Appendix A

UML Graphical Notation

The inside covers of the book summarize the graphical notations for the class, state, and interaction models. You can use these four pages as a quick reference while constructing or reading diagrams. However, we must caution you that a novice cannot simply take these four pages and understand them. To understand the concepts represented by the notation, refer to the chapters of Part 1. To learn how to apply the notation and concepts within the software development life cycle, consult the chapters in Part 2 and 3. The index can also help you find relevant material in the book.

With the exception of the label for each construct and a few descriptive comments, all of the diagram elements, text names, and punctuation symbols shown are part of the notation. The names in these diagrams (such as *Class*, *attribute1*, *operation*, and *event2*) indicate what kind of element they are examples of. You may wish to modify the syntax of names and the declarations of attributes and signatures to make them consistent with the syntax of your implementation language.

Most of the items shown are optional, especially during early stages of modeling. Even in design, it is unwise to overspecify by including superfluous names and notations. For example, when an association is labeled by end names, it is usually not necessary to give the association itself a name. We have not indicated which elements are optional, because we wanted to show only the actual UML notation wherever possible, without obscuring it with an additional metanotation.

Please feel free to copy the notation summaries on the inside covers. You can obtain an electronic copy at www.modelsoftcorp.com.

Appendix B

Glossary

The following terms are used in OO modeling for analysis, design, and implementation.

abstract class a class that has no direct instances. The UML notation is to italicize an abstract class name or place the keyword *{abstract}* below or after the name. (Contrast with *concrete class*.)

abstract operation an operation that lacks an implementation. A concrete descendant class must provide a method to implement the operation. The UML notation is to italicize an abstract operation name or place the keyword *{abstract}* after its name.

abstraction the ability to focus on essential aspects of an application while ignoring details.

access modifier (in Java) the means of controlling access to methods and data via *public*, *private*, *protected*, and *package* visibility.

access specifier (in C++) the means of controlling access to methods and data via *public*, *private*, and *protected* visibility as well as a *friend* declaration.

activation the period of time for an object's execution. The UML notation is a thin rectangle. (Synonymous with *focus of control*.)

active object an object that has its own thread of control. (Contrast with *passive object*.)

activity a specification of executable behavior.

activity diagram a diagram that shows the sequence of steps that make up a complex process.

activity token a token that can be placed on an activity symbol to show the progress of an execution.

actor a direct external user of a system. The UML notation is a "stick man" icon.

aggregation a kind of association in which a whole, the assembly, is composed of parts. Aggregation is often called the "a-part-of" or "parts-explosion" relationship and may be

nested to an arbitrary number of levels. Aggregation bears the transitivity and antisymmetry properties. The UML notation is a small *hollow* diamond superimposed on the association end next to the assembly class. (Contrast with *composition.*)

analysis the development stage in which a real-world problem is examined to understand its requirements without planning the implementation.

ancestor class a class that is a direct or indirect superclass of a given class. (Contrast with *descendant class.*)

API (acronym) *application programming interface.*

application analysis the second substage of analysis that addresses the computer aspects of the application that are visible to users.

application programming interface a collection of methods that provide the functionality of an application.

architecture the high-level plan or strategy for solving an application problem.

assembly (for an aggregation) a class of objects that is composed of part objects.

association a description of a group of links with common structure and common semantics. The UML notation is a line between classes that may consist of several line segments.

association class an association that is also a class. Like the links of an association, the instances of an association class derive identity from instances of the related classes. Like a class, an association class can have attributes, operations, and participate in associations. The UML notation is a box (a class box) attached to the association by a dashed line.

association end an end of an association. A binary association has two ends, a ternary has three ends, and so forth.

attribute a named property of a class that describes a value held by each object of the class. The UML notation lists attributes in the second compartment of the class box.

automatic transition an unlabeled transition that automatically fires when the activity associated with the source state is completed.

bag an unordered collection of elements with duplicates allowed. The UML notation is to annotate an association end with *{bag}*.

base class (in C++) a superclass.

batch transformation (architectural style) a sequential input-to-output transformation, in which inputs are supplied at the start and the goal is to compute an answer. There is no ongoing interaction with the outside world. (Contrast with *continuous transformation.*)

boundary class a class of objects that provide a staging area for communications between a system and an external source.

call-by-reference (in a programming language) a mechanism that passes arguments to a method by passing the address of each argument. (Contrast with *call-by-value.*)

call-by-value (in a programming language) a mechanism that passes arguments to a method by passing a copy of the data values. If an argument is modified, the new value will not take effect outside of the method that modifies it. (Contrast with *call-by-reference.*)

candidate key (in a relational database) a combination of columns that uniquely identifies each row in a table. The combination must be minimal and include only those columns that are needed for unique identification. No column in a candidate key can be null.

cardinality the count of elements that are in a collection. (Contrast with *multiplicity.*)

change event an event that is caused by the satisfaction of a boolean expression. The intent of a change event is that the expression is continually tested—whenever the expression changes from false to true the event happens. The UML notation is the keyword *when* followed by a parenthesized boolean expression. (Contrast with *guard condition.*)

changeability an indication whether a property (such as an association end) can be modified after the initial value is created. The possibilities are *changeable* (can be updated) and *readonly* (can only be initialized).

class a description of a group of objects with similar properties (attributes), common behavior (operations and state diagrams), similar relationships to other objects, and common semantics. The UML notation is a box with the name in the top compartment.

class design the development stage for expanding and optimizing the analysis models so that they are amenable to implementation.

class diagram a graphic representation that describes classes and their relationships, thereby describing possible objects. (Contrast with *object diagram.*)

class model a description of the structure of the objects in a system including their identity, relationships to other objects, attributes, and operations.

classification a grouping of objects with the same data structure and behavior.

client a subsystem that requests services from another subsystem. (Contrast with *server.*)

coherence a property of an element, such as a class, an operation, or a package, such that it is organized on a consistent plan and all its parts fit together toward a common goal.

completion transition a transition that automatically fires when the activity associated with the source state is completed.

composite state a state that provides shared behavior for nested states. (Contrast with *nested state.*)

composition a form of aggregation with two additional constraints. A part can belong to at most one assembly. Furthermore, once a part has been assigned an assembly, it has a coincident lifetime with the assembly. The UML notation is a small *solid* diamond superimposed on the association end next to the assembly class. (Contrast with *aggregation.*)

concrete class a class that can have direct instances. (Contrast with *abstract class.*)

concurrent two or more activities or events whose execution may overlap in time.

condition (see *guard condition*).

constraint a boolean condition involving model elements such as objects, classes, attributes, associations, and generalization sets. The UML notation for simple constraints is a text string enclosed in braces or placed in a "dog-eared" comment box. For complex constraints, you can use the Object Constraint Language.

constructor (in C++ and Java) an operation that initializes a newly created instance of a class. (Contrast with *destructor*.)

container class a class of *container objects*. Examples include sets, arrays, dictionaries, and associations.

container object an object that stores a collection of other objects and provides various operations to access or iterate over its contents.

continuous transformation (architectural style) a system in which the outputs actively depend on changing inputs and must be periodically updated. (Contrast with *batch transformation*.)

control the aspect of a system that describes the sequences of operations that occur in response to stimuli.

controller an active object that manages control within an application.

database a permanent, self-descriptive store of data that is contained in one or more files. Self-description is what sets a database apart from ordinary files.

database management system the software for managing access to a database.

data dictionary the definition of all modeling elements (classes, associations, attributes, operations, and enumeration values) and an explanation of the rationale for key modeling decisions.

DBMS (acronym) *database management system*.

default value the value used to initialize an attribute or method argument.

delegation an implementation mechanism in which an object, responding to an operation on itself, forwards the operation to another object.

denormalization the violation of normal forms. Developers should violate normal forms only for good cause, such as to increase performance for a bottleneck. (See *normal form*.)

derived class (in C++) a subclass.

derived element (in UML) an element that is defined in terms of other elements. Classes, attributes, and associations can all be derived. Do not confuse the UML term *derived* with the C++ *derived class*. A C++ derived class refers to the subclass of a generalization and has nothing to do with UML's meaning of derived element. The UML notation is a slash preceding the element name.

descendant class a class that is a direct or indirect subclass of a given class. (Contrast with *ancestor class*.)

destructor (in C++) an operation that cleans up an existing instance of a class that is no longer needed. (Contrast with *constructor*.)

development the construction of software.

development life cycle an approach for managing the process of building software.

development stage a step in the process of building software. This book covers the following sequence of development stages: system conception, domain analysis, application analysis, system design, class design, implementation modeling, and implementation. Even though the development stages are ordered, all portions of an application need not proceed in tandem. We do not mean to imply waterfall development.

dictionary an unordered collection of object pairs with duplicates allowed. Each pair binds a key to an element. You can then use the key to look up the element.

direction whether an argument to an operation/method is an input (*in*), output (*out*), or an input argument that can be modified (*inout*).

do-activity an activity that continues for an extended time. The UML notation is *"do /"* followed by the do-activity name.

domain analysis the first substage of analysis that focuses on modeling real-world things that carry the semantics of an application.

dynamic binding a form of method resolution that associates a method with an operation at run time, depending on the class of one or more target objects.

dynamic simulation (architectural style) a system that models or tracks objects in the real world.

effect a reference to a behavior that is executed in response to an event. The UML notation for an effect is a slash ("/") followed by the activity name.

encapsulation the separation of external specification from internal implementation. (Synonymous with *information hiding*.)

enterprise model a model that describes an entire organization or some major aspect of an organization.

Entity-Relationship (ER) model a graphical approach to modeling originated by Peter Chen that shows entities and the relationships between them. The UML class model is based on the ER model.

entry activity an activity that is executed upon entry to a state. The UML notation is to list an entry activity within a state preceded by *"entry /"*. (Contrast with *exit* activity.)

enumeration a data type that has a finite set of values. The UML notation is the keyword *«enumeration»* above the enumeration name in the top section of a box. The second section lists the enumeration values.

ER (acronym) *Entity-Relationship model*.

event an occurrence at a point in time. (Contrast with *state*.)

event-driven control an approach in which control resides within a dispatcher or monitor that the language, subsystem, or operating system provides. Developers attach application methods to events, and the dispatcher calls the methods when the corresponding events occur ("callback"). (Contrast with *procedure-driven control.*)

exit activity an activity that is executed just before exit from a state. The UML notation is to list an exit activity within a state preceded by *"exit /"*. (Contrast with *entry* activity.)

extend (use case relationship) a relationship that adds incremental behavior to a use case. Note that the extension adds itself to the base; in contrast, for an *include* relationship the base explicitly incorporates the inclusion. The UML notation is a dashed arrow from the extension use case to the base use case. The keyword *«extend»* annotates the arrow. (Contrast with *include.*)

extensibility a property of software such that new kinds of objects or functionality can be added to it with little or no modification to existing code.

extent (of a class) the set of objects for a class.

feature an attribute or an operation.

final (for a Java class) a directive that prevents further subclassing.

final (for a Java method) a directive that prevents the method from being overridden.

fire to cause a transition to occur.

focus of control the period of time for an object's execution. The UML notation is a thin rectangle. (Synonymous with *activation.*)

foreign key (in a relational database) a reference to a candidate key (normally a reference to a primary key). It is the glue that binds tables.

forward engineering the building of an application from general requirements through to an eventual implementation. (Contrast with *reverse engineering.*)

fourth-generation language a framework for straightforward database applications that provides screen layout, simple calculations, and reports.

framework a skeletal structure of a program that must be elaborated to build a complete application.

friend (in C++) a declaration that permits selective access to members. The class containing the *friend* declaration grants access to a named function, method, or class.

garbage collection (in a programming language) a mechanism for automatically deallocating data structures that can no longer be accessed and are therefore not needed.

generalization an organization of elements (such as classes, signals, or use cases) by their similarities and differences. The UML notation is a triangle with the apex next to the superelement. (Contrast with *specialization.*)

generalization set name an enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization.

guard condition a boolean expression that must be true in order for a transition to occur. A guard condition is checked only once, at the time the event occurs, and the transition fires if the condition is true. The UML notation is to list a guard condition in square brackets after an event.

identifier one or more attributes in an implementation that unambiguously differentiate an object from all others.

identity the inherent property of an object which distinguishes each object from all others.

implementation the development stage for translating a design into programming code and database structures.

implementation inheritance an abuse of inheritance that seeks to reuse existing code, but does so with an illogical application structure that can compromise future maintenance.

implementation method (style) a method that implements specific computations on fully specified arguments, but does not make context-dependent decisions. (Contrast with *policy method.*)

implementation modeling the development stage for adding fine details to a model that transcend languages. Implementation modeling is the immediate precursor to the actual implementation.

include (use case relationship) a relationship that incorporates one use case within the behavior sequence of another use case. The UML notation is a dashed arrow from the source (including) use case to the target (included) use case. The keyword *«include»* annotates the arrow. (Contrast with *exclude.*)

index a data structure that maps one or more attribute values into the objects or database table rows that hold the values. Indexes are used for optimization (to quickly locate objects and table rows) and to enforce uniqueness.

information hiding (see *encapsulation*)

inheritance the mechanism that implements the generalization relationship.

integration testing testing of code from multiple developers to determine how the classes and methods fit together. (Contrast with *unit testing* and *system testing.*)

interaction model the model that describes how objects collaborate to achieve results. It is a holistic view of behavior across many objects, whereas the state model is a reductionist view of behavior that examines each object individually.

interactive interface (architectural style) a system that is dominated by interactions between the system and agents, such as humans, devices, or other programs.

interface (in Java) an uninstantiable class specification the contains only constants and method declarations.

iterative development the development of a system by a process broken into a series of steps, or iterations, each of which provides a better approximation to the desired system than the previous iteration. (Contrast with *rapid prototyping* and *waterfall development.*)

iterator (in a programming language) a construct that controls iteration over a range of values or a collection of objects.

layer a subsystem that provides multiple services, all of which are at the same level of abstraction. A layer can be built on subsystems at a lower level of abstraction. (Contrast with *partition.*)

leaf class a class with no subclasses. It must be a concrete class. In Java, this is the same as a final class.

library a collection of classes that are reusable across applications.

life cycle (see *development life cycle*).

lifeline the period of time during which an object exists.

link a physical or conceptual connection among objects. A link is an instance of an association. The UML notation is a line between objects that may consist of several line segments.

lock a logical object associated with some defined subset of a resource that gives the lock holder the right to access the resource directly.

member (in C++) data or methods of a class.

metaclass a class describing other classes.

metadata data that describes other data.

method the implementation of an operation for a class. The UML notation lists methods in the third compartment of the class box. (Contrast with *operation.*)

method caching (in a programming language) an optimization of method searching in which the address of a method is found the first time an operation is applied to an object of a class and then stored in a table attached to the class.

method resolution (in a programming language) the process of matching an operation on an object to the method appropriate to the object's class.

methodology (in software engineering) a process for the organized production of software using a collection of predefined techniques and notational conventions.

model an abstraction of some aspect of a problem. We express models with various kinds of diagrams.

modularity the organization of a system into groups of closely related objects.

multiple inheritance a type of inheritance that permits a class to have more than one superclass and to inherit features from all ancestors. (Contrast with *single inheritance.*)

multiplicity (of an association end) the number of instances of one class that may relate to a single instance of an associated class. Multiplicity is a constraint on the size of a collection. The UML notation is a numeric interval or the special symbol "*" denoting "many" (zero or more). (Contrast with *cardinality.*)

multiplicity (of an attribute) the possible number of values for each instantiation of an attribute. The most common specifications ar a mandatory single value [1], an optional single value [0..1], and many [*].

namespace (in C++) a means for providing a semantic scope for symbols to alleviate name conflicts.

