involve realization or implementation constructs. These are mechanisms that are used to implement solutions to problems. They include declarative constructs, procedural constructs, and business constructs.

Declarative Constructs

Declarative constructs represent the problem a system is to solve, and the solution once the problem is solved. These constructs

- Are data oriented; that is, they focus on datalike elements (information) within problems and solutions.
- Capture information regarding problems and solutions.
- Are often realized as data sections within programs or processes, tables within database management systems or file systems, and data stores within information systems.

The realization of declarative constructs involves the following:

- Data constructs that represent information.
- Variables that represent values that may be manipulated.
- Data types that determine the possible values and operations applicable for a variable.
- Primitive data types, which are existing or predefined data types.
- Complex data types, which are new or user-defined data types.
- Scope that determines the accessibility of a variable.

These constructs constitute half of the technology aspect of a solution.

Procedural Constructs

Procedural constructs manipulate and transform the problem into the solution. These constructs

- Are process oriented; that is, they focus on algorithmlike elements that manipulate problems and derive solutions.
- Capture processes for transforming or deriving a solution from a problem.
- Are often realized as code sections within programs or processes, stored procedures within database management systems or file systems, and processes within information systems.

The realization of procedural constructs involves the following:

- Code constructs that represent actions and activities.
- Subprograms that represent manipulators that manipulate data constructs. Subprograms consist of statements or expressions using sequential, conditional, and repetitive logic to express how data constructs are manipulated. They are also known as functions and procedures.
- Parameters, which are variables used to pass information to a subprogram or used to receive information from a subprogram. A subprogram declares formal parameters in which it receives and may return data. The expressions invoking the subprogram supply actual parameters and values to pass to the subprogram and variables to receive information passed back from the subprogram.
- Input parameters, which are used to pass information to a subprogram.
- Output parameters, which are used to receive information from a subprogram. They may send and receive information in variables.
- Return parameters, which are used only to receive information from a subprogram. They may only receive information in variables; that is, they don't send any information to a subprogram.
- Preconditions, which are conditions that must be true when a subprogram is invoked.
- Postconditions, which are conditions that must be true when a subprogram is finished executing.
- Scope that determines the accessibility of a subprogram by other subprograms. Global scope means it is accessible from anywhere within a system. Local scope means it is accessible from within another subprogram.
- Implemented systems, which are a collection of subprograms that invoke one another to accomplish some purpose for the end user.

These constructs constitute the other half of the technology aspect of a solution.

Business Constructs

Business constructs exist within a business context in which the solution is realized. These constructs include the following:

- Workers, who are people or teams, within organizations, that have responsibility for doing some set of activities.

- Work units, which are work products that are manipulated by workers participating in activities.
- Workflows, which are sequences of activities or relevant pieces of work, within an organization, that involve workers and work units. Workflows are the means through which value is derived for other workers or customers and clients.

These constructs constitute the human and organizational aspect of a solution.

Paradigms

The following paradigms are common:

- The *function-driven paradigm* focuses on behavioral and dynamic characteristics of a problem to derive the solution.
- The *data-driven paradigm* focuses on structural characteristics of a problem to derive the solution.
- The *object-oriented paradigm* focuses on problem concepts as a whole to derive the solution.

Function-Driven Paradigm

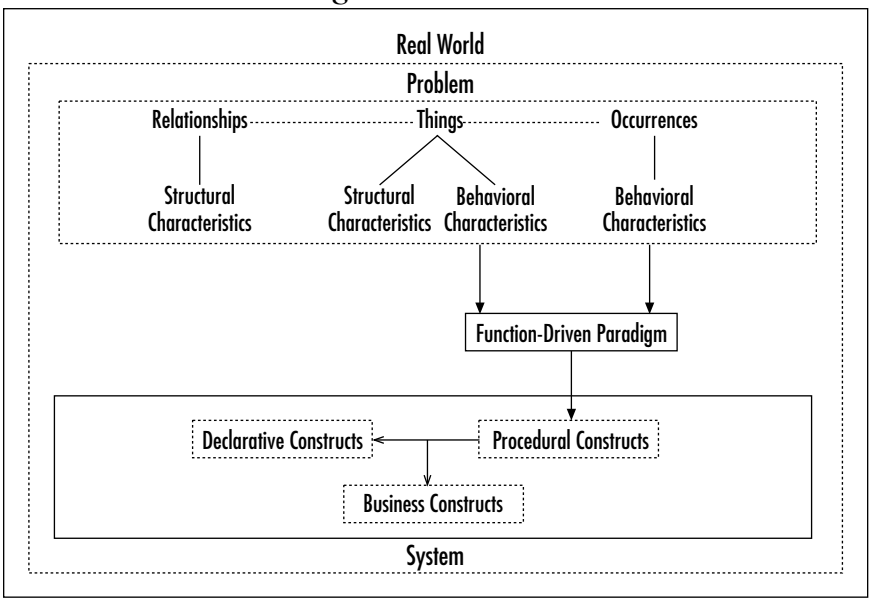

Figure 3-3: Function-Driven Paradigm

The function-driven paradigm (Figure 3-3) drives problem-solving efforts by emphasizing processlike elements of problems. Such approaches to problem solving are classified as function-centric. This paradigm

- Facilitates the following problem-solving approach to realize solutions (systems):

  - Identify processes within the problem. Process-oriented elements of problems are regarded as primary in facilitating and driving problem-solving efforts.
  - Identify data manipulated by processes. Data-oriented elements of problems are regarded as secondary in facilitating problem-solving efforts.
  - Decompose or partition and elaborate processes to further understand the problem and solution. Elaboration consists of specifying the inputs and outputs of processes and reducing processes into subordinate processes.
  - Derive a solution from this decomposition by satisfying process-oriented requirements of the problem with procedural constructs and then supporting declarative constructs.
  - Utilize an architecture based on process decomposition rather than data decomposition.
- Is more suitable for domains in which problems are more process intensive. This includes most real-time systems where there are more process elements than data elements involved in a system.

Data-Driven Paradigm

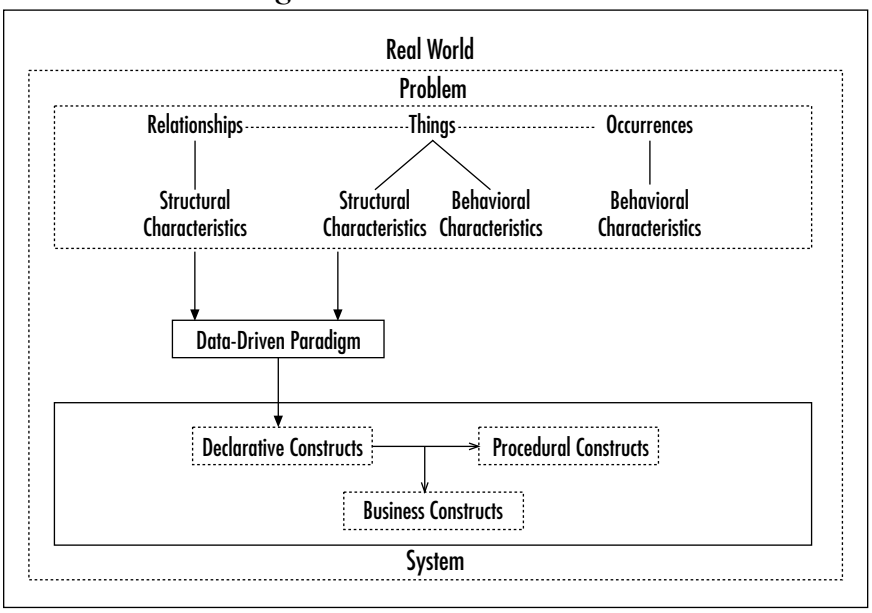

Figure 3-4: Data-Driven Paradigm

The data-driven paradigm (Figure 3-4) drives problem-solving efforts by emphasizing datalike elements of problems. Such approaches to problem solving are classified as data-centric. This paradigm

- Facilitates the following problem-solving approach to realize solutions (systems):

  - Identify data within the problem. Data-oriented elements of problems are regarded as primary in facilitating and driving problem-solving efforts.
  - Identify processes that manipulate data. Process-oriented elements of problems are regarded as secondary in facilitating problem-solving efforts.
  - Decompose or partition and elaborate data to further understand the problem and solution. Elaboration consists of specifying relationships among data elements and reducing data elements into subordinate data elements.
  - Derive a solution from this decomposition by satisfying data-oriented requirements of the problem with declarative constructs and then supporting procedural constructs.
  - Utilize an architecture based on data decomposition rather than process decomposition.
- Is more suitable for domains in which problems are more data intensive. This includes most business or database systems where there are more data elements than process elements involved in a system.