***n*-ary association** an association involving three or more association ends. The UML symbol is a diamond with lines connecting to the related classes. If the association has a name, it is written in italics next to the diamond.

navigability the direction of traversal of a binary association in an implementation. The possibilities are none, either direction, or both directions. The UML shows navigability with an arrowhead on the association end attached to the target class.

navigation a traversal of associations and generalizations in a class model to go from source objects to target objects.

nested state a state that shares behavior from its composite state and adds additional behavior of its own. (Contrast with *composite state.*)

new (in C++ and Java) the operator to create objects.

normal form (in a relational database) a guideline for relational database design that increases data consistency.

***n*-tier architecture** an extension of the three-tier architecture, permitting any number of application layers. (Contrast with *three-tier architecture.*)

null a special value denoting that an attribute value is unknown or not applicable.

object a concept, abstraction, or thing that can be individually identified and has meaning for an application. An object is an instance of a class.

Object Constraint Language (OCL) a language for defining constraints that is part of the UML. You can also use the OCL to navigate class models.

object diagram a graphical representation that shows individual objects and their relationships. (Contrast with *class diagram.*)

object identity (in a relational database) the use of an artificial number to identify each record in a table. (Contrast with *value-based identity.*)

Object Management Group (OMG) a standards forum that is the owner of the UML.

object-orientation (OO) a strategy for organizing systems as collections of interacting objects that combine data and behavior.

OCL (acronym) *Object Constraint Language.*

OMG (acronym) *Object Management Group.*

OO (acronym) *object-oriented.*

OO database a database that is perceived as objects that mix data and behavior. (Contrast with relational database.)

OO-DBMS a DBMS that provides persistent objects in addition to the transient objects provided by OO programming languages. (Contrast with relational DBMS.)

OO development a software development technique that uses objects as a basis for the construction of software.

OO programming language a language that supports objects (combining identity, data, and operations), method resolution, and inheritance.

operation a function or procedure that may be applied to or by objects in a class. (Contrast with *method.*)

ordered a sorted collection of elements with no duplicates allowed. The UML notation is to annotate an association end with *{ordered}*. (Contrast with *sequence.*)

origin class the topmost class in an inheritance hierarchy that defines a feature.

overloading (in a programming language) binding the same name to multiple methods whose signatures differ in number or types of arguments. A call to an overloaded operation is resolved at compile time based on the types of the calling arguments.

override to define a method for an operation that replaces an inherited method for the same operation.

package (class modeling construct) a group of elements (classes, associations, generalizations, and lesser packages) with a common theme. The UML notation is a box with a tab; the package name is placed in the box.

package (referring to visibility) accessible by methods of classes in the same package as the containing class.

partition a subsystem that provides a particular kind of service in parallel to other subsystems. A partition may itself be built from lower-level subsystems. (Contrast with *layer.*)

passive object an object that does not have its own thread of control. (Contrast with *active object.*)

pattern a parameterized excerpt of a model that is important and recurring. It is mathematical in nature and worthy of reuse across multiple applications.

peer two or more subsystems that are mutually interdependent for services. (Contrast with *client* and *server.*)

persistent object an object that is stored in a database and can span multiple application executions. (Contrast with *transient object.*)

policy method (style) a method that makes context-dependent decisions but calls on implementation methods for detailed computations. (Contrast with *implementation method.*)

polymorphism takes on many forms; the property that an operation may behave differently on different classes.

primary key (in a relational database) a candidate key that is preferentially used to access the records in a table. A table can have at most one primary key; normally each table should have a primary key.

private (referring to visibility) accessible by methods of the containing class only.

procedure-driven control an approach in which control resides within the program code. Procedures request external input and then wait for it; when input arrives, control resumes within the procedure that made the call. The location of the program counter and the stack of procedure calls and local variables define the system state. (Contrast with *event-driven control.*)

programming-in-the-large the creation of large, complex programs with teams of programmers.

protected (referring to C++ visibility) accessible by methods of the containing class and any of its descendant classes.

protected (referring to Java visibility) accessible by methods of the containing class, any of its descendant classes, and classes in the same package as the containing class.

public (referring to visibility) accessible by methods of any class.

qualified association an association in which one or more attributes (called qualifiers) disambiguate the objects for a "many" association end. The UML notation is a small box on the end of the association line near the source class.

qualifier an attribute that distinguishes among the objects at a "many" association end. The UML notation is to place a qualifier in a small box on the end of the association line near the source class.

race condition a situation in which the order of receiving concurrent signals can affect the final state of an object.

rapid prototyping the quick development of a portion of a system for experimentation and evaluation. Prototyping is proof of concept and often throwaway by intent. (Contrast with *iterative development* and *waterfall development.*)

real-time system (architectural style) an interactive system for which time constraints on actions are particularly tight or in which the slightest timing failure cannot be tolerated.

refactoring changes to the internal structure of software to improve its design without altering its external functionality.

reference an attribute value in one object that refers to another object.

reflection a property of a system such that it can examine its own structure dynamically and reason about its own state.

region a portion of a state diagram.

reification the promotion of something that is not an object into an object.

relational database a database in which the data are perceived as tables. (Contrast with OO database.)

relational DBMS a DBMS that manages tables of data and associated structures that increase the functionality and performance of tables. (Contrast with OO-DBMS.)

responsibility something that an object knows or something it must do. A responsibility is not a precise concept; it is meant to get the thought process going.

reverse engineering the process of examining implementation artifacts and inferring the underlying logical intent. (Contrast with *forward engineering*.)

robust a property of software such that it does not fail catastrophically when some of its design assumptions are violated.

scenario a sequence of events that occur during one particular execution of a system.

schema the structure of the data in a database.

scope an indication if a feature applies to an object or a class. An underline distinguishes features with class scope (static) from those with object scope.

sequence a sorted collection of elements with duplicates allowed. The UML notation is to annotate an association end with *{sequence}*. (Contrast with *ordered*.)

sequence diagram a diagram that shows the participants in an interaction and the sequence of messages among them.

server a subsystem that provides a service to other subsystems. (Contrast with *client*.)

service a group of related functions or operations that share some common purpose.

shopping-list operation an operation that is meaningful in its own right. Bertrand Meyer coined the term *shopping list* because discovery of such an operation is driven by the intrinsic meaning of a class and not by the needs of a particular application. Sometimes the real-world behavior of classes suggests operations.

signal an explicit one-way transmission of information from one object to another. The UML notation is the keyword *«signal»* above the signal class name in the top section of a box. The second section lists the signal attributes.

signal event the event of sending or receiving a signal.

signature the number and types of the arguments for an operation and the type of its result.

single inheritance a type of inheritance in which a class may have only a single superclass. (Contrast with *multiple inheritance*.)

software engineering a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software.

specialization the refinement of a class into variants. Specialization has the same meaning as generalization but takes a top-down perspective. In contrast, generalization takes a bottom-up perspective. (Contrast with *generalization*.)

SQL the standard language for interacting with a relational database.

state an abstraction of the values and links of an object. The UML notation is a rounded box containing an optional state name. (Contrast with *event*.)

state diagram a graph whose nodes are states and whose directed arcs are transitions between states.

state model a description of those aspects of a system concerned with time and the sequencing of operations. The state model consists of multiple state diagrams, one state diagram for each class with important temporal behavior.

static (in C++ and Java) data and methods that belong not to an instance of a class, but to the class itself.

stored procedure (in a relational database) a method that is stored in a database.

strong typing (in a programming language) a requirement that the type of each variable must be declared. (Contrast with *weak typing*.)

subclass a class that adds specific attributes, operations, state diagrams, and associations for a generalization. (Contrast with *superclass*.)

submachine a state diagram that may be invoked as part of another state diagram. The UML notation for invoking a submachine is to list a local state name followed by a colon and the submachine name.

substate a state that expresses an aspect of concurrent behavior for a parent state.

subsystem a major piece of a system that is organized around some coherent theme. A system may be divided into subsystems using either *partitions* or *layers*.

superclass the class that holds common attributes, operations, state diagrams, and associations for a generalization. (Contrast with *subclass*.)

swimlane a column in an activity diagram that shows the person or organization who performs activities; a partition.

system an application that is the subject of interest.

system architecture (see *architecture*).

system boundary the outline of the scope of a system determining what the system includes and what the system omits.

system conception the development stage that deals with the genesis of an application.

system design the development stage during which the developer devises the architecture and establishes general design policies.

system testing the checking of an entire application. (Contrast with *unit testing* and *integration testing*.)

table (in a relational database) an organization of data that has a specific number of columns and an arbitrary number of rows.

ternary association an association among three association ends. The UML symbol is a diamond with lines connecting to the related classes. If the association has a name, it is written in italics next to the diamond.

this (in C++ and Java) the default name of the target object of a method.

thread of control a single path of execution through a program, a state model, or some other representation of control flow.

three-tier architecture an approach that separates data management, application functionality, and the user interface. The data management layer holds the database schema and data. The application layer holds the methods that embody the application logic. The user-interface layer manages the forms and reports that are presented to the user. (Contrast with *n-tier architecture*.)

time event an event caused by the occurrence of an absolute time or the elapse of a time interval. The UML notation for an absolute time is the keyword *when* followed by a parenthesized expression involving time. The notation for a time interval is the keyword *after* followed by a parenthesized expression that evaluates to a time duration.

transaction manager (architectural style) a database system whose main function is to store and access information.

transient object an object that exists only in memory and disappears when an application terminates execution. Thus a transient object is an ordinary programming object. (Contrast with *persistent object*.)

transition an instantaneous change from one state to another. The UML notation is a line (with possibly multiple line segments) from the origin state to the target state; an arrowhead points to the target state.

transitive closure (from graph theory) the set of nodes that are reachable by some sequence of edges.

UML (acronym, trademark of the OMG) *Unified Modeling Language*.

Unified Modeling Language (trademark of the OMG) a comprehensive suite of object-oriented models intended to represent software and other kinds of applications fully. The UML has been developed under the auspices of the OMG.

UML1 informal term for the first release of the UML approved in 1997.

UML2 informal term for the second release of the UML approved in 2004. This book is based on UML2.

unit testing testing by developers of their own code for classes and methods. (Contrast with *integration testing* and *system testing*.)

use case a coherent piece of functionality that a system can provide by interacting with actors. The UML notation is an ellipse with the use case name inside.

use case diagram a graphical notation for summarizing actors and use cases.

user interface an object or group of objects that provides the user of a system with a coherent way to access its domain objects, commands, and application options.

value a piece of data. A value is an instance of an attribute.

value-based identity (in a relational database) the use of some combination of real-world attributes to identify each record in a table. (Contrast with *existence-based identity*.)

view (in a relational database) a table that a relational DBMS dynamically computes.

virtual (in C++) an operation that can be overridden by a descendant class.

visibility the ability of a method to reference a feature from another class. The UML denotes visibility with the following prefixes. The possibilities are *public* ("+"), *protected* ("#"), *private* ("-"), and *package* ("~").

waterfall development the development life cycle of performing the software development stages in a rigid linear sequence with no backtracking. (Contrast with *iterative development* and *rapid prototyping*.)

weak typing (in a programming language) the lack of a requirement that the type of each variable be declared. (Contrast with *strong typing*.)

wrapper a collection of interfaces that allow access into a system.

Answers to Selected Exercises

We selected answers with the following criteria: exercises with short answers in the core chapters, exercises that extend chapters by introducing new material, key exercises in a series of questions, answers that clarify subtle or difficult points, and prototypes for real problems. Most exercises have multiple correct answers, so use our answers only as a guide and not as a test of correctness.

1.5b. Criminal investigations can use combinations of photographs, fingerprinting, blood-typing, DNA analysis, and dental records to identify people, living and/or deceased, who are involved in, or the subject of, a criminal investigation.

d. Telephone numbers are adequate for identifying almost any telephone in the world. In general a telephone number consists of a country code plus a province, city, or area code, plus a local number plus an optional extension number. Businesses may have their own telephone systems with other conventions. Depending on the relative location of the telephone that you are calling, parts of the number may be implied and can be left out, but extra access digits may be required to call outside the local region.

In North America most local calls require 7 digits. Long distance calls in North America use an access digit (0 or 1) + area code (3 digits) + local number (7 digits). Dialing Paris requires an access code (011) + country code (33) + city code (1) + local number (8 digits). The access code is not part of the identifier.

g. One way that employees are given restricted, after-hours access to a company is through the use of a special, electronically readable card. Of course, if an employee loses a card and does not report it, someone who finds it could use it for unauthorized entry. Other approaches include a picture ID which requires inspection by a guard, fingerprint readers, and voice recognition.

1.8a. Electron microscopes, eyeglasses, telescopes, bomb sights, and binoculars are all devices that enhance vision in some way. With the exception of the scanning electron microscope, all these devices work by reflecting or refracting light. Eyeglasses and binoculars are designed for use with two eyes; the rest of the objects on the list are designed for use with one eye. Telescopes, bomb sights, and binoculars are used to view things far away. A microscope is used to magnify something that is very small. Eyeglasses may enlarge or reduce, depending on whether the pre-

scription is for a nearsighted or a farsighted person. Some other classes that could be included in this list are optical microscopes, cameras, and magnifying glasses.

b. Pipes, check valves, faucets, filters, and pressure gauges are all plumbing supplies with certain temperature and pressure ratings. Compatibility with various types of fluids is also a consideration. Check valves and faucets may be used to control flow. With the exception of the pressure gauge, all of the items listed have two ends and have a pressure-flow characteristic for a given fluid. All of the items are passive. Some other classes include pumps, tanks, and connectors.

2.3a. For a transatlantic cable, resistance to salt water is the main consideration. The cable must lie unmaintained at the bottom of the ocean for a long time. Interaction of ocean life with the cable and the effect of pressure and salinity on cable life must be considered. The ratio of strength/weight is important to avoid breakage while the cable is being installed. Cost is an important economic factor. Electrical parameters are important for power consumption and signal distortion.

c. Weight is very important for wire that is to be used in the electrical system of an airplane, because it affects the total weight of the plane. Toughness of the insulation is important to resist chafing due to vibration. Resistance of the insulation to fire is also important to avoid starting or feeding electrical fires in flight.

3.2 Figure A3.2 shows a class diagram for polygons and points. The smallest number of points required to construct a polygon is three.

The multiplicity of the association depends on how points are identified. If a point is identified by its location, then points are shared and the association is many-to-many. On the other hand, if each point belongs to exactly one polygon, then several points may have the same coordinates. The next answer clarifies this distinction.