Object-Oriented Paradigm

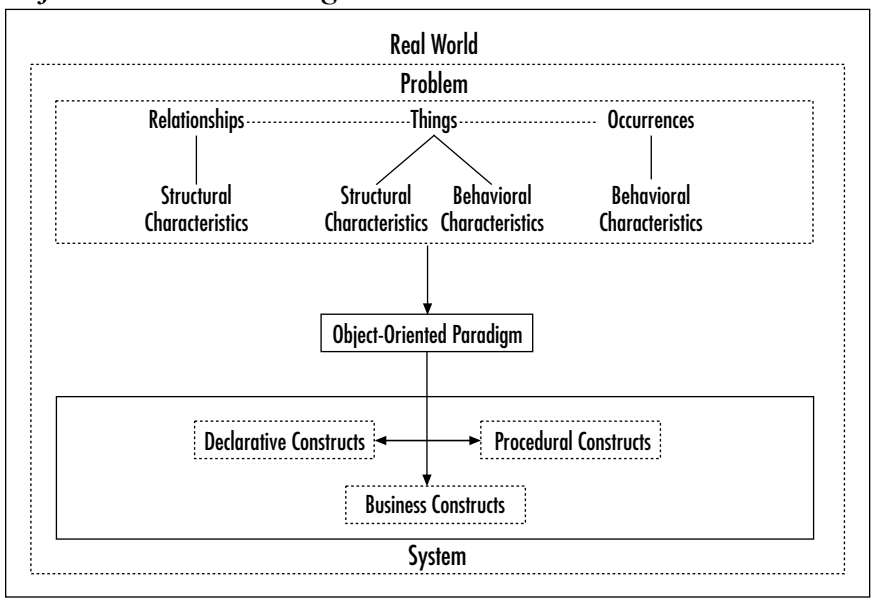

Figure 3-5: Object-Oriented Paradigm

The object-oriented paradigm (Figure 3-5) drives problem-solving efforts by emphasizing processlike and datalike elements of problems as complete units. Such approaches to problem solving are classified as concept-centric because they focus on both the processlike and datalike elements that constitute a given concept. This paradigm

- Focuses on understanding real-world concepts in terms of the composition, relationships, and interactions of entities.

- Provides intrinsic mechanisms for packaging datalike elements and process-like elements together to represent real-world concepts. These mechanisms mimic the naturally occurring scheme where entities have structural characteristics, behavioral characteristics, and dynamic characteristics.
- Provides intrinsic mechanisms for organizing real-world concepts into representations and extending representations of real-world concepts without having to change existing representations. These mechanisms mimic the naturally occurring scheme of how entities evolve over time.
- Facilitates the following problem-solving approach to realize solutions:
  - Identify and elaborate entities within the problem and solution.
  - Identify and elaborate relationships within the problem and solution.
  - Identify and elaborate occurrences within the problem and solution.
  - Regard some real-world concepts as essential to domains in which problems and solutions exist.
  - Regard other real-world concepts as incidental to domains in which problems and solutions exist.
  - Regard concepts as essential or incidental. The paradigm does not primarily focus on process-oriented elements or data-oriented elements; rather, it focuses on concepts as a whole.
  - Decompose or partition and elaborate concepts to further understand the problem and solution. Elaboration consists of elaborating structural characteristics, behavioral characteristics, and dynamic characteristics.
  - Derive a solution from this decomposition by satisfying real-world requirements with realization constructs.
  - Utilize an architecture based on real-world concept decomposition.

Comparing Paradigms

A comparison of the function-driven, data-driven, and object-oriented paradigms reveals the following.

- An ideal paradigm aspires to
  - Facilitate communication among stakeholders involved in a problem-solving effort.
  - Facilitate the creation and utilization of assets to deliver value and increase productivity and consistency.
  - Facilitate the management of change and complexity.
- The function-driven paradigm and data-driven paradigm have the following characteristics:
  - They extract realization constructs from within the problem domain. Stakeholders are forced to communicate using realization concepts; thus, communications may be hindered by being biased toward realization concepts. This often results in a technology-focused solution to a problem rather than the best business solution that is supported by technology.

  - They focus on process-oriented or data-oriented characteristics of the real world to drive problem-solving efforts. Artifacts become problem specific and skewed with a process-oriented or data-oriented emphasis; thus, artifacts are tightly coupled with the specific problem and are not likely to be dramatically exploited in other efforts requiring a different emphasis. Furthermore, attempts to reuse artifacts require extrinsic packaging, organizing, and extending mechanisms (library systems, component management systems, etc.) to mechanize the reuse process.
  - With a bias toward realization concepts and no intrinsic mechanisms for reuse, changes in requirements have a tendency to dramatically increase complexity within a system. Changes that require a shift in focus between data-oriented and process-oriented elements may cause extensive modifications in an existing system, thus increasing the complexity of the system. The severity of such changes can be very costly and time consuming.

- The object-oriented paradigm has the following characteristics:
  - It focuses on real-world concepts rather than realization constructs. Communications among stakeholders are more natural and focused on the real-world problem to be solved and its solution. This often establishes the foundation for a business-oriented solution that is supported by technology.
  - It focuses on real-world concepts rather than realization constructs. A foundation for reuse among multiple efforts is established. By providing intrinsic packaging mechanisms, representational constructs may be classified and organized by their purpose rather than how they are realized; thus, knowledge is packaged for the possibility of opportunistic reuse. By providing intrinsic organizing and extending mechanisms, emphasis is placed on the systematic reuse of artifacts across multiple problem-solving efforts and domains.
  - It focuses on real-world concepts and applying intrinsic mechanisms for packaging, organizing, and extending representational constructs. Solutions are more readily accommodating to change. Since the emphasis is on real-world concepts, changes in real-world concepts can be more readily realizable in solutions. Since real-world concepts are quite stable and don't change adversely (that is, the processes and procedures that involve these real-world entities do change, but the entities evolve rather than adversely change), solutions and the representational constructs they utilize are more stable and less likely to require adverse changes. As technology changes, the knowledge about real-world concepts may be reapplied using new technology to generate updated solutions. This minimizes the effort required to rediscover what was previously learned, but not captured in a reusable form, about the problem when it was originally solved using existing technology. Furthermore, intrinsic packaging mechanisms aid in managing the impact of changes on solutions by localizing modifications around representational constructs. This has the potential to deliver tremendous value.

- The object-oriented paradigm empowers organizations to gain a competitive advantage by delivering value (increasing quality, reducing costs, reducing

time to market) while managing complexity and change via the creation and use of assets.

Object Orientation

The object-oriented paradigm (Figure 3-6) focuses on abstractions that are closely associated with their real-world counterparts.