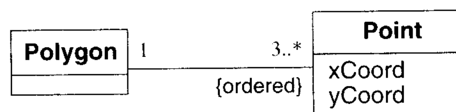


Figure A3.2 Class diagram for polygon and points

3.3a. Figure A3.3 shows objects and links for two triangles with a common side in which a point belongs to exactly one polygon.

b. Figure A3.4 shows objects and links for two triangles with a common side in which points may be shared.

3.20 Graphs occur in many applications. Several variations of the model are possible, depending on your viewpoint. Figure A3.23 accurately represents undirected graphs as described in the exercise. Although not quite as accurate, your answer could omit the class *UndirectedGraph*.

We have found it useful for some graph related queries to elevate the association between vertices and edges to the status of a class as Figure A3.24 shows.

3.23 Figure A3.27 shows a class diagram describing directed graphs.The distinction between the two ends of an edge is accomplished with a qualified association. Values of the qualifier end are *from* and *to*.

Figure A3.28 shows another representation of directed graphs. The distinction between the two ends of an edge is accomplished with separate associations.

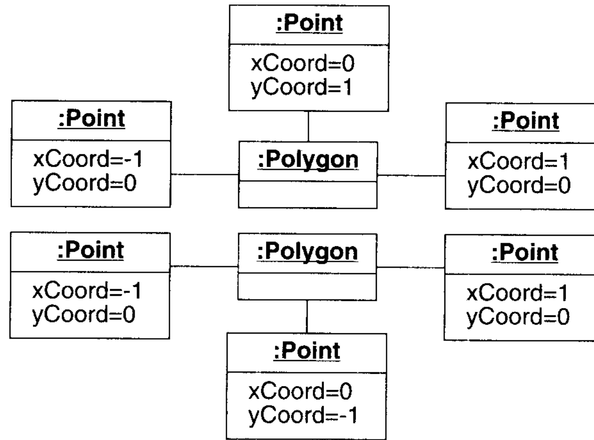


Figure A3.3 Object diagram where each point belongs to exactly one polygon

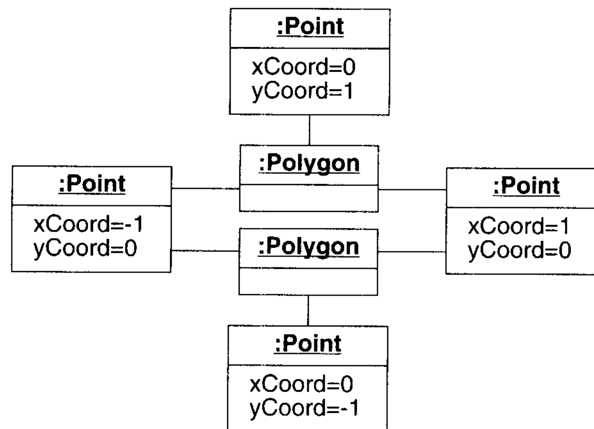


Figure A3.4 Object diagram where each point can belong to multiple polygons

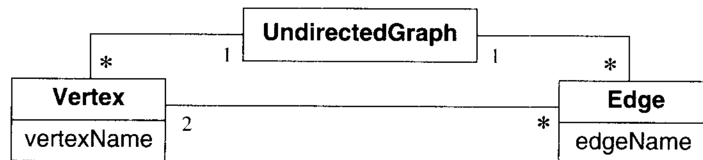


Figure A3.23 Class diagram for undirected graphs

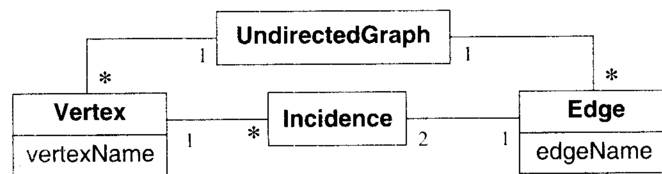


Figure A3.24 Class diagram for undirected graphs in which the incidence between vertices and edges is treated as a class

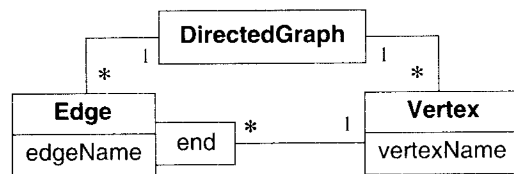


Figure A3.27 Class diagram for directed graphs using a qualified association

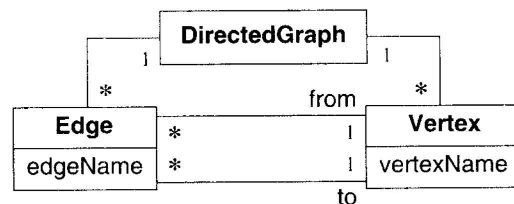


Figure A3.28 Class diagram for directed graphs using two associations

The advantage of the qualified association is that only one association must be queried to find one or both vertices that a given edge is connected to. If the qualifier is not specified, both vertices can be found. By specifying *from* or *to* for the *end* qualifier, you can find the vertex connected to an edge at the given *end*.

The advantage of using two separate associations is that you eliminate the need to manage enumerated values for the qualifier *end*.

3.25 Figure A3.30 shows a class diagram for car loans in which pointers are replaced with associations.

In this form, the arguably artificial restriction that a person have no more than three employers has been eliminated. Note that in this model an owner can own several cars. A car can have several loans against it. Banks loan money to persons, companies, and other banks.

3.28 Figure A3.34 shows a class diagram for the dining philosophers problem. The one-to-one associations describe the relative locations of philosophers and forks. The *InUse* association describes who is using forks. Other representations are possible, depending on your viewpoint. An object diagram may help you better understand this problem.

3.31 The following OCL expression computes the set of airlines that a person flew in a given year.

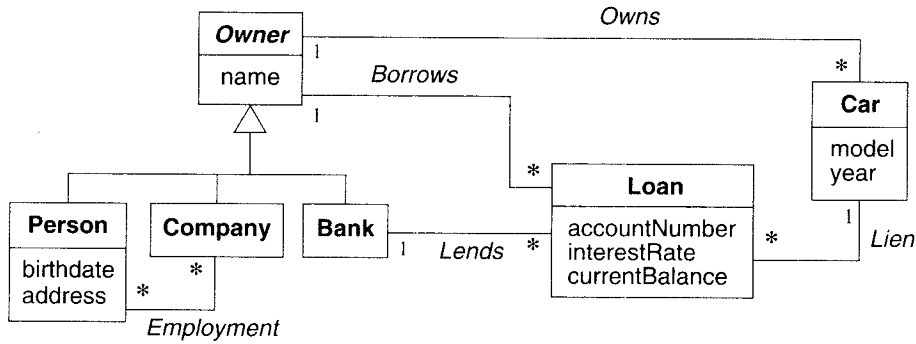


Figure A3.30 Proper class diagram for car loans

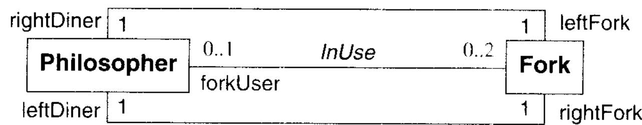


Figure A3.34 Class diagram for the dining philosopher problem

```
aPassenger.Flight->SELECT(getYear(date)=aGivenYear).
Airline.name->asSet
```

The OCL *asSet* operator eliminates redundant copies of the same airline.

3.34 Figure E3.13 (a) states that a subscription has derived identity. Figure E3.13 (b) gives subscriptions more prominence and promotes subscription to a class.

The (b) model is a better model. Most copies of magazines have subscription codes on their mailing labels; this could be stored as an attribute. The subscription code is intended to identify subscriptions; subscriptions are not identified by the combination of a person and a magazine, so we should promote *Subscription* to a class. Furthermore, a person might have multiple subscriptions to a magazine; only the (b) model can readily accommodate this.

4.2 The class diagram in Figure A4.2 generalizes the classes *Selection*, *Buffer*, and *Sheet* into the superclass *Collection*. This is a desirable revision. The generalization promotes code reuse, because many operations apply equally well to the subclasses. Six aggregation relationships in the original diagram, which shared similar characteristics, have been reduced to two. Finally, the structure of the diagram now captures the constraint that each *Box* and *Line* should belong to exactly one *Buffer*, *Selection*, or *Sheet*.

4.4 Figure A4.3 shows a class diagram for a graphical document editor. The requirement that a *Group* contain 2 or more *DrawingObjects* is expressed as a multiplicity of 2..* on *DrawingObject* in its aggregation with *Group*. The fact that a *DrawingObject* need not be in a *Group* is expressed by the zero-one multiplicity.

It is possible to revise this diagram to make a *Circle* a special case of an *Ellipse* and to make a *Square* a special case of a *Rectangle*.

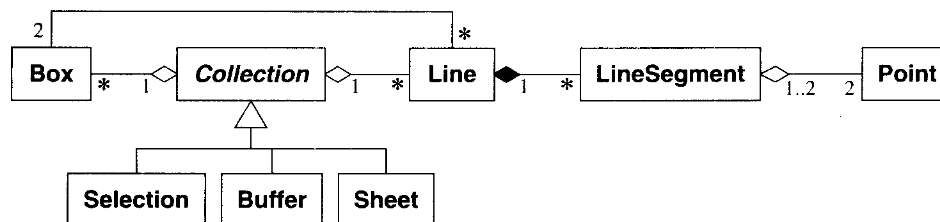


Figure A4.2 Generalization of the classes *Selection*, *Buffer*, and *Sheet* into the class *Collection*

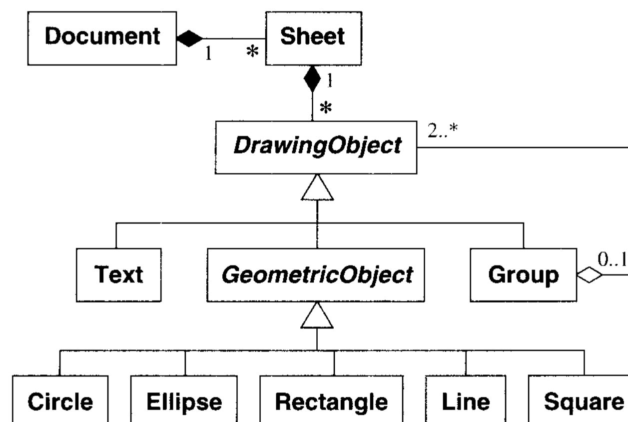


Figure A4.3 Class diagram for a graphical document editor that supports grouping

We presume that a *DrawingObject* belongs to a *Sheet* and has a coincident lifetime with it. Similarly, we presume that a *Sheet* belongs to one *Document* for its lifetime. Hence both are composition relationships.

4.5 Figure A4.4 shows a class diagram with several classes of electrical machines. We have included attributes that were not requested.

4.6 Figure A4.5 converts the overlapping combination of classes into a class of its own to eliminate multiple inheritance.

4.7 Figure A4.6 is a metamodel of the following UML concepts: class, attribute, association, association end, multiplicity, class name, and attribute name.

4.10 The class diagram in Figure E4.3 does support multiple inheritance. A class may have multiple generalization roles of subclass participating in a variety of generalizations.

4.11 To find the superclass of a generalization using Figure E4.3, first query the association between *Generalization* and *GeneralizationRole* to get a set of all roles of the given instance of *Generalization*. Then sequentially search this set of instances of *GeneralizationRole* to find the one with *roleType* equal to superclass. (Hopefully only one instance will be found with *role-*

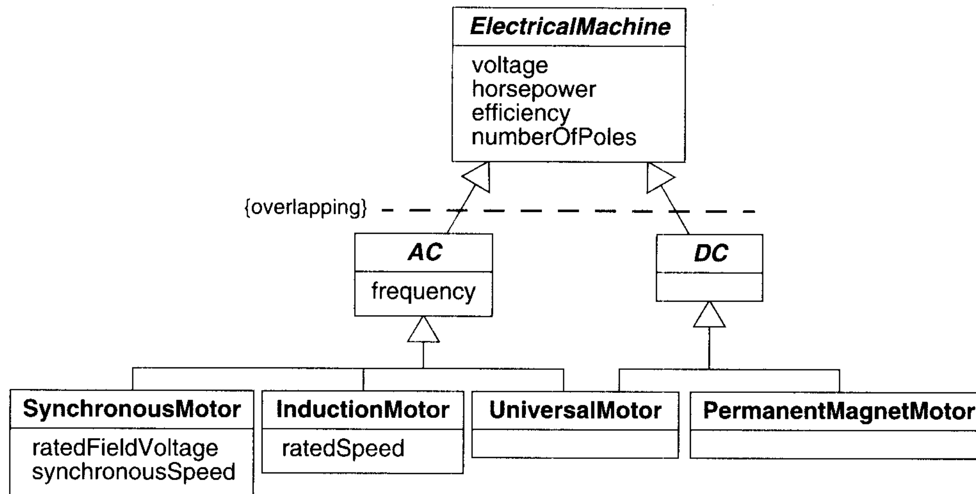


Figure A4.4 Partial taxonomy for electrical machines

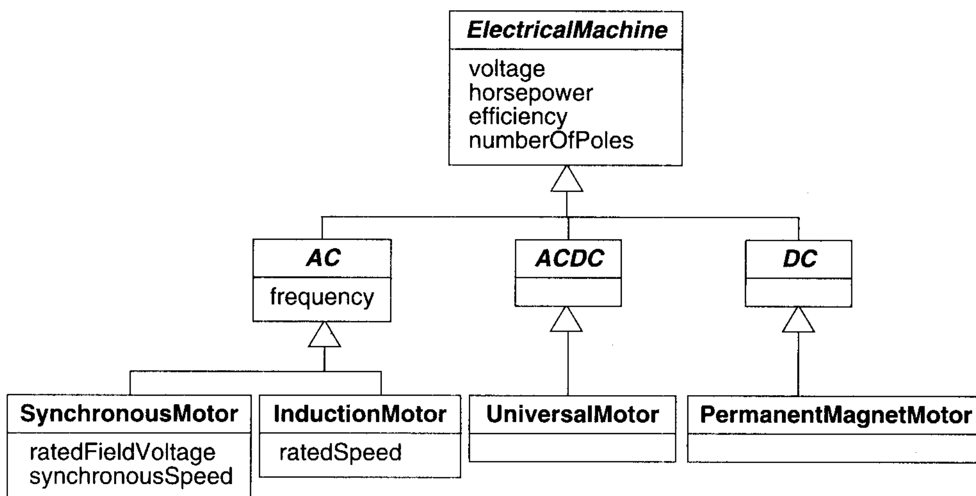


Figure A4.5 Elimination of multiple inheritance

Type equal to superclass, which is a constraint that the model does not enforce.) Finally, scan the association between *GeneralizationRole* and *Class* to get the superclass.

Figure A4.9 shows one possible revision which simplifies superclass lookup. To find the superclass of a generalization, first query the association between *Generalization* and *SuperclassRole*. Then query the association between *SuperclassRole* and *Class* to find the corresponding instance of *Class*.

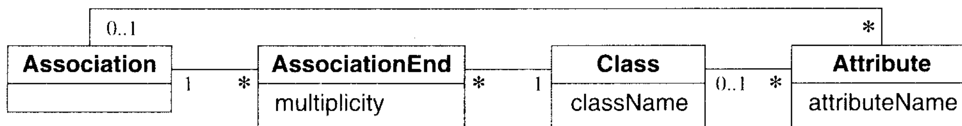


Figure A4.6 Metamodel for some UML concepts

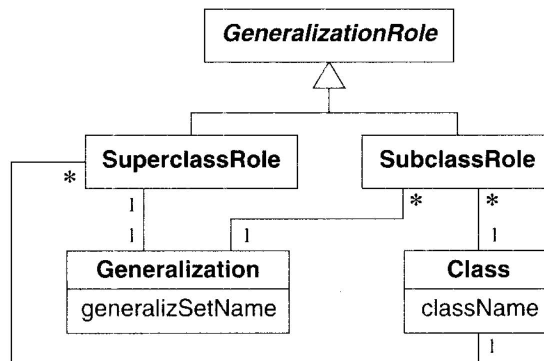


Figure A4.9 Metamodel of generalizations with separate subclass and superclass roles

Figure A4.10 shows another metamodel of generalization that supports multiple inheritance. To find the superclass of a generalization using this metamodel, simply query the *Superclass* association.

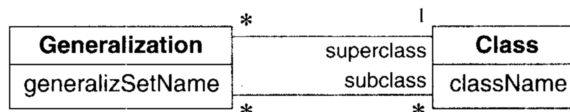


Figure A4.10 Simplified metamodel of generalization relationships

We do not imply that the metamodel in Figure A4.10 is the best model of generalization, only that it simplifies the query given in the exercise. The choice of which model is best depends on the purpose of the metamodel.

The following query finds the superclass, given a generalization for Figure E4.3.

- aGeneralization.GeneralizationRole->SELECT(roleType='superclass').Class

The following query finds the superclass, given a generalization for Figure A4.9.

- aGeneralization.SuperclassRole.Class

The following query finds the superclass, given a generalization for Figure A4.10.

- aGeneralization.superclass

4.16 The simple class model in Figure A4.14 is sufficient for describing the given recipe data.

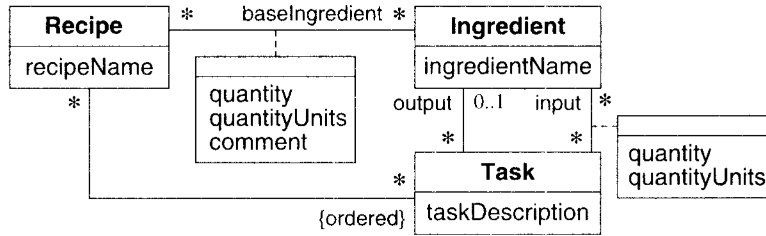


Figure A4.14 A simple class model for recipes

4.17 Figure A4.15 shows our initial solution to the exercise—merely adding an association that binds original ingredients to substitute ingredients. This model has two flaws.

The first problem is that the model awkwardly handles interchangeable ingredients. For example, in some recipes you can freely substitute butter, margarine, and shortening for each other. Figure A4.15 would require that we store each possible pair of ingredients. Thus we would have the following combinations of original and substitute ingredients—(butter, margarine), (butter, shortening), (margarine, butter), (margarine, shortening), (shortening, butter), and (shortening, margarine).

The second problem is that the substitutability of ingredients does not always hold, but can depend on the particular recipe.

Figure A4.16 shows a better class model that remedies both flaws.

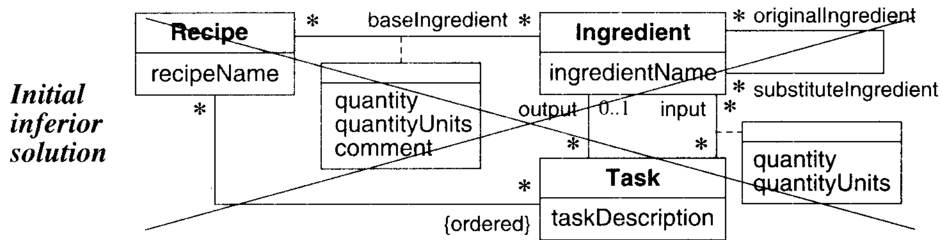


Figure A4.15 Initial class model for recipes with alternate ingredients

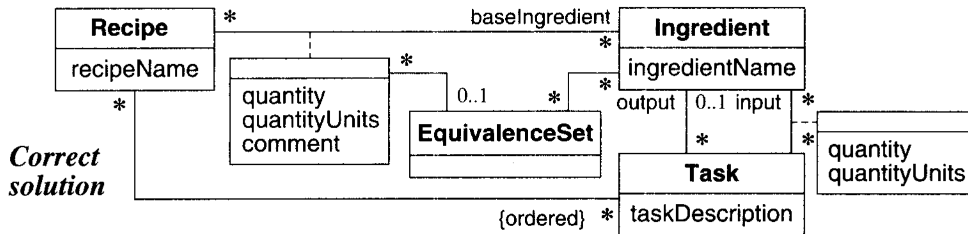


Figure A4.16 Correct class model for recipes with alternate ingredients

5.2 In Figure A5.2 the event *A* refers to pressing the *A* button. In this diagram, releasing the button is unimportant and is not shown (although you must obviously release the button before you can press it again). Note that a new button event cannot be generated while any button is pressed. You can consider this a constraint on the input events themselves and need not show it in the state diagram (although it would not be wrong to do so).

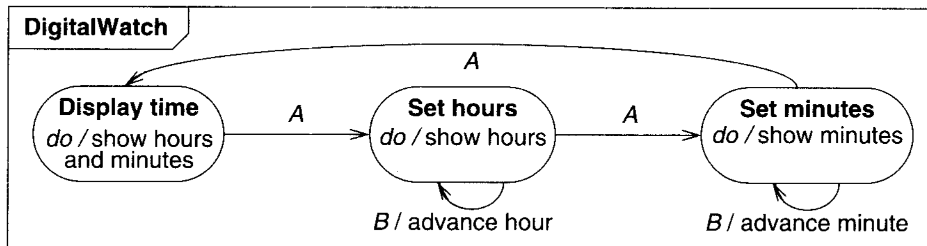


Figure A5.2 State diagram for a simple digital watch

5.6 Figure A5.6 shows the completed state diagram for the motor control.

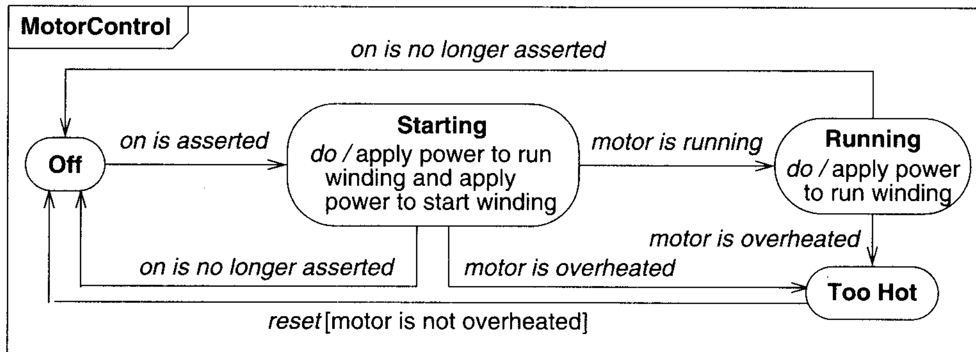


Figure A5.6 State diagram for a motor control

5.11 Figure A5.11 shows the state diagram. Note that even simple state diagrams can lead to complex behavior. A change event occurs whenever the candle is taken out of its holder or whenever it is put back. The condition at north is satisfied whenever the bookcase is behind the wall. The condition at north, east, south, or west is satisfied whenever the bookcase is facing front, back, or to the side.

When you first discovered the bookcase, it was in the *Stopped* state pointing south. When your friend removed the candle, a change event drove the bookcase into the *Rotating* state. When the bookcase was pointing north, the condition at north put the bookcase back into the *Stopped* state. When your friend reinserted the candle, another change event put the bookcase into the *Rotating* state until it again pointed north. Pulling the candle out generated another change event and would have caused the bookcase to rotate a full turn if you had not blocked it with your body. Forcing the bookcase back is outside the scope of the control and does not have to be explained.

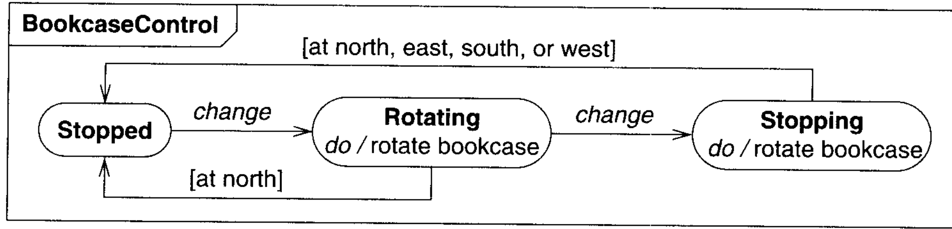


Figure A5.11 State diagram for bookcase control

When you put the candle back again, another change event was generated, putting the bookcase into the *Rotating* state once again. Taking the candle back out resulted in yet another change event, putting the bookcase into the *Stopping* state. After 1/4 turn, the condition at north, east, south or west was satisfied, putting the bookcase into the *Stopped* state.

What you should have done at first to gain entry was to take the candle out and quickly put it back before the bookcase completed 1/4 turn.

6.1 The headlight (Figure A6.1) and wheels (Figure A6.2) each have their own state diagram. Note that the stationary state for a wheel includes several substates.

We have shown default initial states for the headlight and wheels. The actual initial state of the wheels may be arbitrary and could be any one of the power off states. The system operates in a loop and does not depend on the initial state, so you need not specify it. Many hardware systems have indeterminate initial states.

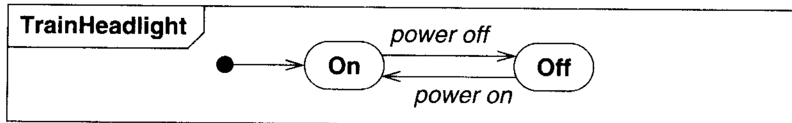


Figure A6.1 State diagram for a toy train headlight

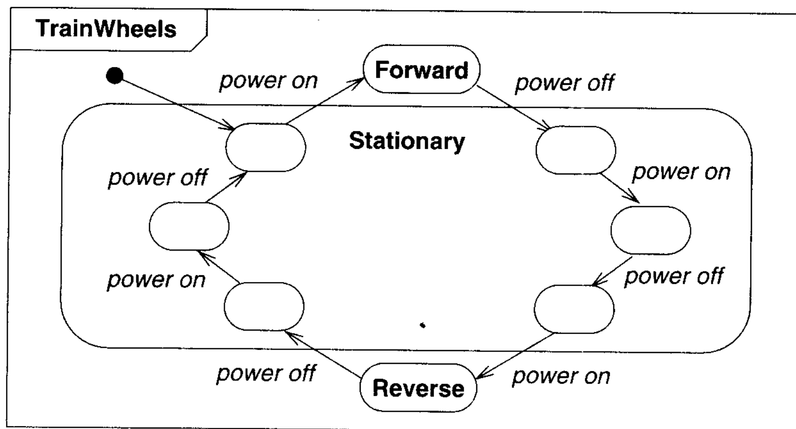


Figure A6.2 State diagram for the wheels of a toy train

6.3 Figure A6.4 adds *Motor On* to capture the commonality of the starting and running state. We have shown a transition from the *Off* state to the *Starting* state. We could instead have shown a transition from *Off* to *Motor On* and made *Starting* the initial state of *Motor On*. Note that the activity *apply power to run winding* has been factored out of both starting and running states.

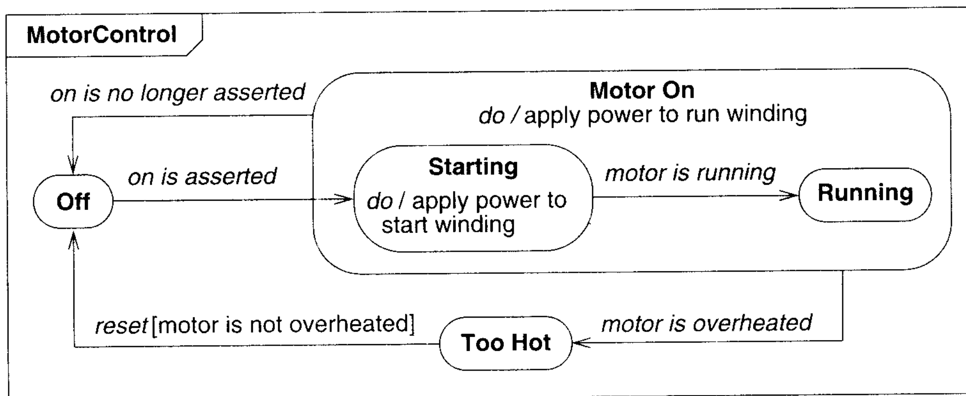


Figure A6.4 State diagram for a motor control using nested states

6.4 Figure A6.5 revises the motor state diagram. Note that a transition from *Off* to either *Forward* or *Reverse* also causes an implicit transition to *Starting*, the default initial state of the lower concurrent subdiagram. An off request causes a transition out of both concurrent subdiagrams back to state *Off*.

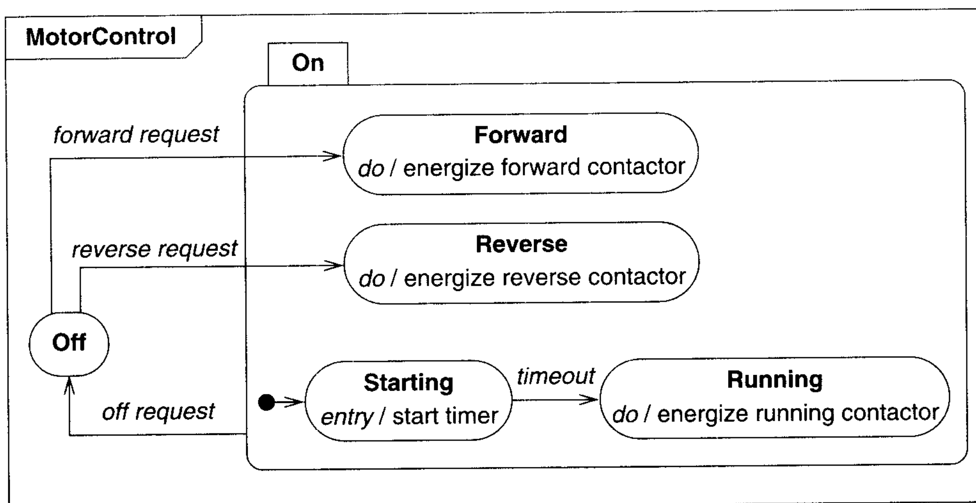


Figure A6.5 Revised state diagram for an induction motor control

7.1 Here are answers for a physical bookstore.

a. Some actors are:

- **Customer**. A person who initiates the purchase of an item.
- **Cashier**. An employee who is authorized to check out purchases at a cash register.
- **Payment verifier**. The remote system that approves use of a credit or debit card.

b. Some use cases are:

- **Purchase items**. A customer brings one or more items to the checkout register and pays for the items.
- **Return items**. The customer brings back items that were previously purchased and gets a refund.

c. Figure A7.1 shows a use case diagram.