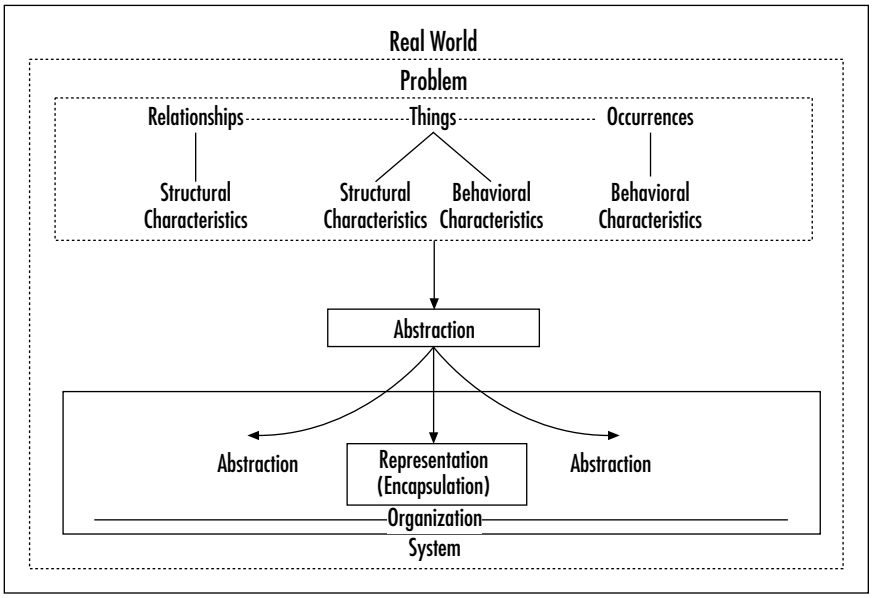

Figure 3-6: Object Orientation

Abstraction

Abstraction involves the formulation of representations concerning a subject in a world. Abstraction

- Involves the following steps:
  - Identifying the subject.
  - Constructing a representation of the subject.
- Establishes a level of detail at which attention and concentration are focused regarding the subject. This level of detail has the following properties:
  - It can be further elaborated to other levels of detail. At higher levels of abstraction, there is less granularity and detail regarding the subject. At lower levels of abstraction, there is more granularity and detail regarding the subject.
  - It establishes a level of detail at which stakeholders can communicate using a common perspective.
  - It constrains decisions regarding the subject to the particular level of detail, thus avoiding premature decisions and commitments.

- Focuses on similarities and differences among a set of identified particular examples or entities so that a representation can be formulated. This involves the following properties of abstraction:
  - Similarities among entities are combined into a single representational construct for the entities.
  - Differences among entities are separated into multiple representational constructs for the entities.
- Focuses on extracting intrinsic essential characteristics of the subject. These characteristics are necessary and sufficient to distinguish a given entity at a given level of detail from all other entities at the same level. This process involves identifying an entity and establishing a boundary around the entity.
- Focuses on avoiding extrinsic incidental characteristics of the subject. These characteristics are either not necessary for representing and manipulating the subject for a given purpose (problem solving) or may belong at a different level of abstraction.
- May be used to focus on the aspects of what something is and what it does while avoiding how it is realized.
- May be declarative, or data oriented, focusing on the logical or conceptual properties of data rather than the details of how data is actualized.
- May be procedural, or process oriented, focusing on the logical or conceptual properties of processes rather than the details of how processes are actualized.
- Facilitates understanding complex problems.
- Facilitates simplifying and manipulating complex problems to derive solutions.

Encapsulation

Encapsulation involves the packaging of representations concerning a subject in a world. Encapsulation

- Focuses on packaging datalike elements and processlike elements together as they occur within concepts themselves in their world.
- Mimics the naturally occurring scheme where entities have structural characteristics, behavioral characteristics, and dynamic characteristics.
- Focuses on the internal architecture of representational constructs or the architecture of a single representational construct.
- Focuses on enabling representational constructs to be self-contained.
- Utilizes modularity, which is the purposeful partitioning of representational constructs to manage size and complexity.
- Utilizes information hiding, which is the appropriate hiding of detail to facilitate abstraction. Information hiding distinguishes between the following aspects of a representational construct or entity:
  - A *specification* describes what an entity is and what an entity does. It is a declarative description used to define interfaces through which communi-

cation with the entity may occur. This involves specifying the outside view of the entity.

– An *implementation* describes how an entity is realized. It is a declarative description used to define how the entity may be realized. This involves specifying the inside view of the entity.

Information hiding facilitates abstraction by partitioning what we know into levels of detail. A level of detail is not concerned with how much of the specification or implementation is discovered and captured; rather, it is concerned with a level of abstraction for describing the inside and outside view of an entity.

- Utilizes localization, which is the physical grouping of logically related constructs to maintain unity among related constructs. Localization facilitates the following:
  – Increased cohesion, or maximizing intradependencies within an entity. Cohesion is a measure of how parts of a whole are logically related to each other and the overall whole. Higher cohesion indicates that changes will more likely be localized. This is maximized by the use of modularity.
  – Decreased coupling, or minimizing interdependencies among multiple entities. Coupling is a measure of the strength of the connections between parts. Tighter coupling among representational constructs indicates that changes will less likely be localized to a single construct since one construct knows more about the implementation of another construct. This is minimized by adhering to and using interfaces.
- Enables representational constructs to communicate and interact together via interfaces without establishing interdependencies among their implementations.
- Enables the localization and containment of changes.
- Is used to combat complexity and localize the impact of changes.

Organization

Organization involves the relating and reusing of representations concerning a subject in a world. Organization

- Enables new representations to be variations of existing representations.
- Focuses on evolutionary relationships as they occur among concepts in a world.
- Mimics the naturally occurring scheme of how entities evolve over time.
- Focuses on the external architecture among representational constructs, or the architecture among multiple representational constructs.
- Focuses on classifying representations into higher or lower levels of abstraction.
- Enables the evolution of existing representational constructs.
- Is used to combat complexity and support the reusability of representational constructs.

Objects and Classes

Objects and classes (Figure 3-7) abstract *entities* from the problem world or solution world.

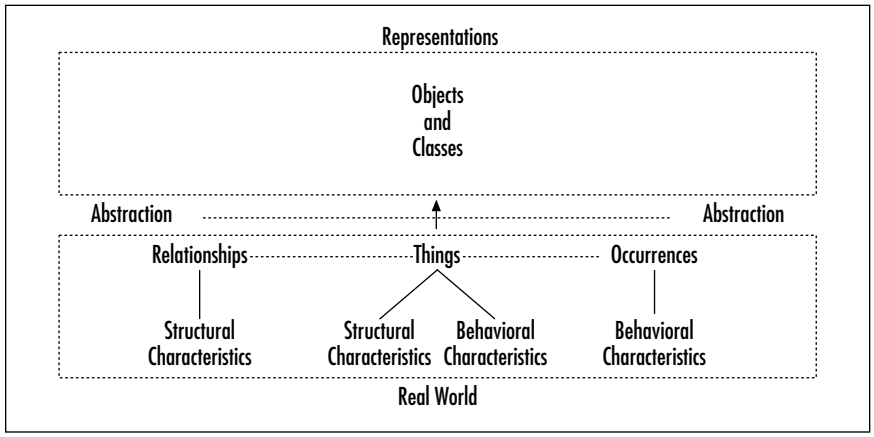

Figure 3-7: Objects and Classes

Objects

Objects (Figure 3-8) are well-defined representational constructs of concrete or conceptual entities.