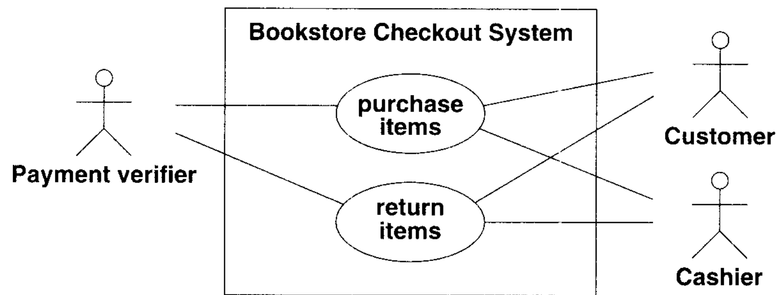


Figure A7.1 Use case diagram for a physical bookstore checkout system

d. Here is a normal scenario for each use case. There are many possible answers.

- **Purchase items**.
  Customer brings items to the counter.
  Cashier scans each customer item.
  Cashier totals order, including tax.
  Cashier requests form of payment.
  Customer gives a credit card.
  Cashier scans card.
  Verifier reports that credit card payment is acceptable.
  Customer signs credit card slip.
- **Return items**.
  Customer brings purchased item to the counter.
  Customer has receipt from earlier purchase.
  Cashier notes that payment was in cash.
  Cashier accepts items and gives customer a cash refund.

e. Here is an exception scenario for each use case. There are many possible answers.

- **Purchase items**.
  Customer brings items to the counter.
  Cashier scans each customer item.
  An item misscans and cashier goes to item display to get the item price.

- **Return items.**
  Customer brings purchased item to the counter.
  Customer has no receipt from earlier purchase.
  Customer is given a credit slip, but no refund.

f. Figure A7.2 shows a sequence diagram for the first scenario in (d). Figure A7.3 shows a sequence diagram for the second scenario in (d).

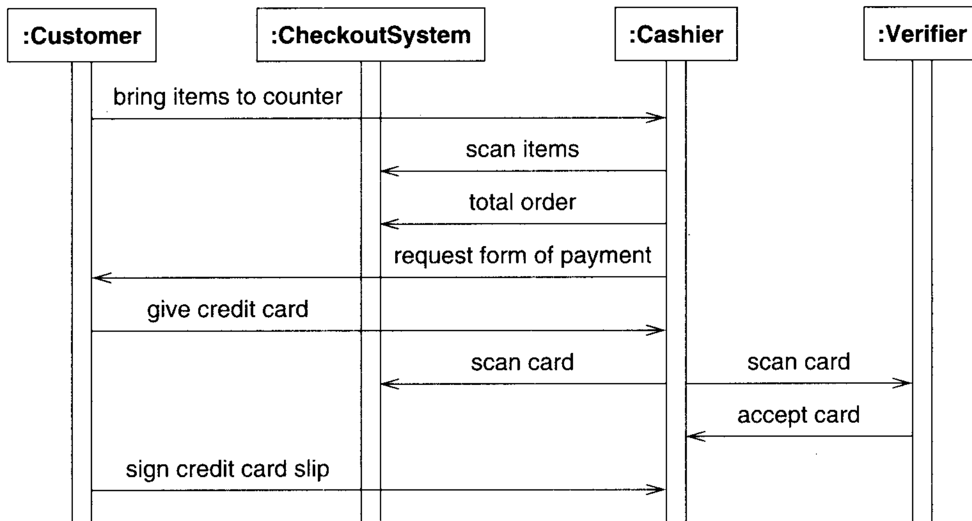


Figure A7.2 Sequence diagram for a purchase of items

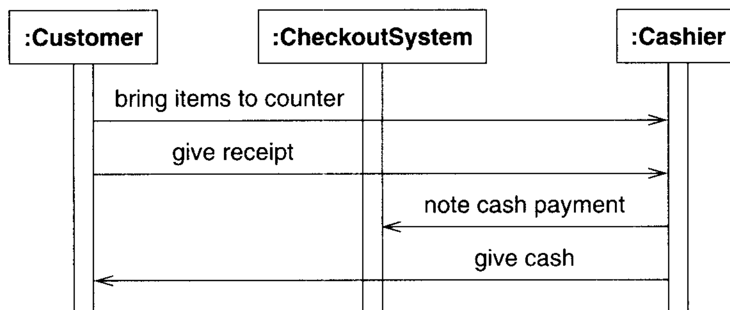


Figure A7.3 Sequence diagram for a return of items

7.8 Figure A7.12 shows an activity diagram for computing a restaurant bill.

8.1 Here are answers for an electronic gasoline pump.

a. Figure A8.1 shows a use case diagram.

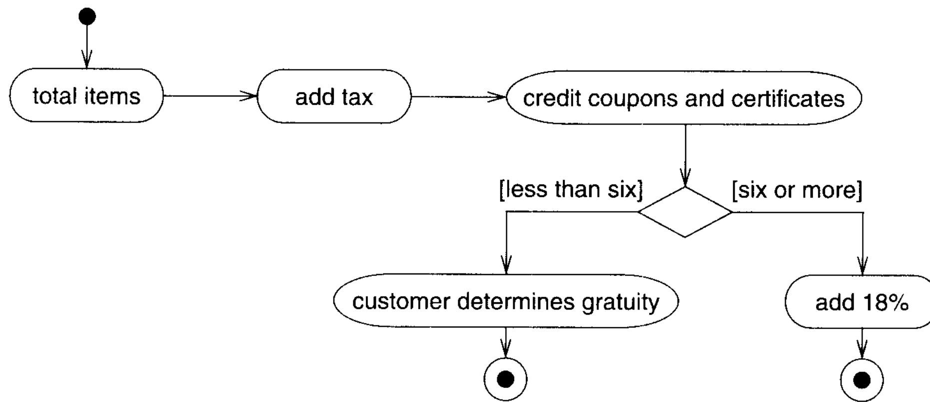


Figure A7.12 Activity diagram for computing a restaurant bill

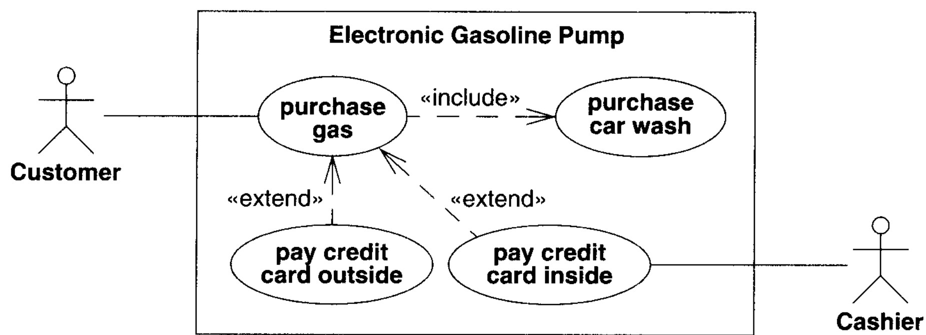


Figure A8.1 Use case diagram for an electronic gasoline pump

b. There are two actors:

- **Customer**. A person who initiates the purchase of gas.
- **Cashier**. A person who handles manual credit card payments and monitors the sale of gas.

c. There are four use cases:

- **Purchase gas**. Obtain gas from the electronic gas pump and pay for it with cash.
- **Purchase car wash**. A customer also decides to purchase a car wash and pays for it with cash.
- **Pay credit card outside**. Instead of cash, pay for the gas and optional car wash with a credit card that is directly handled by the gas system.
- **Pay credit card inside**. Instead of cash, pay for the gas and optional car wash with a credit card that is manually handled by the cashier.

8.6 Figure A8.6 computes the contents of a portfolio of stocks.

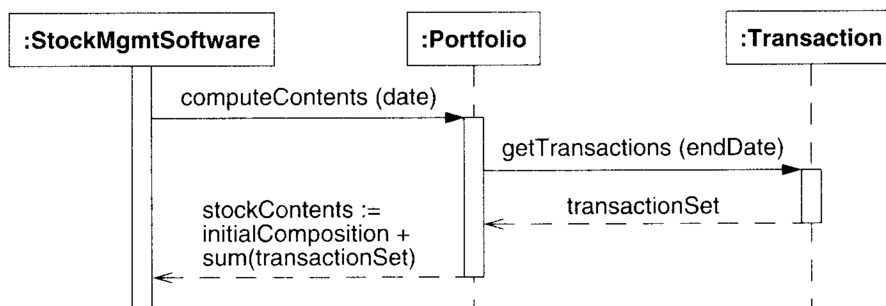


Figure A8.6 Sequence diagram for computing the contents of a portfolio of stocks

11.1 Here is elaboration for an antilock braking system for an automobile.

a. An antilock braking system could target the mass market. If the antilock system was inexpensive and safer than current technology, it could be government mandated and installed on all cars. (Further study would be needed to determine what price is "inexpensive" and what would be a "significant" safety improvement.)

There would be several stakeholders. Auto customers would expect improved safety and minimal detriment to drivability. Auto manufacturers would want to minimize the cost and quantify the benefit so they could tout the technology in their advertising. The government would be looking for a statistical safety improvement without compromising fuel efficiency.

If the new system was inexpensive, worked well, and did not hurt drivability, all car owners could be potential customers. An expensive antilock system could be a premium option on high-end cars.

b. Desirable features would include: effective prevention of brake locking, ability to detect excessive brake wear, and acquisition of data to facilitate auto maintenance. Some undesirable features would be: reduced fuel efficiency, reduced drivability, and greater maintenance complexity.

c. An antilock system must work with the brakes, steering, and automotive electronics.

d. There would be a risk that an antilock braking system could fail, leading to an accident and a lawsuit. Also it might be difficult to understand fully how the antilock system would interact with the brakes.

12.9 The following tentative classes should be eliminated.

- **Redundant classes**. *Child, Contestant, Individual, Person, Registrant* (all are redundant with *Competitor*).
- **Vague or irrelevant classes**. *Back, Card, Conclusion, Corner, IndividualPrize, Leg, Pool, Prize, TeamPrize, Try, WaterBallet.*
- **Attributes**. *address, age, averageScore, childName, date, difficultyFactor, netScore, rawScore, score, teamName.*
- **Implementation constructs**. *fileOfTeamMemberData, listOfScheduledMeets, group, number.*
- **Derived class**. *ageCategory* is readily computed from a competitor's age.
- **Operations**. *computeAverage, register.*

- **Out of scope.** *routine.*

After eliminating improper classes we are left with *Competitor, Event, Figure, Judge, League, Meet, Scorekeeper, Season, Station, Team,* and *Trial.*

12.12 We use a combination of the OCL and pseudocode to express our queries.

[Some of our answers to these problems traverse a series of links. Chapter 15 explains that each class should have limited knowledge of a class model and that operations for a class should not traverse associations that are not directly connected to it. We have violated this principle here to simplify our answers. A more robust answer would define intermediate operations to avoid these lengthy traversals.]

a. Find all the members of a given team.

```
Team::retrieveTeamMembers ()
returns set of competitors
  return self.competitor;
```

c. Find the net score of a competitor for a given figure at a given meet. There are several ways to answer this question, one of which is listed below.

```
Competitor::findNetScore (figure, meet)
returns netScore
  event:= meet.event intersect figure.event;
     /* the above code should return exactly one      */
     /* event (otherwise there is an implementation */
     /* error). This is a constraint implicit in the */
     /* problem statement that is not expressed in   */
     /* the class model.                              */
  trial := event.trial intersect self.trial;
  if trial == NIL then return ERROR
  else return trial.netScore;
  end if
```

e. Find the average score of a competitor over all figures in a given meet.

```
Competitor::findAverage (meet) returns averageScore
  trials:= meet.event.trial intersect
     self.trial;
  if trials == NIL then return ERROR
  else
     compute average as in answer (d)
     return average;
  end if
```

g. Find the set of all individuals who competed in any events in a given season.

```
Season::findCompetitorsForAnyEvent ()
returns set of competitors
  return self.meet.event.trial.competitor;
```

12.14 The revised diagrams are shown in Figure A12.7–Figure A12.10. Figure A12.7 is a better model than the ternary because dateTime is really an attribute. Figure A12.8 is also better than the ternary because *UniversityClass* is likely to be a class with attributes, operations, and other relationships. The third ternary is not atomic because the combination of a *Seat* and a *Concert* determine the *Person*. The fourth ternary also is not atomic; this one can be restated as two binary associations.

13.14 The application manages data for competitive meets in a swimming league. The system stores swimming scores that judges award and computes various summary statistics.

13.15 Actors are competitor, scorekeeper, judge, and team.

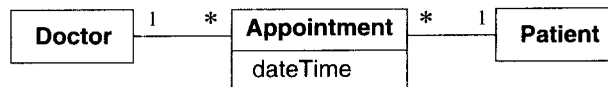


Figure A12.7 Class diagram for appointments

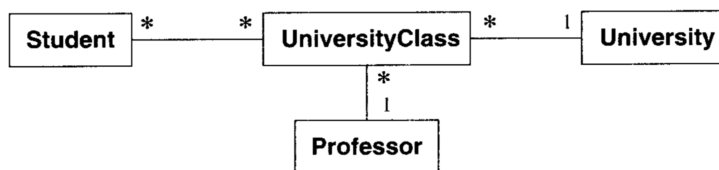


Figure A12.8 Class diagram for university classes

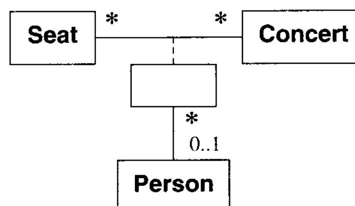


Figure A12.9 Class diagram for reservations

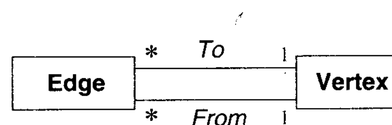


Figure A12.10 Class diagram for directed graphs

13.16 Here are definitions for the use cases. Figure A13.12 shows a use case diagram.

- **Register child**. Add a new child to the scoring system and record the name, age, address, and team name. Assign the child a number.
- **Schedule meet**. Assign competitors to figures and determine their starting times. Assign scorekeepers and judges to stations.
- **Schedule season**. Determine the meets that comprise a season. For each meet, determine the date, the figures that will be performed, and the competing teams.
- **Score figure**. A scorekeeper observes a competitor's performance of a figure and assigns what he/she considers to be an appropriate raw score.
- **Judge figure**. A judge receives the scorekeepers' raw scores for a competitor's performance of a figure and determines the net score.