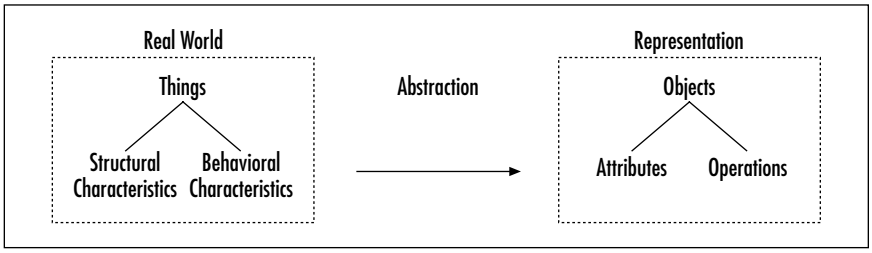

Figure 3-8: Objects

Objects

- Encapsulate structural characteristics known as *attributes*. Attributes
  - Are representational constructs of structural characteristics of entities.
  - Determine the possible states of an object.
  - Are extracted from a domain using declarative, or data-oriented, abstraction.
  - May be of a simple or complex data type. Simple data types are not reducible to any subordinate parts. Complex data types are conglomerates reducible to subordinate parts.

  - May be single valued or multi-valued. Single-valued attributes resolve to one value. Multi-valued attributes resolve to a collection or set of values.
- Encapsulate behavioral characteristics known as *operations*. Operations
  - Are representational constructs of behavioral characteristics of entities.
  - Determine the possible behaviors of an object.
  - Are extracted from a domain using procedural, or process-oriented, abstraction.
  - Have a signature consisting of a name, input parameters, output parameters, and possibly return parameters.
  - Are invoked in response to receiving a message.
- May be active or passive.
  - *Active* objects have a thread of control or may initiate activity. They may take the initiative to request services from other objects.
  - *Passive* objects do not have a thread of control and may not take the initiative to request services from other objects unless they receive control from an active object.
- May be persistent or transient. *Persistent* objects exist after their creator has ceased to exist. *Transient* objects exist only during the time that their creator exists.
- May be referenced, that is, denoted in some manner.
- Have identity. All objects are unique and distinguishable from other objects. They may be compared for the following:
  - Pure identity, yielding true for the same object.
  - Shallow equality, yielding true for objects with the same attribute values.
  - Deep equality, yielding true for objects with the same attribute values recursively within their subordinate parts.
- Participate in relationships.
- Participate in occurrences.
- Have semantics, that is, meaning or purpose within a problem or solution.
- May be complex and reducible into subordinate objects, or may be primitive and irreducible.
- Are instances of classes. They are said to *instantiate* classes. The relationship between an object and its class is known as an *is-a* relationship.

Classes

Classes (Figure 3-9) are descriptions of objects or a set of objects with a common implementation. Classes

- Are concerned with the implementation of uniform structural characteristics and behavioral characteristics.

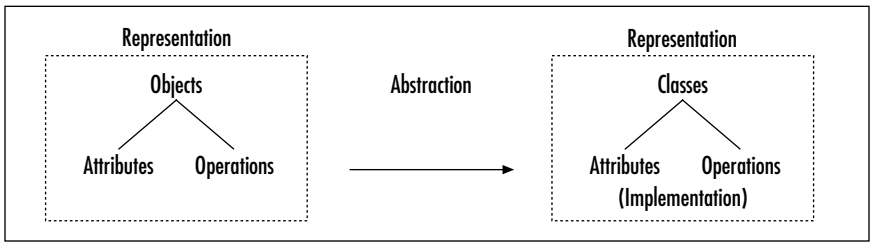

Figure 3-9: Classes

- Have an intensional notion or capability. This is the ability of a class to define a pattern for objects of the class. A class determines the structure and behavior of these objects.
- Have an extensional notion or capability. This is the ability of a class to create objects of the class. The class is known as an *object factory*. The extensional capability is also the ability of a class to maintain references to all objects of the class. The objects of a class are known as the class's *extent*.
- Encapsulate structural characteristics called attributes. They
  - Are implemented by the class.
  - May be associated with objects of a class. That is, each object has its own value. These are known as *object attributes* and have a scope that affects an individual object.
  - May be associated with the class as a whole. That is, all objects of a class share a value. These are known as *class attributes* and have a scope that affects all objects of the class.
- Encapsulate behavioral characteristics called operations. They
  - Are implemented by the class as methods or subprograms.
  - Are known as *services* offered by the class (or objects of the class).
  - May be associated with objects of a class. That is, they may be applied on objects of the class. These are known as *object operations* and have instance or object scope.
  - May be associated with the class as a whole. That is, they may be applied on the class itself. These are known as *class operations* and have class scope.
  - May be classified as *abstract*. That is, they may specify an interface but no implementation. When a class has one or more abstract operations, it may not have any instances. Once all abstract operations have an implementation provided by a subclass, the subclass may have instances. When a class has all operations defined, it is classified as *concrete*. By making an operation abstract, the class may use the operation; however, the implementation of the operation is delayed until a subclass provides a method. Subclasses are discussed in the section "Generalizations."
- Define the accessibility of attributes and operations from outside an object of a class. The following accessibility criteria may be associated with characteristics:

  - *Public* accessibility means characteristics are accessible from outside an object.
  - *Protected* accessibility means characteristics are not accessible from outside an object. They are like private characteristics when accessed from outside an object of a class.
  - *Private* accessibility means characteristics are not accessible from outside an object.
- May have an introspective protocol or mechanism through which attributes and operations may be discovered by other objects, and an intercessory protocol or mechanism through which attributes and operations may be added, deleted, or modified by other objects.
- May be parameterized; that is, may require parameters in order to become a defined class that is capable of instantiation. The parameters are bound to actual values to create a real class. Parameterized classes are known as *template classes* or *generic classes*.
- May allow their objects to be statically or dynamically classified. Static classification disallows objects from changing their classes. Dynamic classification allows objects to change their classes.
- May allow their objects to be singly or multiply classified. Single classification disallows objects from belonging directly to more than one class. Multiple classification allows objects to belong directly to more than one class.

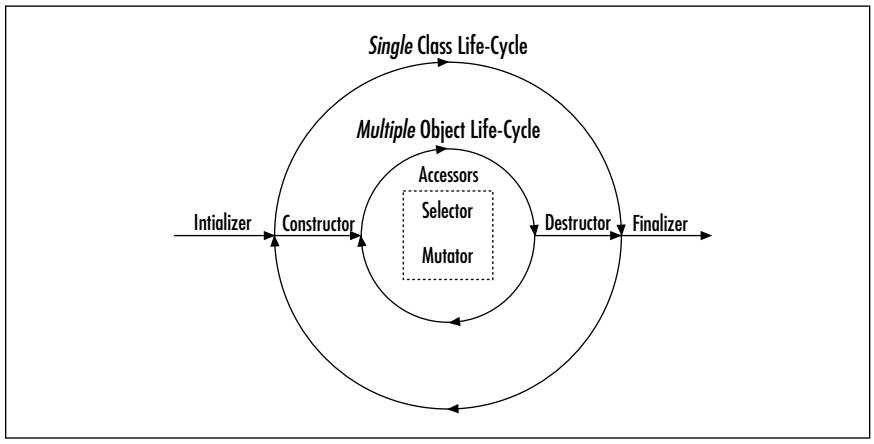

Figure 3-10: Life Cycle of Classes and Objects

- Have a life cycle (Figure 3-10) that is shared with their objects. This life cycle is depicted by the following types of operations:
  - *Initializer operations* are class operations that are implicitly utilized to initialize a class. They are invoked when the first object of the class is created.
  - *Constructor operations* are class operations that create or construct objects of the associated class.

  - Accessors known as *selector operations* provide an interface for "getting" information about and from within an object or class.
  - Accessors known as *mutator operations* provide an interface for "setting" information about and within an object or class. Mutator operations also involve the general behavior of the object or class.
  - *Destructor operations* are class operations that destroy objects of the associated class.
  - *Finalizer operations* are class operations that are implicitly utilized to uninitialize a class. They are invoked when the last object of the class is destroyed.
- Participate in relationships.
- Participate in occurrences.
- Define an implementation for their objects.
- May be complex and reducible into subordinate classes or may be primitive and irreducible.