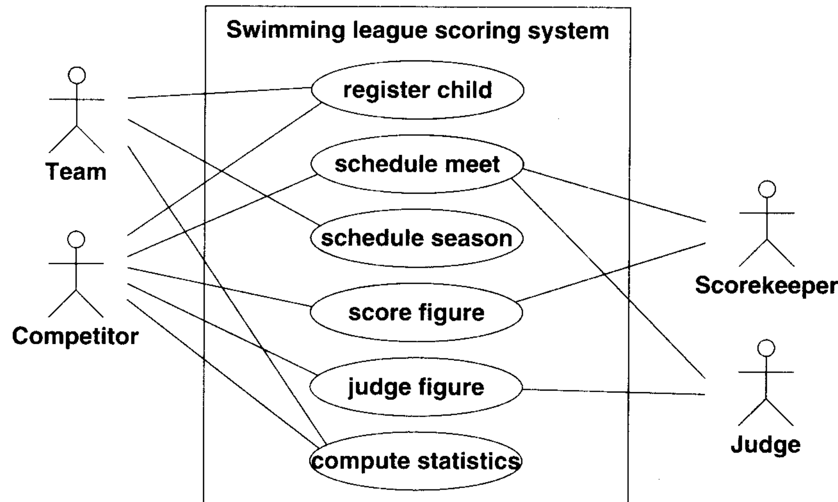


Figure A13.12 Use case diagram for the swimming league scoring system

- **Compute statistics.** The system computes relevant summary information, such as top individual score for a figure and total team score for a meet.

13.21 Figure A13.14 shows a partial shopping list of operations.

14.6 Figure A14.1 shows one possible partitioning.

14.7 A single program provides faster detection and correction of errors and eliminates the need to implement an interface between two programs. With a single program, any errors that the system detects in the process of converting the class diagram to a database schema can be quickly communicated to the user for correction. Also, the editing and the conversion portions of the program can share the same data, eliminating the need for an interface such as a file to transfer the class diagram from one program to another.

Splitting the functionality into two programs reduces memory requirements and decouples program development. The total memory requirement of a single program would be approximately equal to the sum of the requirements of two separate programs. Since both programs are likely to use a great deal of memory, performance problems could arise if they were combined. Using two separate programs also simplifies program development. The two programs can be developed independently, so that changes made in one are less likely to impact the other. Also, two programs are easier to debug than one monolithic program. If the interface between the two programs is well defined, problems in the overall system can be quickly identified within one program or the other.

Another advantage of splitting the system into two programs is greater flexibility. The editor can be used with other back ends, such as generating language code declarations. The relational database schema generator can be adapted to other graphical front ends.

14.10 Here is an evaluation of each solution.

a. **Do not worry about it at all**. Reset all data every time the system is turned on. This is the cheapest, simplest approach. It is relatively easy to program, since all that is needed is an ini-

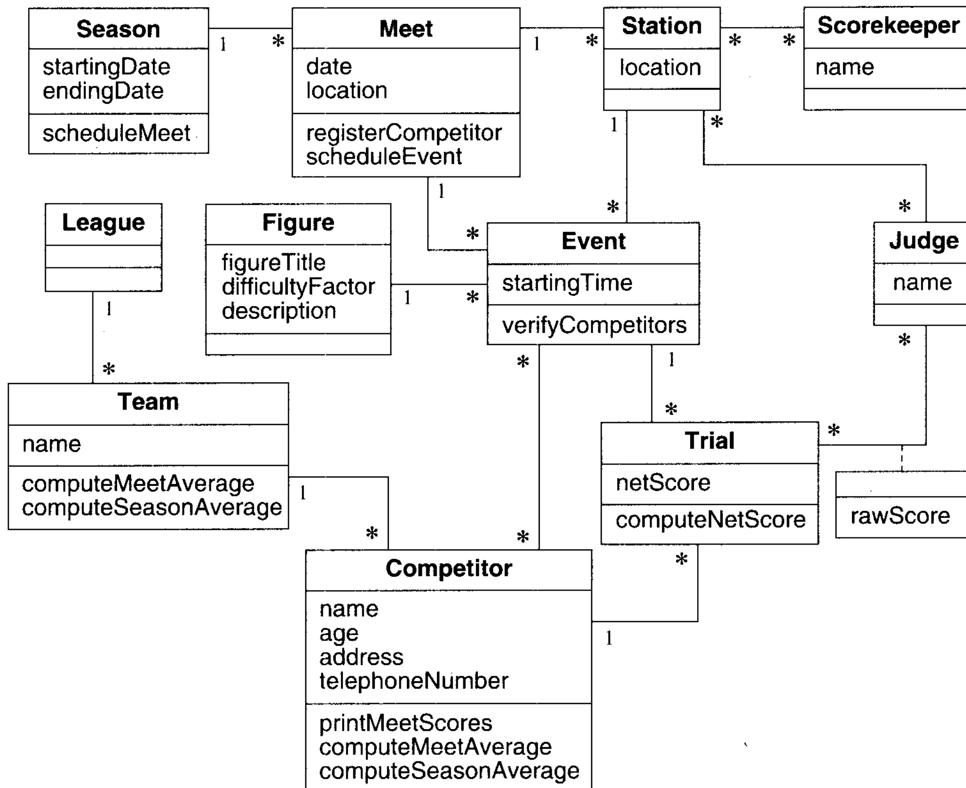


Figure A13.14 Partial class diagram for a scoring system including operations

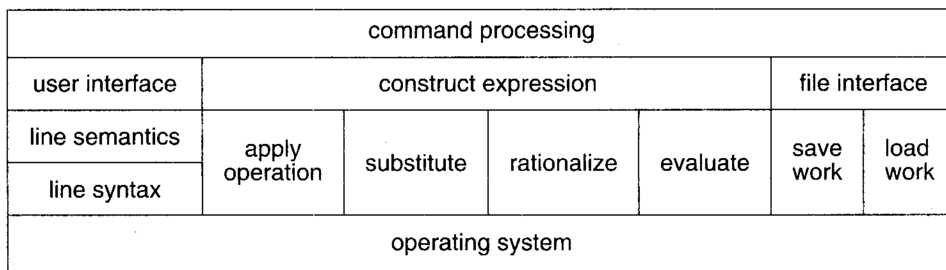


Figure A14.1 Block diagram for an interactive polynomial symbolic manipulation system

tialization routine on power-up to allow the user to enter parameters. However, this approach cannot be taken for systems that must provide continuous service or that must not lose data during power loss.

c. **Keep critical information on a magnetic disk drive. Periodically make full and/or incremental copies on magnetic tape**. This approach is moderately expensive and bulky. In the event of a power failure, the system stops running. An operating system is required to cope with the disk and tape drive. An operator is required to manage the tapes, which would preclude applications where unattended operation is required.

e. **Use a special memory component**. This approach is relatively cheap and is automatic. However, the system cannot run when power is off. Some restrictions may apply, such as a limit on the number of times data can be saved or on the amount of data that can be saved. A program may be required to save important parameters as power is failing.

14.11a. Four-function pocket calculator. Do not worry about permanent data storage at all. All of the other options are too expensive to consider. This type of calculator sells for a few dollars and is typically used to balance checkbooks. Memory requirements are on the order of 10 bytes.

c. **System clock for a personal computer**. Only a few bytes are required, but the clock must continue to run with the main power off. Battery backup is an inexpensive solution. Clock circuits can be designed that will run for 5 years from a battery.

e. **Digital control and thermal protection unit for a motor**. On the order of 10 to 100 bytes are needed. This application is sensitive to price. An uninterruptable power supply is too expensive to consider. Tape and disk drives are too fragile for the harsh environment of the application. Use a combination of switches, special memory components, and battery backup. Switches are a good way to enter parameters, since an interface is required anyway. Special memory components can store computed data. A battery can be used to continue operation with power removed but presents a maintenance problem in this application. We would question the last requirement, seeking alternatives such as assuming that the motor is hot when it is first turned on or using a sensor to measure the temperature of the motor.

14.12a. A description of the diagram, ignoring tabs, spaces, and line feeds, is:

```
(DIAGRAM
  (CLASS
     (NAME "Polygon"))
  (CLASS
     (NAME "Point")
     (ATTRIBUTE "x")
     (ATTRIBUTE "y"))
  (ASSOCIATION
     (END (NAME "Polygon") ONE)
     (END (NAME "Point") MANY)))
```

14.13 The hardware approach is fastest, but incurs the cost of the hardware. The software approach is cheapest and most flexible, but may not be fast enough. Use the software approach whenever it is fast enough. General-purpose systems favor the software approach, because of its flexibility. Special-purpose systems can usually integrate the added circuitry with other hardware.

Actually, there is another approach, firmware, that may be used in hardware architectures. Typically, in this approach a hardware controller calculates the CRC under the direction of a microcoded program, which is stored in a permanent memory that is not visible externally. We will count this approach as hardware.

a. **Floppy disk controller**. Use a hardware approach. Flexibility is not needed, since a floppy disk controller is a special-purpose system. Speed is needed, because of the high data rate.

c. **Memory board in the space shuttle**. Use hardware to check memory. This is an example of a specific application, where the function can probably be integrated with the circuitry in the memory chips. The data rate is very high.

e. **Validation of an account number**. Use a software approach. The data rate is very low. (The system handling the account number is probably running on a general-purpose computer.)

15.6 Figure A15.1 enforces a constraint that is missing in Figure E15.1: Each *BoundingBox* corresponds to exactly one *Ellipse* or *Rectangle*. One measure of the quality of an class model is how well its structure captures constraints.

We have also shown *BoundingBox* as a derived object, because it could be computable from the parameters of the graphic figure and would not supply additional information.

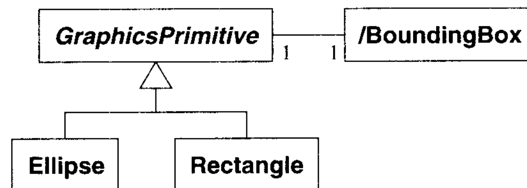


Figure A15.1 Revised class diagram for a bounding box

15.9 The derived association in Figure A15.3 supports direct traversal from *Page* to *Line*. Derived entities have a trade-off—they speed execution of certain queries but incur an update cost to keep the derived data consistent with changes in the base data. The *Page_Line* association is the composition of the *Page_Column* and *Column_Line* associations.

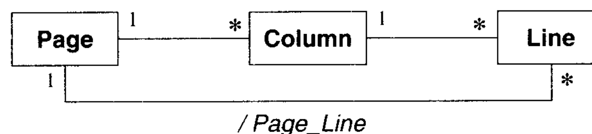


Figure A15.3 A revised newspaper model that can directly determine the page for a line

15.13 The code listed below sketches out a solution. This code lacks internal assertions that would normally be included to check for correctness. For example, error code should be included to handle the case where the end is a subclass and the relationship is not generalization. In code that interacts with users or external data sources, it is usually a good idea to add an error check as an else clause for conditionals that "must be true."

```
traceInheritancePath (class1, class2): Path
{
  path := new Path;
// try to find a path from class1 as descendant of class2
  classx := class1;
  while classx is not null do
      add classx to front of path;
      if classx = class2 then return path;
      classx := classx.getSuperclass();
```

```
// didn't find a path from class1 up to class2
// try to find a path from class2 as descendant of class 1
  path.clear();
  classx := class2;
  while classx is not null do
      add classx to front of path;
      if classx = class1 then return path;
      classx := classx.getSuperclass();
  // the two classes are not directly related
  // return an empty path
  path.clear();
  return path;
}

Class::getSuperclass (): Class
{
  for each end in self.connection do:
      if the end is a Subclass then:
         relationship := end.relationship;
         if relationship is a Generalization then:
            otherEnds := relationship.end;
            for each otherEnd in otherEnds do:
               if otherEnd is a Superclass then:
               return otherEnd.class
  return null;
}
```

15.16 Figure A15.5 shows the revised model. Political party membership is not an inherent property of a voter but a changeable association. The revised model better represents voters with no party affiliation and permits changes in party membership. If voters could belong to more than one party, then the multiplicity could easily be changed. *Parties* are instances, not subclasses, of class *PoliticalParty* and need not be explicitly listed in the model; new parties can be added without changing the model and attributes can be attached to parties.

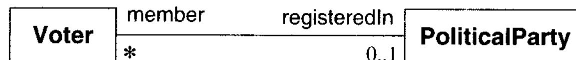


Figure A15.5 A revised model that reifies political party

15.18 The left model in Figure A15.6 shows an index on points using a doubly qualified association. The association is sorted first on the *x* qualifier and then on the *y* qualifier. Because the index is an optimization, it contains redundant information also stored in the *Point* objects.

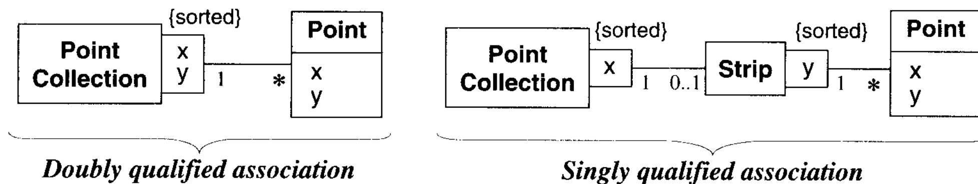


Figure A15.6 Models for sorted collections of points

The right model shows the same diagram using singly qualified associations. We introduced a dummy class *Strip* to represent all points having a given *x*-coordinate. The right model would be easier to implement on most systems, because a data structure for a single sort key is more likely to be available in a class library. The actual implementation could use B-trees, linked lists, or arrays to represent the association.

The code listed below specifies search, add, and delete methods.

```
PointCollection::search (region: Rectangle): Set of Point
{
  make a new empty set of points;
  scan the x values in the association until x ≥ region.xmin;
  while the x qualifier ≤ region.xmax do:
     scan the y values for the x value until y ≥ region.ymin;
     while the y qualifier ≤ region.ymax do:
        add (x,y) to the set of points;
        advance to the next y value;
     advance to the next x value;
  return the set of points;
}

PointCollection::add (point: Point)
{
  scan the x values in the association until x ≥ point.x;
  if x = point.x then
     scan the y values for the x value until y ≥ point.y
  insert the point into the association at the current
     location;
}

PointCollection::delete (point: Point)
{
  scan the x values in the association until x ≥ point.x;
  if x = point.x then
     scan the y values for the x value until y ≥ point.y
     if y = point.y then
        for each collection point with the current x,y value
           if collection point = point
              then delete it and return
  report point not found error and return
}
```

Note that the scan operation should be implemented by a binary search to achieve logarithmic rather than linear times. A scan falls through to the next statement if it runs out of values.

17.2 An arrow indicates that the association is implemented in the given direction.

- **Text <-> Box**. The user can edit text and the box must resize, so there should be a pointer from text to box. Text is allowed only in boxes, so we presume that a user may grab a box and move it, causing the enclosed text to also move. So there should be a pointer from box to text.
- **Connection <-> Box**. A box can be dragged and move its connections, so there must be pointers from box to connections. Similarly, a link can be dragged and move its connections to boxes, so there must also be a pointer from connection to box. There is no obvious ordering.
- **Connection <-> Link**. Same explanation as previous bullet.