Types

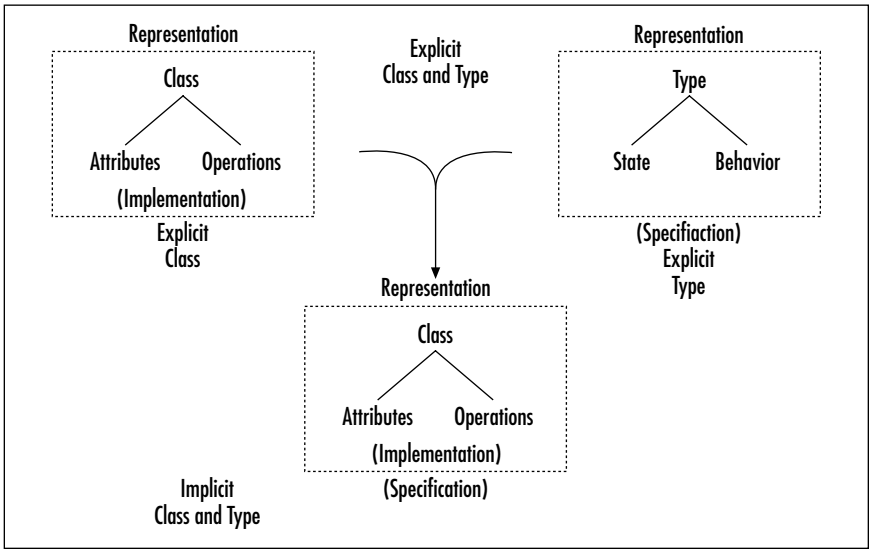

Figure 3-11: Types

Types (Figure 3-11) are descriptions of objects or a set of objects with a common specification or interface. Types

- Are concerned with the specification of uniform structural characteristics and behavioral characteristics.
- Encapsulate structural characteristics called attributes. They
  - Are specified by the type.

  - May be object attributes or class attributes.
- Encapsulate behavioral characteristics called operations. They
  - Are specified by the type.
  - May be object operations or class operations.
- Define the accessibility of attributes and operations from outside an object or class using the same accessibility criteria as classes.
- Participate in relationships.
- Participate in occurrences.
- Define a specification or one or more interfaces for classes.
- May be complex and reducible into subordinate types or may be primitive and irreducible.
- May be explicitly related to classes. That is, the class of an object may be distinguished from the type of an object. This involves a class receiving an interface from a type and providing an implementation for the interface.
- May be implicitly related to classes. That is, the class of an object implicitly includes its type. This is accomplished via the class's intensional notion and involves a class defining an interface and providing an implementation.

Links and Associations

Links and associations (Figure 3-12) abstract *relationships* among entities within the problem world or solution world.

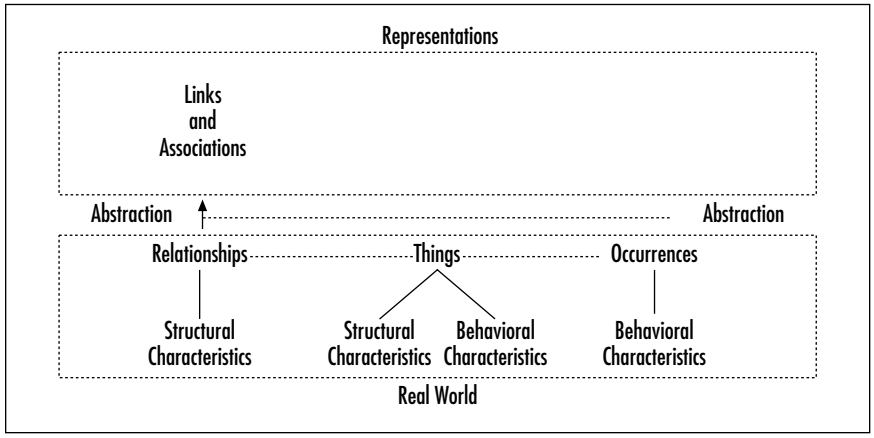

Figure 3-12: Links and Associations

Links

Links (Figure 3-13) are well-defined representational constructs of concrete or conceptual entities that relate other entities. Links

- Are objects that relate other objects.
- Are dependent on all of the objects they relate.

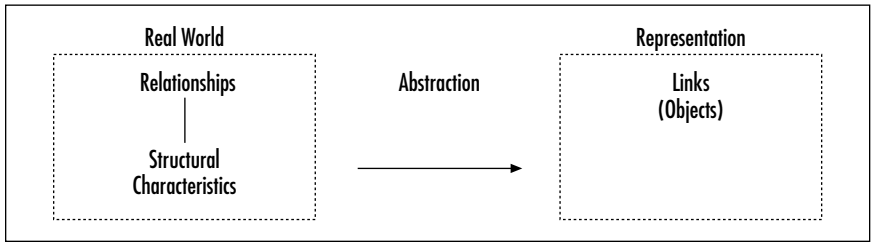

Figure 3-13: Links

- Are not part of any of the objects they relate.
- May be binary, ternary, or higher order. That is, they may relate two, three, or more than three other objects.
- Are instances of associations. They are said to instantiate associations. The relationship between a link and its association is known as an *is-a* relationship.

Associations

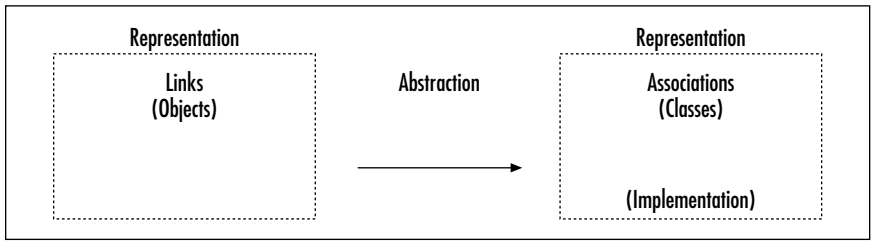

Figure 3-14: Associations

Associations (Figure 3-14) are descriptions of links or a set of links with a common implementation. Associations

- Are classes that relate other classes.
- Are to links as classes are to objects.
- Define an implementation for their links.

Aggregations

Aggregations (Figure 3-15) are associations and links specifying a whole-part relationship. Aggregations

- Are known as *has-a* relationships.
- Are abstractions of concrete or conceptual whole-part relationships among entities.
- Involve aggregates or wholes that are connected to their component parts.
- Involve component parts that exist independent of their aggregates.

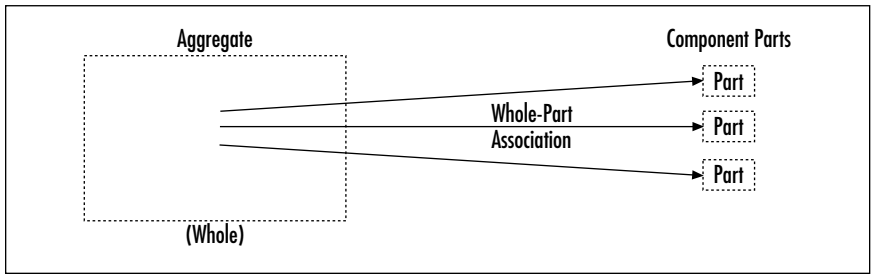

Figure 3-15: Aggregations

- Specify that aggregates "loosely" contain component parts. The relationship is "loose" because there are no other semantics associated with the relationship.
- Are transitive; that is, if *A* is a part of *B* and *B* is a part of *C*, *A* is a part of *C*.
- Are antisymmetric; that is, if *A* is a part of *B*, *B* is not a part of *A*.

Compositions

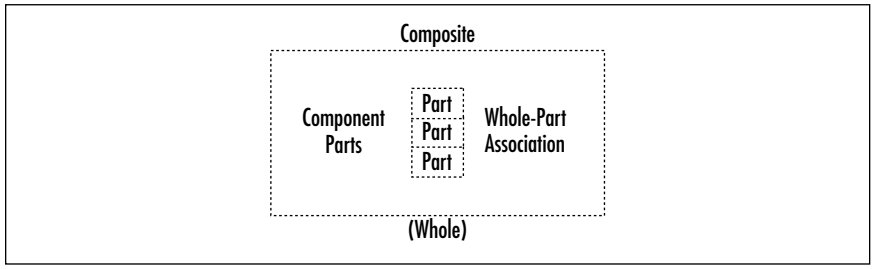

Figure 3-16: Compositions

Compositions (Figure 3-16) are aggregations with strong ownership and coincident lifetime constraints. These are also known as *composite aggregations*. Compositions

- Are known as *contains-a* relationships.
- Are abstractions of concrete or conceptual whole-part relationships with ownership and coincident lifetime constraints among entities.
- Specify that composites or aggregates own their component parts.
- Specify that the component parts may only have one owner.
- Specify that component parts exist or live and die with their composite owner.
- Specify that composites "tightly" contain component parts. The relationship is "tight" because there are semantics other than aggregation associated with the relationship. That is, parts live and die with their owners.

Generalizations

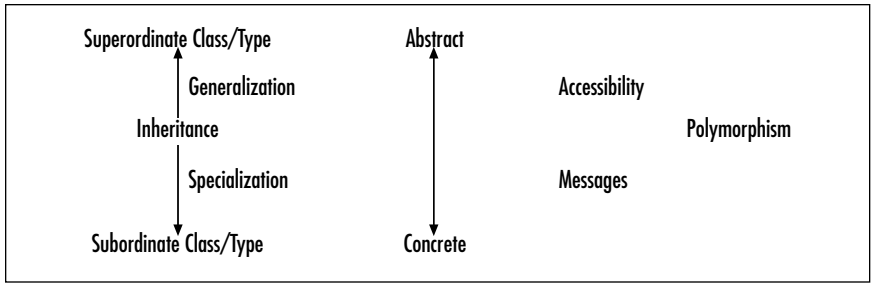

Figure 3-17: Generalizations

Generalizations (Figure 3-17) are associations specifying a taxonomic relationship. Generalizations

- Are known as *is-a-kind-of* relationships.
- Relate more general representational constructs and more specific representational constructs. These constructs include entities, classes, and types.
- Enable new constructs to be derived from existing constructs.
- Involve existing classes and types known as *superclasses* (superordinate classes) and *supertypes* (superordinate types). These classes and types are also called *ancestors* or *base constructs*. They have the following characteristics:
  - Types provide existing specifications or interfaces.
  - Classes provide existing implementations.
- Involve new classes and types known as *subclasses* (subordinate classes) and *subtypes* (subordinate types), also called *descendent* or *derived* classes and types. They have the following characteristics:
  - They acquire the characteristics of their superclasses and supertypes.
  - They can add characteristics including attributes and operations.
  - They can redefine operations.
- Indicate that more specific constructs are derived from more general constructs.
- Are called *specializations* to indicate that more specific constructs are derived from more general constructs.
- Utilize *inheritance* as the sharing mechanism through which more specific constructs acquire the characteristics (attributes, operations, methods, and associations) of more general constructs. Inheritance has the following characteristics:
  - Subtyping enables inheritance of specifications or interfaces. This is known as *interface inheritance* and establishes an interface hierarchy.
  - Subclassing enables inheritance of implementation. This is known as *implementation inheritance* and establishes an implementation hierarchy.

  - In single inheritance, a more specific construct receives characteristics from one or more general constructs, resulting in a treelike hierarchy.
  - In multiple inheritance, a more specific construct receives characteristics from multiple general constructs, resulting in a latticelike hierarchy.
  - Inheritance is transitive; that is, if *A* is a subclass of *B* and *B* is a subclass of *C*, *A* is a subclass of *C*. If *A* is a subtype of *B* and *B* is a subtype of *C*, *A* is a subtype of *C*.
  - Inheritance is antisymmetric; that is, if *A* is a subclass of *B* and *B* is a subclass of *A*, *A* and *B* are the same class. If *A* is a subtype of *B* and *B* is a subtype of *A*, *A* and *B* are the same type.
  - Inheritance enables substitution; that is, a subclass instance may be substituted where a superclass instance is required, and a subtype instance may be substituted where a supertype instance is required. The reverse is not true, however: a superclass cannot be substituted for a subclass.
  - Inheritance enables inclusion; that is, a subclass instance, is a superclass instance, and a subtype instance is a supertype instance.
  - Inheritance enables specialization; that is, a subclass instance is a superclass instance with more specific information, and a subtype instance is a supertype instance with more specific information.
- May be used to create abstract classes or concrete classes.
  - Abstract classes are incompletely implemented; that is, the class has interfaces without implementations. Abstract classes may not have any instances.
  - Concrete classes are completely specified and completely implemented; that is, the class has implementations for all interfaces. Concrete classes may have instances.
- Affect the life cycle for classes and types in the following manner during instantiation of an object:
  - Initializer operations are invoked from most general to most specific.
  - Constructor operations are invoked from most general to most specific.
  - Destructor operations are invoked from most specific to most general.
  - Finalizer operations are invoked from most specific to most general.
- Affect the accessibility of attributes and operations from outside an object, class, or type in the following manner:
  - Public accessibility means that characteristics are accessible when inherited by more specific constructs.
  - Protected accessibility means that characteristics are accessible when inherited by more specific constructs. They are like public characteristics when inherited.
  - Private accessibility means that characteristics are not accessible when inherited by more specific constructs.

More specialized constructs may further restrict the accessibility of inherited characteristics.

- Involve *polymorphism* (meaning "many forms"), that is, the ability of an interface to have many implementations.
- Involve *overloading;* that is, the ability of an operation to have the same name, different signatures, different implementations, and possibly different semantics within the same class or different classes. It enables the same message to invoke different operations within the same class or different classes. The invoked operation is determined by statically considering the signature of the message. This is also known as *ad hoc polymorphism.*
- Involve *parametric polymorphism,* that is, the use of parameterized classes and types. The invoked operation is a generic function that is dependent on the class or type of the arguments used in the operation invocation. These functions are known as generic functions or templates.
- Involve *pure polymorphism,* that is, the ability of an operation to have the same name, same signature, different implementations, and possibly different semantics within different classes. This involves *overriding,* that is, the ability of a subclass to specialize an inherited operation by redefining the implementation of the operation but not its specification. It enables the same message to invoke different operations within different classes based on the class of the object on which the operation is invoked. The invoked operation is determined by dynamically searching the inheritance hierarchy. The actual implementation of an operation is found by searching the class of the object on which the operation is applied. If no implementation is found, the class's superclasses are searched; this process continues until an implementation is found. This is known as *inclusion polymorphism* or *polymorphism by inheritance.* The operations are said to be *virtual.*
- Involve *delegation,* that is, the ability of an object or class to issue a message to another object or class in response to a message.

Scenarios and Interactions

Scenarios and interactions (Figure 3-18) abstract *occurrences* among entities within the problem world or solution world.

Scenarios

Scenarios (Figure 3-19) are well-defined representational constructs of concrete or conceptual entities that are conduits for a sequence of message exchanges among other entities.