- **Collection -> ordered Box**. Given a collection, we must be able to find the boxes. There does not seem to be a need to traverse the other way. There likely is an ordering of boxes, regarding their foreground / background hierarchy for visibility.
- **Collection -> ordered Link**. Same explanation as previous bullet.

18.6a. Here is Java code to implement a bidirectional, one-to-one association between classes *A* and *B*, using a pointer (reference variable) in each class. Each class maintains its own association end and calls on the associated class to maintain the other side. Each class contains an internal attribute, *_updateInProgress*, that breaks the potential infinite recursion. We show only the attributes and methods needed to implement the association.

We demonstrate class *A*; class *B* would contain the same code, but classes *A* and *B* and objects *a* and *b* would be substituted with *B, A, b,* and *a*, respectively. Thus the field *private B b* would become *private A a*, and the method *A.SetB(B newB)* would become *B.SetA(A newA)*. Note that we have minimized error handling as well as omitted boolean or enumerated returns and proper exception handling.

This code assumes the most rigid of access control. Classes are presumed to exist in separate packages and so can access only each other's public `elements`.

```
// in Java
// class A with a one-to-one association to class B

import BPackage.*;

public class A {
  private B b = null;
  private boolean _updateInProgress = false;

  // Check if A has a B
  public boolean hasB () {
     return b != null;
  }

  // Given an A, bind newB to it with a one-to-one association.
  public void setB (B newB) {
     if (newB == null) return; // don't "associate" to null;
             // caller should call RemoveB instead!
     if (_updateInProgress) return; // break mutual recursion
     if (b == newB) return; // this A already bound to newB
     if (newB.hasA()) return;
        // newB must lack an association
     if (hasB()) removeB();
        // remove current b, if any; only 1:1 allowed

     _updateInProgress = true;
     newB.setA(this);
        // request newB to update its end of association
     b = newB; // update this end of association
     _updateInProgress = false;
```

```
    }

    // Remove the one B that may be associated.
    // Note that a 1-to-1 assoc does not need a remove argument.
    public void removeB() {
        if ( hasB() == false ) return; // no B to remove!
        if (_updateInProgress) return; // break mutual recursion

        _updateInProgress = true;
        b.removeA();
            // request B to remove its end of association
        b = null; // remove this end of association
        _updateInProgress = false;
    }
};
```

Often, classes that mention one another in their interfaces are packaged together and may therefore have more extensive and privileged knowledge of one another. In such cases, it may reasonable for one side or another of an association to take responsibility for maintaining both association ends. This can provide optimization, centralize update code, and avoid the need for devices (see *_updateInProgress* in code) to terminate recursion. However, it may imply an increased level of code dependency among associated classes.

This is not necessarily "bad" or "disencapsulating." A logical dependency already exists, expressed in the interface(s). The ability to restrict operations to selected callers may result in safer, more accurate, more encapsulated (from the public view) code, even as it exposes selected internals to an associated class. Consider the case where the ability to trigger termination of a link should be restricted to the linked object itself. In Java, the publicly available *remove()* methods would instead be given default package access, and by packaging *A* and *B* together, invocation of those methods would be reserved for call only across the link and not by those outside the package.

Alternatively, *A* and *B* might selectively expose their pointers and allow one end of the association to perform all update activities. We offer the essentials of such a C++ solution.

```
//in C++
class B; // forward declaration

class A {

    friend class B; // or per function if B has been declared:
        //   friend void B::setA(A&);
        //   friend void B::removeA();
    B* b;

    void removeB(); // B can ask A to remove B; others cannot

public:

    bool hasB () { return b != 0; }
```

```
    void setB (B& newB);
        // or use pointer parameter to allow null b
    };


void A::setB(B& newB)
{
    if (b == &newB) return; // this A already bound to newB
    if (newB.hasA()) return; // newB must lack an association
    if (hasB()) removeB();
        // remove current b, if any; only 1:1 allowed

    b = &newB; // update this end of association
    b->a = this;
}

void A::removeB()
{
    if ( !hasB() ) return; // no B to remove!
    if (b->a != this) return; // whoops -- not bidirectional!

    b->a = 0; // remove old b's pointer to this A
    b = 0; // remove this end of association
}
```

19.13 We infer that a *Route* has 2 *Cities* from the problem statement. We could not deduce that from the SQL code alone.

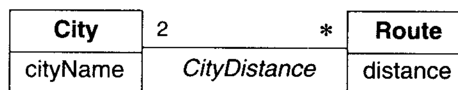


Figure A19.12 Class model for Figure E19.6

19.14 SQL code to determine distance between two cities for Figure E19.6.

```
SELECT distance
FROM Route R, City C1, City C2,
     City_Distance CD1, City_Distance CD2
WHERE C1.city_ID = CD1.city_ID AND
      CD1.route_ID = R.route_ID AND
      R.route_ID = CD2.route_ID AND
      CD2.city_ID = C2.city_ID  AND
      C1.city_name = :aCityName1  AND
      C2.city_name = :aCityName2;
```

19.15 Here is the class diagram.

19.16 SQL code to determine distance between two cities for Figure E19.7. We don't know which name is 1 and which name is 2, so the SQL code allows for either possibility.

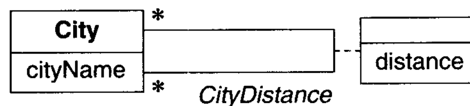


Figure A19.13 Class model for Figure E19.7

```
SELECT distance
FROM City C1, City C2, City_Distance CD
WHERE C1.city_ID  = CD.city1_ID AND
      CD.city2_ID = C2.city_ID  AND
      ((C1.cityName = :aCityName1 AND
        C2.cityName = :aCityName2) OR
       (C1.cityName = :aCityName2 AND
        C2.cityName = :aCityName1));
```

19.17 We make the following observations about Figure A19.12 and Figure A19.13.

- Figure A19.12 has an additional table. Figure A19.12 could store multiple routes between the same cities with different distances. Given the lack of explanation about route in the problem statement (is it a series of roads with different distances or is it the distance by air?), this may or may not be a drawback.
- Figure A19.13 is awkward because of the symmetry between city1 and city2. Either data must be stored twice with waste of storage, update time, and possible consistency problems, or special application logic must enforce an arbitrary constraint.

We need to know more about the requirements to choose between the models.

20.3a. This is an example of poor programming style. The assumption that the arguments are legal and the functions called are well behaved will cause trouble during program test and integration.

The following statements will cause the program to crash if the argument to *strlen* is zero:

```
rootLength = strlen(rootName);
suffixLength = strlen(suffix);
```

The following statement will assign zero to *sheetName* if the program runs out of memory, causing a program crash during the call to *strcpy* later in the function:

```
sheetName = malloc(rootLength + suffixLength + 1);
```

The following statements will cause the program to crash if any of the arguments are zero:

```
sheetName = strcpy(sheetName, rootName);
sheetName = strcat(sheetName, suffix);
```

If *sheetType* is invalid, the switch statement will fall through, leaving *sheet* without an assigned value. Also, it is possible that the call to *vertSheetNew* or the call to *horizSheetNew* could return zero for some reason. Either condition would make it possible for the following statement to crash:

```
sheet->name = sheetName;
```

Index

A

abstract class **69–70**, 81, 163, 326
- convention for 70
- notation for 69

abstract operation 69, 326
- notation for 70

abstract signal 114, 125, 163
abstract use case 149, 163
abstraction **16**, 22, 76, 93, 199–200, 405, 415
- exercise 212–213

access control 341
- in C++ 315, **322–325**
- in Java 317, **322–325**

accidents of software 3
activation **152**
active object 152
active value 283
activity **99–103**, **141–143**
- entry 100–101, 103, 113
- exit 100–101, 103, 113
- notation for 99, 140, 142

activity diagram **140–144**, **154–157**, 223
- notation for 140, 155, 156
- practical tips 143–144

activity token 143
actor **131–132**, 134–136
- finding 217
- notation for 134–135

Ada 288
aggregation 64, **66–69**, 163–164, 191
- and concurrency 114–115
- exercise 83, 84
- implementing
  - for a programming language 310
  - for a relational database 354
- notation for 67
- vs. association 67
- vs. composition 67–68

agile programming 311
algorithm, design of 274–278
allocation
- of subsystems to processors 248–250

analysis 4, 167–169, **181–215**, **216–239**, 299
- application analysis 168–169, **216–239**
- building application class model 224–227
- building application interaction model 216–224
- building application state model 227–233
- building domain class model 183–201
- building domain interaction model 204
- building domain state model 201–204
- choosing packages 201
- data dictionary 187
- domain analysis 168, **181–215**
- finding associations 187–192
- finding attributes 192–194
- finding classes 183–186
- iteration of 196–199, 204–206
- shifting abstraction 199–200
- testing access paths 196

using generalization 194–196
ancestor class 38
antisymmetry 66, 67
a-part-of. *See* aggregation
API 367
application model 4, 168–169
construction of 216–234
application programming interface. *See API*
architecture 5, 167, 169, **240**, **244–246**
canonical architectures **256–262**
association **27–36**, 66
as a class **33–36**
convention for 28
directionality 28
exercise 84
finding 187–192
implementing 306–310, 341
for a relational database 353–354, 360–361
in C++ 329–331
in Java 329–331
importance of 28, 50
meaning of 30
n-ary **64–66**, 81, 83, 189–190, 310
notation for 28
qualified **36**, 45, 48, 64, 78, 191, 193, 310
ternary **64–66**, 81, 83, 189–190
traversal of 31, 64, 65
vs. aggregation 67
association class **33–36**, 78, 193–194
exercise 55, 59
implementing 310
for a relational database 353
in C++ 330
in Java 330
notation for 33
traversal of 45
vs. class 36
association end **31–32**, 48, **63–64**
bag **33**, 51, 310
changeability 64
multiplicity **29–31**, 48, 64, 78, 191
navigability 64
notation for 31, 65
ordered **32**, 64, 78, 310
sequence **33**, 51, 310
uniqueness of name 32
visibility 64
attribute 2, **23–24**
convention for 24
finding 192–194
missing compartment 26
multiplicity **61**, 78
notation for 24, 26, 61
scope **62**
autoboxing (in Java) 316

B

bag **33**, 45, 48, 51, 310
batch transformation 257
boundary class 226, 289, 421
boundary condition 255
branch (in activity diagram) **142**
notation for 142

C

C 288
C++ 62, 76, 288, **314–347**
access control 315, 322–325
constructor 332–334
destructor 335–337
friend 323, 341
implementing
association 329–331
association class 330
class 322
data type 318–322
enumeration 320–322
generalization 325–329
namespace 315, 324
practical tips 341–342
candidate key 83, 350
cardinality 30, 61
change event **92**
vs. guard condition 95
notation for 92
changeability 64
class 2, **22**
abstract **69–70**, 81
ancestor 38
concrete **69–70**, 81
convention for 23
descendant 38
finding 183–186
implementing

for a relational database 352–353
in C++ 322
in Java 322
notation for 23, 26
class descriptor 75
class design 5, 167, 169, **270–297**, 300
class generalization **37–41**, 163
vs. enumeration 61
class icon 80, 81
class library **242–243**
class model 6, 17, 19, **21–59**, **60–87**, 161
construction of 183–201, 224–227
vs. Entity-Relationship (ER) 50
navigation **43–47**
practical tips 48–49, 81, 197–198
reverse engineering 418–419
vs. other models 123–124, 162–164, 224, 227, 233
classification 2
classifier 126, 163
client-server architecture 244
CLOS 25
closed architecture 245
command 225
Common Lisp Object System. *See* CLOS
completion transition 102
composite state 112
composition **67–68**, 73, 163–164
implementing 310
for a relational database 354
notation for 68
vs. aggregation 67–68
concrete class **69–70**, 81
convention for 70
notation for 69
concrete signal 125
concrete use case 149
concurrency 90, **114–118**, 124, **143**, **246–248**, 254, 311, 349
notation for 115, 143
synchronization 116
condition. *See* guard condition
constraint **77–78**, 81
exercise 83
in a relational database 350
on link 29
notation for 72, 78
on generalization set 77
on link 78
on object 77
constructor 316, **332–334**, 341
container class 276
continuous transformation 258
control 17, **253–255**
concurrent 254
event-driven 254
implementation of 226–227
merge 117, 143
procedure-driven 253–254
split 117, 143
controller 226–227
convention for
abstract class 70
association 28
attribute 24
class 23
concrete class 70
generalization 37
link 28
object 23
operation 25
package 80, 81
propagation 69
scope 62
state 92
state diagram 99
transition 95
value 24

D

data conversion 368–369, 372
data dictionary 187, 405, 418
data flow diagram 19
data type
enumeration **60–61**, 81, 194
implementing
in C++ 318–322
in Java 318–322
database 24, **250–252**, **348–379**
See also DBMS
object-oriented 62, **370–371**, 372
relational **349–370**
vs. files 251
database layer 368
database management system. *See* DBMS

DBMS 76, 116, 260–261, 348–349
See also database
choosing a product 351
delegation 72
as a substitute for multiple inheritance 73–75
to avoid improper inheritance 287–288, 328, 341, 384
Demeter, law of 289, 292, 370, 385
denormalization 351
deployment 167, 170
derived entity **79–80**, 81, 186, 190, 340–341
exercise 86
for design optimization 281–283
notation for 79
descendant 38
destructor 316, **335–337**, 342
development life cycle **170–171**, 298–299
iterative 171, 395–402
rapid prototyping 396–397
waterfall 170–171, 395–396
development stage **167–170**
analysis 4, 167–169, **181–215**, **216–239**, 299
class design 5, 169, **270–297**, 300
implementation 5, 169, **303–313**
system conception 4, 168, **173–179**, 299
system design 5, 169, **240–269**, 300
DFD. *See* data flow diagram
diagram layout 48
directed graph
exercise 57, 374–375
direction (of an argument) **26**
directionality (of association) 28
distribution 62, 140, 311, 349
do-activity **100**
notation for 100
documentation 49, 388
domain model 4, 168
construction of 183–204
dynamic model 19
dynamic simulation 259–260

E

effect **99**
encapsulation **6**, 28, 369–370, 372, 384
enterprise model 404
Entity-Relationship (ER) model 50
entry activity **100–101**, 103, 113
notation for 101
enumeration **60–61**, 194, 341
implementing
for a relational database 361, 372
in C++ 320–322
in Java 320–322
notation for 61
vs. generalization 61, 81, 195
essence of software 3, 8
event **90–92**
finding 203, 219, 220–222, 229
notation for 99
event-driven control 254
exit activity **100–101**, 103, 113
notation for 101
extend (for use case) **148**, 151
notation for 148
extensibility **384–385**
improved by use of inheritance 286
extent 62, 81

F

feature 2, **25**, 62
file **250–252**
final (in Java) 327
finding
associations 187–192
attributes 192–194
classes 183–186
generalizations 194–196
finite state machine. *See* state diagram
firing of transition 94
flowchart 140
focus of control **152**
folding association attributes into a class 34, 48
foreign key 350, **360–361**, 372
indexing 361–362
Fortran 288
fourth-generation language (4GL) 367
framework **243–244**
friend (in C++) 315, 323, 341
functional model 19