Scenarios

- Are objects that are conduits for a sequence of message exchanges among other objects.
- Are concerned with a specific sequence of message exchanges (Figure 3-20). The specific sequence involves messages and actions between a sender (cli-

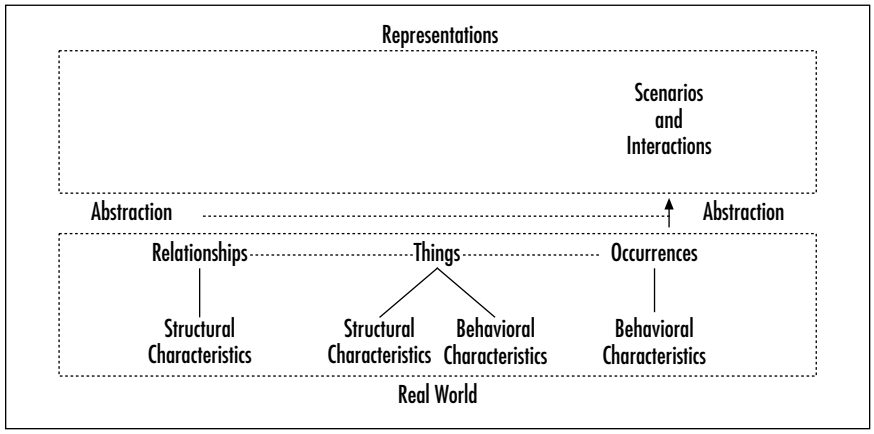

Figure 3-18: Scenarios and Interactions

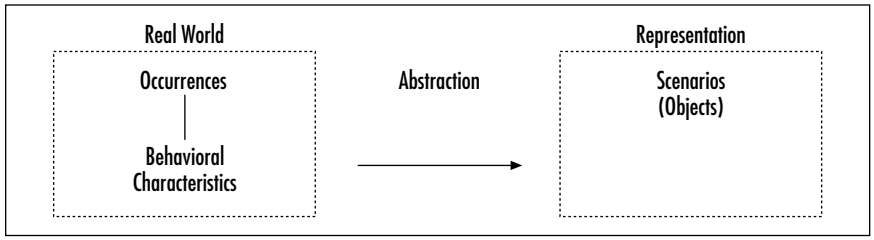

Figure 3-19: Scenarios

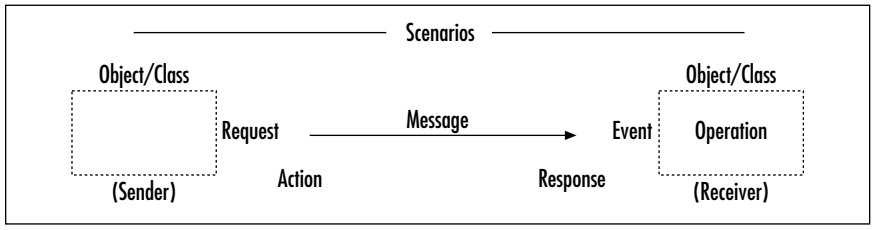

Figure 3-20: Message Exchanges

ent) object or class and a receiver (supplier or server) object or class in the following manner:

- The sender sends a message (operation or signal) or request to the receiver. The message is an instance of a message class and conveys information with the expectation that the receiver will act or perform some activity.
- The receiver receives the message as an event and responds by performing an activity. The event is an instance of an event class. The

response involves the invocation of an operation that is carried out by the operation's implementation or method.

  - Synchronous communication involves the sender waiting for the receiver to respond. The sender calls the receiver, and the event generated is a call event. If the receiver is not an active object, the sender must pass control to the receiver and wait for the receiver to respond and send control back to the sender.
  - Asynchronous communication involves the sender not waiting for the receiver to respond. The sender signals the receiver (or raises the signal), and the event generated is a signal event. The receiver must be an active object, and both the sender and receiver continue processing.
  - This manner of interaction may continue.

  Fundamentally, the sender is said to apply an operation on the receiver by sending a message that invokes the appropriate operation within the receiver.

  Within an interaction

  - Messages focus on the communication of information and requests for some activity.
  - Events focus on the occurrence of receiving a message. These may be call events or signal events.
  - Operations focus on specifying a service that is offered.
  - Methods focus on the implementation of a service.
- Are dependent on all of the objects involved in the sequence of message exchanges.
- Are not part of any of the objects involved in the sequence of message exchanges.
- May be binary, ternary, or higher order. That is, they may involve two, three, or more than three other objects.
- Are instances of interactions.

Interactions

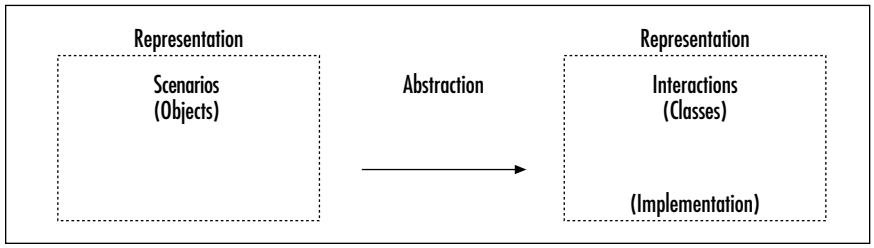

Figure 3-21: Interactions

Interactions (Figure 3-21) are descriptions of scenarios or a set of scenarios with a common implementation. Interactions

- Are classes that are conduits for a set of message exchange sequences among other classes.
- Are concerned with a set of message exchange sequences.
- Are to scenarios as classes are to objects.
- Are divided into two constructs:
  - *Sequences* focus on the messages exchanged among objects (classes) within scenarios.
  - *Collaborations* focus on the messages exchanged among objects (classes) and their links (associations) within scenarios.

Variations and Summary

Variations of the Object-Oriented Paradigm

An in-depth analysis of object-oriented paradigms (Figure 3-22) reveals the following points.

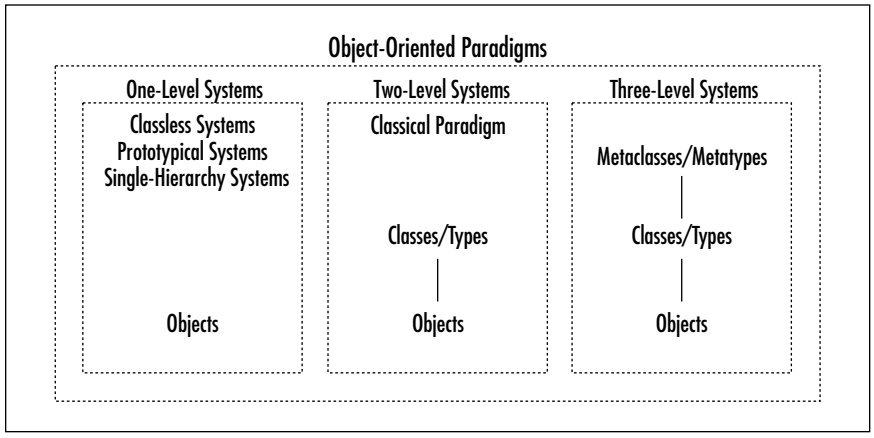

Figure 3-22: Object-Oriented Paradigms

- The following aspects of the object-oriented paradigm establish the foundation for various semantic variations or interpretations of the paradigm:
  - Definition and interpretation of the concepts that constitute the paradigm.
  - Application and utilization of the paradigm.
  - Closure and extensibility of the paradigm.

  All object-oriented paradigms must encompass and support the pillars (first principles) of object orientation
  - Abstraction as an identification principle.
  - Encapsulation as an organizing principle.
  - Inheritance as a sharing principle.

  - Polymorphism as a sharing principle.
- To distinguish among variations of the object-oriented paradigm, the following classification based on the distinct levels of instantiation may be utilized:
  - One-level systems, known as classless systems, prototypical systems, or single-hierarchy systems, stipulate that all objects are classes and all classes are objects. An instance is created by copying another object.
  - Two-level systems, known as classical object orientation, involve objects that are instances of classes and types. This is the most common interpretation of the object-oriented paradigm.
  - Three-level systems involve objects that are instances of classes and types, and classes and types that are instances of metaclasses and metatypes. Metaclasses are classes whose instance are classes. Metatypes are types whose instance are types.
  - *n*th-level systems involve *n* levels of instantiation.
- The UML utilizes a three-level system that includes an extension mechanism for metamodel access and customization. The UML may also be used for working with one-level systems and two-level systems.
- The object-oriented paradigm is reducible to the function-driven paradigm or the data-driven paradigm. That is, while using the object-oriented paradigm, activities can be skewed to independently leverage function-driven concepts or data-driven concepts in a given problem-solving approach. Therefore, because the UML is based on the object-oriented paradigm, it may be used to facilitate and express artifacts within a function-driven approach, a data-driven approach, or an object-oriented approach. This is possible because the object-oriented paradigm is a convergence of the other two paradigms.
- Applying the object-oriented paradigm within a problem-solving effort does not imply any specific constraints on the realization of the solution. That is, using the object-oriented approach does not imply that a realization toolset must be object oriented or support the paradigm. An object-oriented system may be implemented using declarative constructs and procedural constructs using an object-oriented toolset or a non-object-oriented toolset. This is possible because the object-oriented paradigm is a convergence of the other two paradigms and may be mapped to tools implementing the other paradigms.

Summary of the Object-Oriented Paradigm

The object-oriented paradigm (Figure 3-23) is summarized in the following.

- The real world consists of concrete and conceptual entities, including things, relationships, and occurrences that have a purpose, structural characteristics, and behavioral characteristics.
- The first level of abstraction within the object-oriented paradigm consists of representational constructs of concrete and conceptual entities.
  - Objects are abstracted entities that encapsulate state and behavior.
  - Links are abstracted relationships among objects.

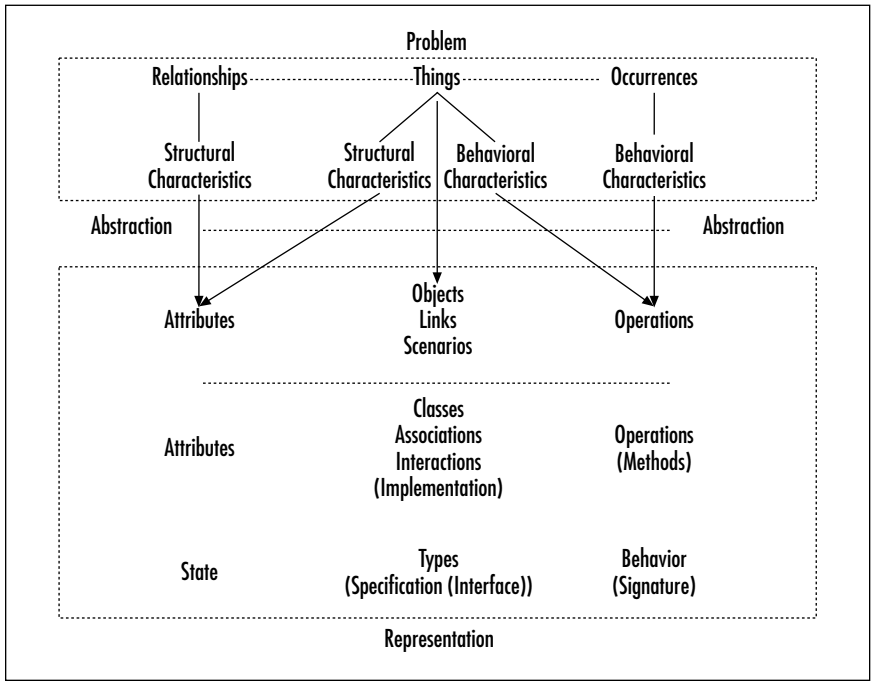

Figure 3-23: Summary of the Object-Oriented Paradigm

  - Scenarios are abstracted message exchanges among objects.
- The second level of abstraction within the object-oriented paradigm consists of a set of representational constructs.
  - Classes are descriptions of a set of objects with common attributes, operation implementations, semantics, associations, and interactions.
  - Types are descriptions of a set of objects with common attributes, operation interfaces, semantics, associations, and interactions.
  - Associations are descriptions of a set of links with common attributes, operation implementations, semantics, associations, and interactions.
  - Interactions are descriptions of a set of scenarios with common message exchange sequences, classes, and associations.
- Objects may be combined to build larger or more involved components. This has become known as the *component-oriented paradigm.*

Society of Objects

The impacts of the object-oriented paradigm on the notion of a system (Figure 3-24) are summarized in the following.

- A *society* is an organized collection of entities joined together for some purpose. The evolution of a society is the summation of the evolution of its entities.

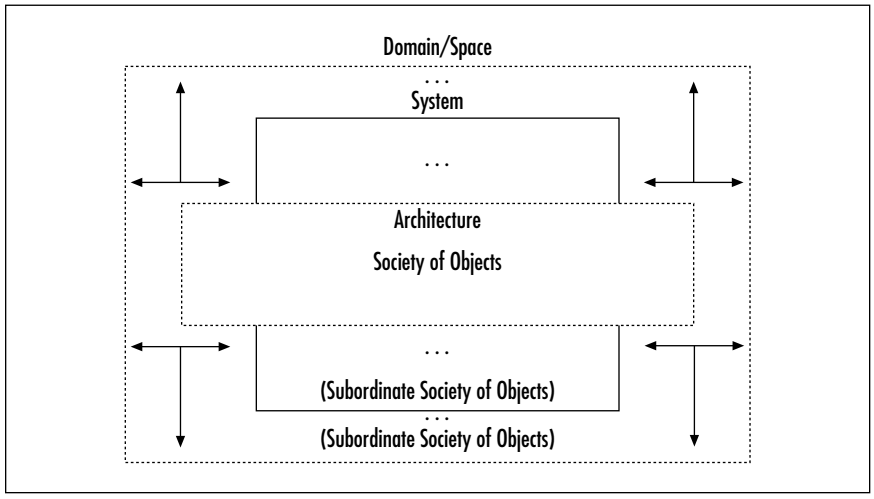

Figure 3-24: Society of Objects

- A society may be recursively decomposed into multiple societies called subordinate societies (or subsocieties).
- When fully decomposed, a society consists of primitive society elements, or objects that are not further decomposable.
- Using the object-oriented paradigm, the architecture of a system within a domain consists of a society of objects and links that interact to achieve the goals of the society.
- Societies of objects that are focused on specific domains and provide appropriate interfaces for adaptability within other societies of objects are known as *frameworks*.
- Objects may be combined to build larger or more involved components. This has become known as the component-oriented paradigm.
- Objects
  - Are abstracted from a world or domain.
  - Live within a society of objects.
  - Encapsulate knowledge in attributes and encapsulate skills in operations.
  - May be classified based on common implementations or interfaces.
  - May inherit characteristics from their ancestors.
  - Communicate via messages.
  - Cooperate in interactions to accomplish complex tasks.
  - Make requests of other objects by sending messages.
  - Fulfill requests made by other objects when receiving messages.
  - Perform operations in response to requests.

- May respond differently to the same request. That is, they respond in their own manner as long as the response is semantically consistent with other object's responses to the same message.
- Need not disclose exactly how they respond to requests.
- May not express exactly how they want other objects to respond to requests. They may express only what they are requesting other objects to do, not how it should be done.
- May determine the appropriate response when requests are received. That is, they are not bound to predetermined responses.

The UML is used to specify, visualize, construct, and document a society of objects that are organized to achieve some purpose.

,ch03.21735 Page 70 Wednesday, May 9, 2001 4:20 PM