G

garbage collection 317, 336, 337
generalization 7, 48, 163
See also inheritance

class generalization **37–41**
convention for 37
exercise 84
finding 194–196
implementing
for a relational database 356–358, 360, 362
in C++ 325–329
in Java 325–329
notation for 37
signal generalization **114**
use case generalization **149–150**
uses of 40
vs. enumeration 61, 195
generalization set name **39**
global resource **252–253**
guard condition **95**, 103
vs. change event 95
notation for 95
guardian object 252
guidelines for programming 380–391

H

hardware 248–250
higraph 126

I

IDEF1X (database notation) 50
identity 1, 22, 24, 51, 73, 75, 193
exercise 12, 57–58
for a relational database 358–359
implementation 5, 167, 169
for a programming language **303–313**, **314–347**
for a relational database **348–379**
vs. policy 289–290, 381–382
include (for use case) **147–148**, 150, 151
notation for 147
index 281–282, **361–362**, 372
information hiding 6, **288–289**, 369–370, 372, 384
inheritance 2, 7, **37–41**, 48
See also generalization
abstracting out common behavior 285–287
misuse for implementation 41, 287–288, 384
multiple inheritance **70–75**, 196
notation for 37
rearranging classes and operations 285
instance 2, 22
integration testing 311
interaction model 6, 18, 19, **131–146**, **147–160**, 162, 311
construction of 204, 216–224
reverse engineering 419–420
vs. other models 162–164, 224, 227, 233
interactive interface 259
interface (in Java) 316, 326, 341
is-a. *See* generalization
iterative development 171, 196–199, 395–402

J

Java 62, 76, 288, **314–347**
access control 317, 322–325
constructor 332–334
implementing
association 329–331
association class 330
class 322
data type 318–322
enumeration 320–322
generalization 325–329
interface 316, 326, 341
package 316–317, 322–323, 323–324, 341
practical tips 341–342

K

key
candidate 350
foreign 350, 372
primary 350

L

law of Demeter 289, 292, 370, 385
layer **245–246**
legacy data 368–369
library. *See* class library
lifeline 137, **152**
link **27–36**
convention for 28
creation 337–339
destruction 339–340
notation for 28
Lisp 76
lock 252

M

maintenance 167, 170, 422
memory management 315, 317, 332–340
mentoring 410
merging control 117, 143
    notation for 118, 143
metaclass 76
metadata **75–76**, 368
metamodel 404
    exercise 85, 86, 130
method 2, **25**, 62
model 15–18
    class 6, 17, 19, **21–59**, **60–87**, 161
    interaction 6, 18, 19, **131–146**, **147–160**, 162, 311
    relationship among 18, 123–124, 162–164
    state 6, 17, 19, **90–109**, **110–130**, 161–162, 311
modeling effort 413
modeling personnel 409–410
modeling pitfall 404–406
modeling session 406–408
multiple classification 72, 73
multiple inheritance **70–75**, 81, 196, 341
    exercise 85, 86
    implementing
        for a relational database 358
        in C++ 328
        in Java 326
    kinds of 71–72
    notation for 72
    workarounds 73–75
multiplicity **29–31**, 48, 64, 78, 191
    for an attribute **61**, 78
    notation for 29
    vs. cardinality 30

N

name, importance of 48, 387
namespace (in C++) 315, 324
n-ary association 63, **64–66**, 81, 83, 189–190, 310
    implementing
        for a programming language 66, 329
        for a relational database 353
    notation for 65
navigability 64
nested state **111–113**, 163
    notation for 112
nested state diagram **110–111**
    notation for 111
new 332
normal form 351, 372
notation for
    abstract class 69
    abstract operation 70
    activity 99, 101, 140, 142
    activity diagram 140, 155, 156
    actor 134–135
    aggregation 67
    association 28
    association class 33
    association end 31
    attribute 24, 26, 61
    branch 142
    change event 92
    class 23, 26
    composition 68
    concrete class 69
    concurrency 115, 143
    constraint 72, 78
    derived entity 79
    do-activity 100
    enumeration 61
    event 99
    generalization 37
    guard condition 95
    inheritance 37
    link 28
    merging control 118, 143
    multiple inheritance 72
    multiplicity 29
    n-ary association 65
    nested state 112
    nested state diagram 111
    object 23
    operation 25, 26
    package 80
    qualified association 36
    qualifier 36
    scope 62
    sequence diagram 152, 153, 154
    signal 91
    signal generalization 114
    splitting control 117, 143
    state 92, 97

state diagram 98
ternary association 65
time event 92
transition 95
use case 134–135
use case diagram 134–135
use case extension 148
use case generalization 149
use case inclusion 147
value 24
visibility 63
null 46, 61, 349

O

object 1, **21–22**
convention for 23
creation 332–334
destruction 335–337
notation for 23
Object Constraint Language. *See* OCL
object diagram **23**
object flow **156–157**
object identity
for a relational database 358–359, 372
in C++ 320
in Java 320
Object Management Group. *See* OMG
Object Modeling Technique. *See* OMT
object-oriented, meaning of 1
object-oriented database 62, **370–371**, 372
OCL
exercise 346
OCL (Object Constraint Language) **44–47**, 51, 65
exercise 59, 208–209, 210–211, 379
OMG (Object Management Group) 9
OMT (Object Modeling Technique) 9, 19, 50
OO-DBMS. *See* object-oriented database
open architecture 245
operation 2, **25**
abstract 69
assigning to a class 276–278
convention for 25
finding 233–234
missing compartment 26
notation for 25, 26
query **79**
scope **62**
shopping-list operation 234, 236, 276
optimization of design 280–283
ordering **32**, 33, 64, 78, 310
overloading (of methods) 315
override 40–41, 49

P

package **80–81**, 81, 290, 388
choosing during analysis 201
convention for 80, 81
Java 316–317, 322–323, 323–324, 341
notation for 80
visibility 62, 317
partition **245–246**
part-whole relationship. *See* aggregation
passive object 152
pattern 200, 206, **243–244**, 284, 411
peer-to-peer architecture 244
performance **241**
Petri net 105
pointer 28, 48
exercise 57
policy vs. implementation 289–290, 381–382
polymorphism 2, 7, **25**, 40, 315
postprocessor 367
practical tips
activity diagram 143–144
C++ 341–342
class model 48–49, 81, 197–198
Java 341–342
relational database 371–372
sequence diagram 140, 154
state model 103, 124–125
use case diagram 135–136, 150–151
preprocessor 367
primary key 350
private 62, 315, 317, 341
procedure-driven control 253–254
product assessment 404
programming language 24, 62, 72, 76, 116
Ada 288
C 288
C++ 76, 288, **314–347**
CLOS 25
coupling to a relational database 366–368, 372
Fortran 288

Java 76, 288, **314–347**
Lisp 76
Smalltalk 76
programming style 380–391
programming-in-the-large **387–390**
propagation 67, 68–69
convention for 69
protected 62, 315, 317, 324
public 62, 315, 317

Q

qualified association **36**, 48, 64, 78, 191, 193, 310
implementing
for a programming language 329
for a relational database 354
notation for 36
traversal of 45
qualifier **36**, 64, 193
notation for 36
query operation **79**
query optimization 369–370, 372

R

race condition **102**
rapid prototyping 396–397
RDBMS. *See* relational database
real-time system 260
refactoring **280**, 383
reference 28, 48
exercise 57
reference (in C++) 315
reification **76**, 284
exercise 86, 130
relational database **349–370**
coupling to a programming language 366–368, 372
data conversion 368–369
implementing
association 353–354, 360–361
class 352–353
generalization 356–358, 360
identity 358–359
practical tips 371–372
relational DBMS. *See* relational database
requirements **176–178**, 205–206
responsibility **273–274**
reuse 40, 71, 201, **242–244**, 348, **380–384**
reverse engineering **416–421**
review 49, 311, 405–406, 413, 415
risk (of development) 400–401
robustness **385–387**

S

scenario **136–137**, 219–220, 311
schema 348
scope **62**, 81, 83
convention for 62
notation for 62
script file 367
sequence **33**, 48, 51, 310
sequence diagram **137–138**, **152–154**, 222
notation for 152, 153, 154
practical tips 140, 154
service **244**
shopping-list operation 234, 236, 276
signal **91**
abstract 114, 125
concrete 125
notation for 91
sending 102–103
signal event **91**
signal generalization **114**, 163
notation for 114
signature **25**, 40, 63
simulation, dynamic 259–260
slicing 420
Smalltalk 76
specialization 40
splitting control 117, 143
notation for 117, 143
SQL code
vs. programming code 370
SQL language **349–370**
state **92–94**
composite 112
convention for 92
final 97
finding 202
initial 97
nested 111–113
notation for 92, 97
statechart 126
state diagram 76, **95–103**
construction of 229–230

convention for 99
nested 110–111
notation for 98
one-shot vs. continuous 96
state model 6, 17, 19, **90–109**, **110–130**, 161–162, 311
construction of 201–204, 227–233
practical tips 103, 124–125
reverse engineering 420
vs. other models 123–124, 162–164, 233
static 81, 334–335
stored procedure 367
subclass 2, **37**
submachine 111
substate 118
subsystem 244–246
superclass 2, **37**
swimlane **155**
synchronization 116
system architecture. *See* architecture
system boundary **217**
system conception 4, 167, 168, **173–179**, 299
system design 5, 167, 169, **240–269**, 300
system testing 311–312

T

table (in RDBMS) 349–350
ternary association 63, **64–66**, 81, 83, 189–190
exercise 211
implementing
for a programming language 329
for a relational database 353
notation for 65
promotion to a class 66
testing 5, 167, 169–170, **310–312**, 386–387, 398
of class model 196
this 331
thread of control **248**
time event **92**
notation for 92
tool (for software development) 411–413
training 167, 170, 410
transaction manager 260–261
transformation 303–306, 312
transition **94–95**
completion 102
convention for 95
notation for 95
transitive closure 66
transitivity 66
traversal of association 64, 65
trigger 68

U

UML 9, 10, 11, 19, 50, 51, 83
UML2 vs. UML1 33, 106, 126, 163
undirected graph
exercise 56–57, 376–377
Unified Modeling Language. *See* UML
unit testing 311
use case 6, **132–136**, 272–274, 311, 397
finding 218–219
notation for 134–135
use case diagram **131–136**, 147–151
notation for 134–135
practical tips 135–136, 150–151
use case extension **148**, 151, 223
notation for 148
use case generalization **149–150**, 163, 223
notation for 149
use case inclusion **147–148**, 150, 151, 223
notation for 147
user interface 259
specification of 225–226

V

value **23–24**
convention for 24
difference from object 24
notation for 24
view
for a relational database 362, 372
virtual method (in C++) 315, 327
visibility **62–63**, 64, 315, 317, 384–385
notation for 63

W

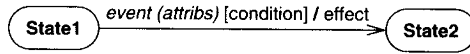
waterfall development 170–171, 395–396
wrapper 421–422

X

XML 369, 422

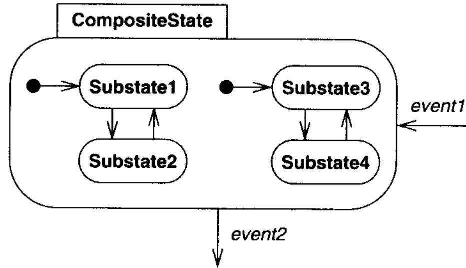
State Model Notation

Event causes Transition between States:

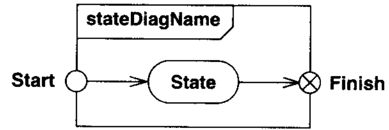


Initial and Final States:

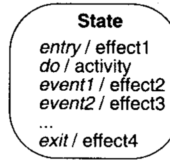
Concurrency within an Object:



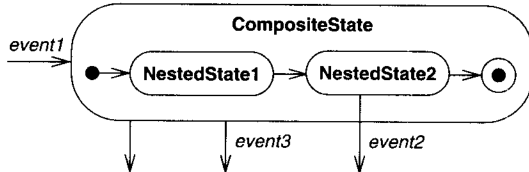
Entry and Exit Points:



Activities while in a State:

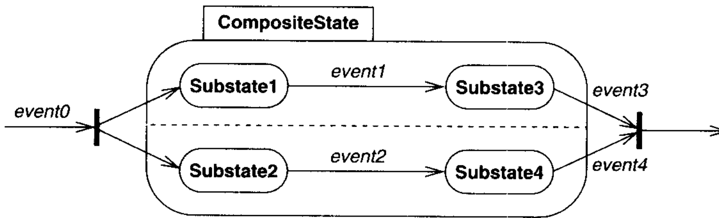


Nested State:



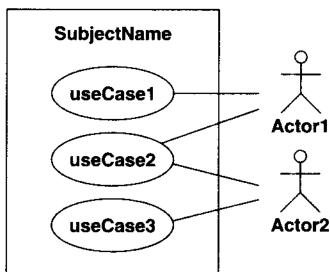
Splitting of control:

Synchronization of control:

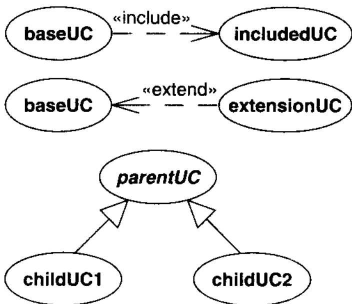


Interaction Model Notation

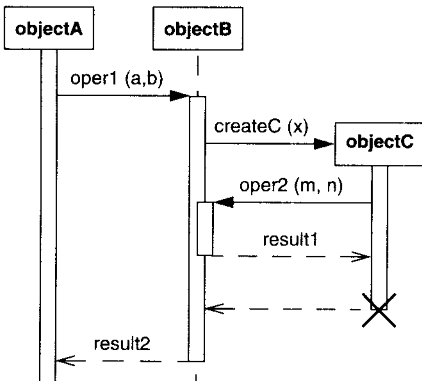
Use Case Diagram:



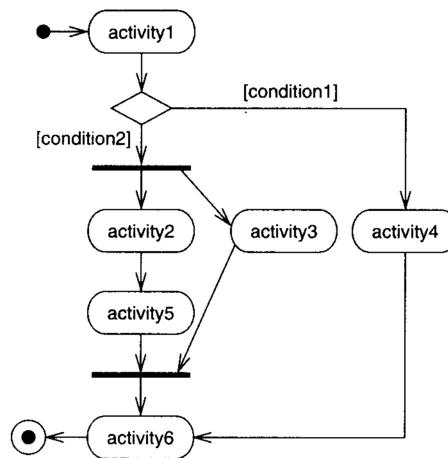
Use Case Relationships:



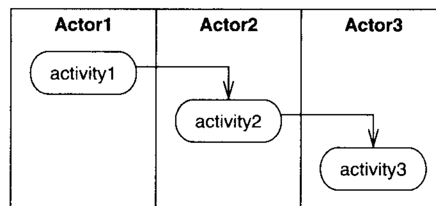
Sequence Diagram:



Activity Diagram:



Activity Diagram with Swimlanes:



Activity Diagram with Object Flows: