An Introduction to the Unified Modeling Language

A picture is worth a thousand words.

Most people refer to the Unified Modeling Language as UML. The UML is an international industry standard graphical notation for describing software analysis and designs. When a standardized notation is used, there is little room for misinterpretation and ambiguity. Therefore, standardization provides for efficient communication (a.k.a. "a picture is worth a thousand words") and leads to fewer errors caused by misunderstanding.

The U in UML stands for unified because the UML is a unification and standardization of earlier modeling notations of Booch, Rumbaugh, Jacobson, Mellor, Shlaer, Coad, and Wirf-Brock, among others. The UML most closely reflects the combined work of Rumbaugh, Jacobson, and Booch – sometimes called *the three amigos*. The UML has been accepted as a standard by the Object Management Group[1] (OMG). The OMG is a non-profit organization with about 700 members that sets standards for distributed object-oriented computing.

In this appendix, we bring together for ease of reference five fundamental UML models: use case, class, sequence, state, and activity diagrams. The intent is not for this to be your only UML reference, but to succinctly provide you with the essential 20% of the UML that will provide you with the 80% of the capability you will use often.

1. Use Case Diagrams

Use case diagrams are used during ***requirements elicitation and analysis*** as a graphical means of representing the functional requirements of the system. Use cases are developed during requirements elicitation and are further refined and corrected as they are reviewed (by stakeholders) during analysis. Use cases are also very helpful for ***writing acceptance test cases***. The test planner can extract scenarios from the use cases for test cases. Note: The use case diagram is accompanied by a textual use case flow of events. The flow of events is not explained in this document.

A *use case,* a concept invented by Ivar Jocbson (Jacobson, Christerson et al., 1992), is a sequence of transactions performed by a system that yields an outwardly visible, measurable result of value for a particular actor. A use case typically represents a major piece of functionality that is complete from beginning to end (Bruegge and Dutoit, 2000).

In UML, a use case is represented as an ellipse, as shown in Figure 1. In a Monopoly game, some use cases are: Enter Player Info, Buy House, and Draw Card. Give your use case a unique name expressed in a few words (generally no more than five words). These few words must begin with a present-tense verb phrase in active voice, stating the action that must take place (notice: **Enter** Player Info**, Buy** House, **Draw** Card, and **Switch** Turn).

[1] For more information on the OMG, see http://www.omg.org

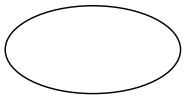

Figure 1: The UML symbol for a use case

An *actor* represents whoever or whatever (person, machine, or other) interacts with the system. The actor is not part of the system itself and represents anyone or anything that must interact with the system to:

- Input information to the system;
- Receive information from the system; or
- Both input information to and receive information from the system.

The total set of actors in a use case model reflects everything that needs to exchange information with the system (Rosenberg and Scott, 1999). In UML, an actor is represented as a stickman, shown below in Figure 2. In a Monopoly game, some actors are the player and a bad player (who has the audacity to want to take two turns in a row!). As you see, actors can be people or they can be other systems. The name of an actor is always a noun. However, the name should not be that of a particular person. Instead, the name should identify the role or set of roles the actor plays relative to one or more use cases.

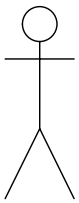

Figure 2: The UML symbol for an actor

A *use case diagram* is a visual representation of the relationships between actors and use cases together that documents the system's intended behavior. A simple use case diagram is shown in Figure 3.

Arrows and lines are draw between actors and use cases and between use cases to show their relationships. We discuss these relationships more detail later in this appendix. The default relationship between an actor and a use case is the «communicates» relationship, denoted by a line. For example, in Figure 3, the actor is communicating with the use case.

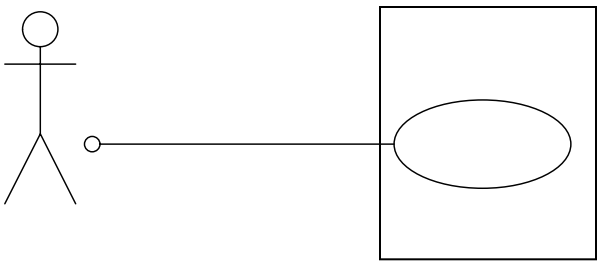

Figure 3: A UML use case diagram

There are several different kinds of relationships between actors and use cases. Earlier, we said that the default relationship is the «communicates» relationship. The `«communicates»` relationship indicates that one of these entities initiated invoked a request of the other. An actor communicates with use cases because actors want measurable results. It might not be quite as obvious that use cases can communicate with other use cases. This happens if a case needs information from or to initiate action of another use case. When a line or an arrow is drawn on a diagram and there is no label on the arrow, it is, by default, a `«communicates»` relationship.

There are two other kinds of relationships between use cases (not between actors and use cases) that you might find useful. These are «include» and «extend». You use the «include» relationship when a chunk of behavior is similar across more than one use case, and you don't want to keep copying the description of that behavior (Bruegge and Dutoit, 2000). This is similar to breaking out re-used functionality in a program into its own methods that other methods invoke for the functionality. For example, suppose many actions of a system require the user to login to the system before the functionality can be performed. These use cases would *include* the login use case.

The «include» relationship is not the default relationship. Therefore in a use case diagram, the arrow is labeled with «include» when one use case makes full use of another use case, as shown in Figure 4. The Draw Card and the Buy House both use the View Information functionality.

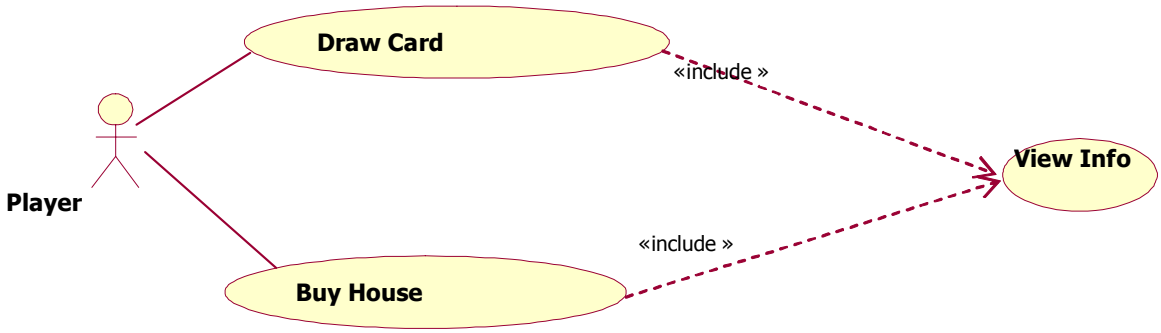

Figure 4: Includes Use Case

You use the «extend» relationship when you are describing a variation on normal behavior or behavior that is only executed under certain, stated conditions. The extend relationship is used when the alternative flow is fairly complex and/or multi-stepped,

possibly with its own sub-flows and alternative flows. For example, consider the players moving on a Monopoly board.

A player moves on the board because he or she has to go to jail.
A player moves on the board because he or she has to go to Free Parking.

This scenario involves a player moving. However, sometimes a player has to deal with "exceptional" situations – rather than just moving to a new property cell. Therefore, we can extend the Move use case with the Go to Jail and the Go to Free Parking use case (and some others) as shown in Figure 5.

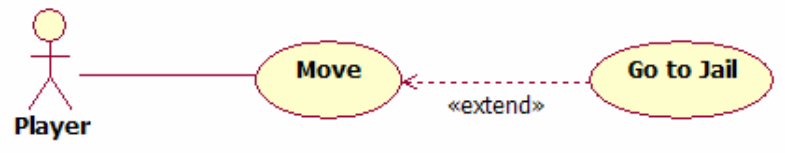

Figure 5: Extends Use Case

It is common to be confused as to whether to use the include relationship or the extend relationship. Consider the following distinctions between the two:

- Use Case X includes Use Case Y:
  X has a multi-step subtask Y. In the course of doing X or a subtask of X, Y will **always** be completed.

- Use Case X extends Use Case Y:
  Y performs a sub-task and X is a similar but more specialized way of accomplishing that subtask (e.g. closing the door is a sub-task of Y; X provides a means for closing a blocked door with a few extra steps). X **only happens in an exception situation**. Y can complete without X ever happening.

In general, extend relationship makes the use cases difficult to understand. It is suggested that developers use this relationship sparingly.

2. Class Diagrams

Class diagrams are used in both the **analysis** and the **design** phases. During the analysis phase, a very high-level conceptual design is created. At this time, a class diagram might be created with only the class names shown or possibly some pseudo code-like phrases may be added to describe the responsibilities of the class. The class diagram created during the analysis phase is used to describe the classes and relationships in the problem domain, but it does not suggest how the system is implemented. By the end of the design phase, class diagrams that describe how the system to be implemented should be developed. The class diagram created after the design phase has detailed implementation information, including the class names, the methods and attributes of the classes, and the relationships among classes.

The class diagram describes the types of objects in a system and the various kinds of static relationships that exist among them (Bruegge and Dutoit, 2000). In UML, a class is represented by a rectangle with one or more horizontal compartments. The upper compartment holds the name of the class. The name of the class is the only required field in a class diagram. By convention, the class name starts with a capital letter. The (optional) center compartment of the class rectangle holds the list of the class attributes/data members, and the (optional) lower compartment holds the list of operations/methods.

The complete UML notation for a class is shown in Figure 6.

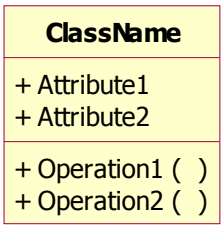

Figure 6: UML notation for a class

2.1 Static Relationships

There are two principle types of static relationships between classes: inheritance and association. The relationships between classes are drawn on class diagram by various lines and arrows.

Inheritance (termed "*generalization*" for class diagrams) is represented with an empty arrow, pointing from the subclass to the superclass, as shown in Figure 7. In this figure, UtilityCell inherits from Cell (a.k.a UtilityCell "is-a" specialized version of a Cell). The subclass (UtilityCell) inherits all the methods and attributes of the superclass (Cell) and may override inherited methods.

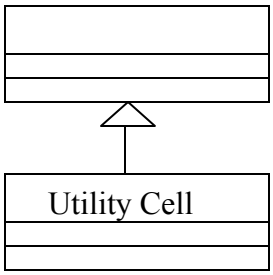

Figure 7: Generalization

An association represents a relationship between two instances of classes. An association between two classes is shown by a line joining the two classes. Association indicates that one class utilizes an attribute or methods of another class. If there is no arrow on the line, the association is taken to be bi-directional, that is, both classes hold information about the other class. A unidirectional association is indicated by an arrow pointing from the

object which holds to the object that is held. There are two different specialized types of association relationships: aggregation, and composition.

If the association conveys the information that one object is part of another object, but their lifetimes are independent (they could exist independently), this relationship is called *aggregation*. For example, we may say that "a Department contains a set of Employees," or that "a Faculty contains a set of Teachers." Where generalization can be though of as an "is-a" relationship, aggregation is often thought of as a "has-a" relationship – "a Department 'has-a' Employee." Aggregation is implemented by means of one class having an attribute whose type is in included class (the Department class has an attribute whose type is Employee).

Aggregation is stronger than association due to the special nature of the "has-a" relationship. Aggregation is unidirectional: there is a container and one or more contained objects. An aggregation relationship is indicated by placing a white diamond at the end of the association next to the aggregate class, as shown in Figure 8.

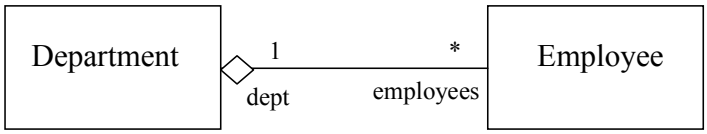

Figure 8: Aggregation

Even stronger than aggregation is *composition.* There is composition when an object is contained in another object, and it can exist only as long as the container exists and it only exists for the benefit of the container. Examples of composition are the relationship Invoice-InvoiceLine, and Drawing-Figure. An invoice line can exist only inside an invoice, and a specific geometric figure only inside a drawing (in the context of a graphic editor). Any deletion of the whole (Invoice) is considered to cascade to all the parts (the InvoiceLine's are deleted).

Composition is shown by a black diamond on the end of association next to the composite class, as shown in Figure 9. In this figure, we show also the fact that the relationship between a Gameboard and its Cells can be navigated only from Gameboard to Cell (an arrow points from Gameboard to Cell). Therefore, this relationship is a composition, and not an aggregation.

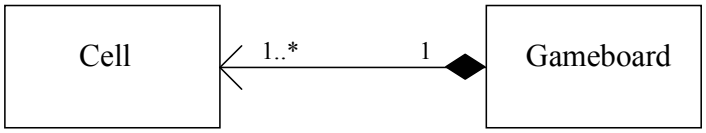

Figure 9: Composition

To summarize – aggregation is a special form of association; composition is a stronger form of aggregation. Both aggregation and composition are a part-whole hierarchy.

2.2 Attributes and Operations

Attributes or data members are shown in the middle box of the class diagram. It is optional to show the attributes. When an attribute is included, it is possible to only specify the name of the attribute. UML notation also allows showing their type (the class of the data type of the attribute), their default value, and their visibility with respect to access from outside the class. Public attributes are denoted with a + sign, protected with a # sign, and private with a -, as shown in Figure 10. The UML syntax for an attribute is:

visibility name : type = defaultValue

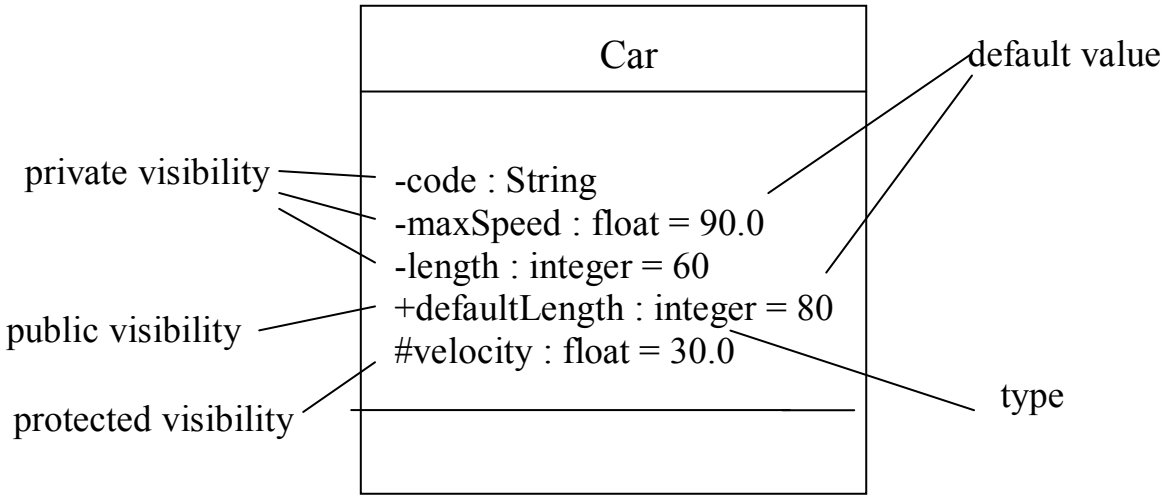

Figure 10: Notation for attributes

The third and bottom compartment of class symbol in UML notation holds a list of class operations or methods. The operations are the services that a class is responsible for carrying out. They may be specified giving their signature (the names and types of their arguments/parameters), the return type, and their visibility (private, protected, public) may be shown. An optional property string indicates property values that apply to the operation. UML notation for operations/methods is shown in Figure 11. The UML syntax for an operation is:

visibility name(parameter-list) : return-type{property string}

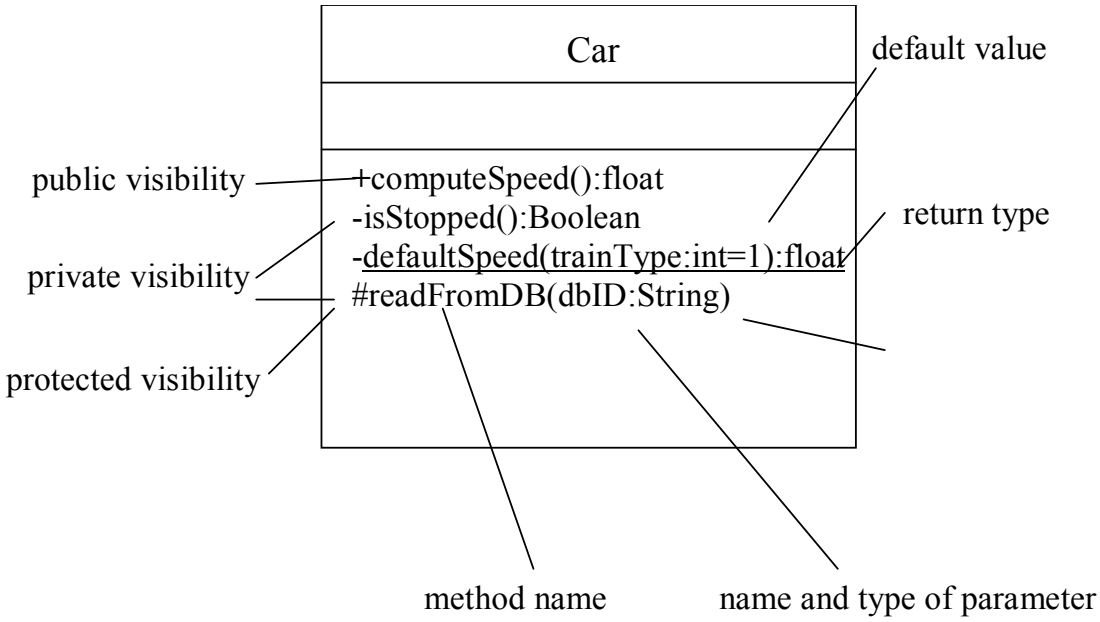

Figure 11: UML notation for operations/methods

2.3 Multiplicity

Associations have a multiplicity (sometimes called cardinality) that indicates how many objects of each class can legitimately be involved in a given relationship. Multiplicity is expressed by the "*n..m*" symbol put near to the association line, close to the class whose multiplicity in the association we want to show. Here "*n*" refers to the minimum number of class instances that may be involved in the association, and "*m*" to the maximum number of such instances. If $n = m$, only an "*n*" is shown. An optional relationship is expressed by writing "0" as the minimum number. Table 1 shows the most common cases of multiplicity.

Table 1: Multiplicity notation

Cardinality and modality	UML symbol
One-to-one and mandatory	1
One-to-one and optional	0..1
One-to-many and mandatory	1..*
One-to-many and optional	*
With lower bound *l* and upper bound *u*	l..u
With lower bound *l* and no upper bound	l..*

We demonstrate several of the aspects of association and multiplicity in Figure 12 .

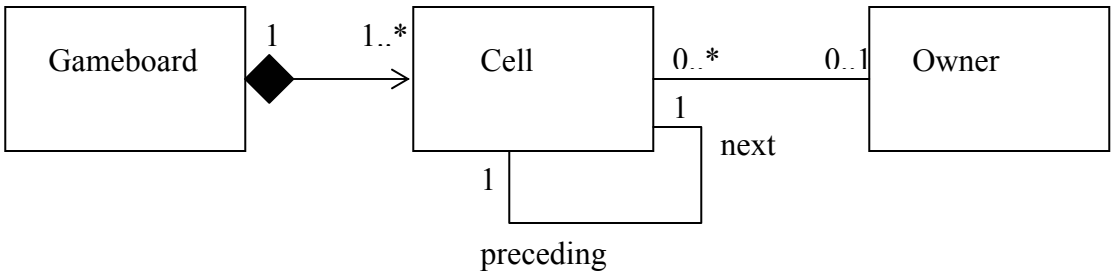

Figure 12: An UML class diagram with three classes, their associations, and multiplicity

Table 2 summarizes the associations between these three classes. Notice the "next" and "preceding" labels on the Cells association. These are called "roles." Labeling the end of associations with role names allows us to distinguish multiple associations originated from a class and clarify the purpose of the association. (Bruegge and Dutoit, 2000)

Table 2: Details about the associations of Figure 12

Classes of association	Kind	Information held
Gameboard, Cell	Composition	A gameboard contains one or more cells. A cell is contained in one and only one gameboard. The gameboard can access its sections but the cells do not need to access their gameboard. The cells cannot exist in isolation, but only if contained by a gameboard.
Cell, Cell	Association	Every Cell is associated with, and must be able to access, its *next* Cell and its *preceding* Cell, along the Gameboard.
Cell, Player	Association	A Cell is owned by zero or more Owners. An Owner owns zero or more Cells. The Cell can access its Owner, and the Owner can access the Cells it owns.

2.4 More Advanced Class Diagram Concepts

The prior sections on class diagram provided you with most of the information you will need to create complete diagrams. There are a few more aspects that you might find helpful for some more advanced diagrams.

2.4.1 Abstract Classes

If you have an abstract class or method, the UML convention is to italicize the name of the abstract item. You can also label the item with {abstract}.

2.4.2 Packages

If a system is big, it should be partitioned in smaller sub-systems, each with its own class diagram. In UML notation, the partitions/sub-systems are called *packages*. A package is a grouping of model elements, and as such it is a UML construct used also in other UML diagrams. Packages themselves may be nested within other packages. A package may contain both subordinate packages and ordinary elements of the class diagram, although it is not usually a good idea to mix in the same diagram packages and classes.

The symbol of two collapsed packages is shown in Figure 13. The name of the package is placed within the large rectangle. A collapsed package does not show its contents (which classes are contained in the package) and are used in a higher-level system diagram that shows all packages composing the system and their dependencies. A package depends on another package if at least one of its classes depends on the classes of the latter package.

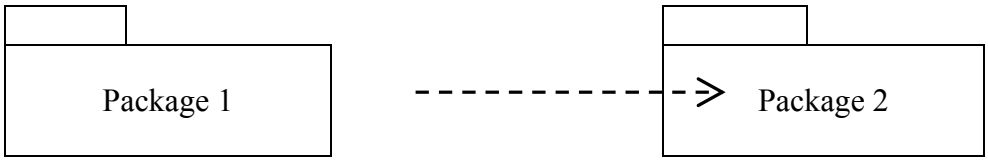

Figure 13: UML notation for two collapsed packages with a dependency relationship

A package may also be drawn showing its contents. In this case, its name is placed in the small rectangle on the upper-left side, while a UML class diagram, showing the classes or the packages contained in it is shown in Figure 14.

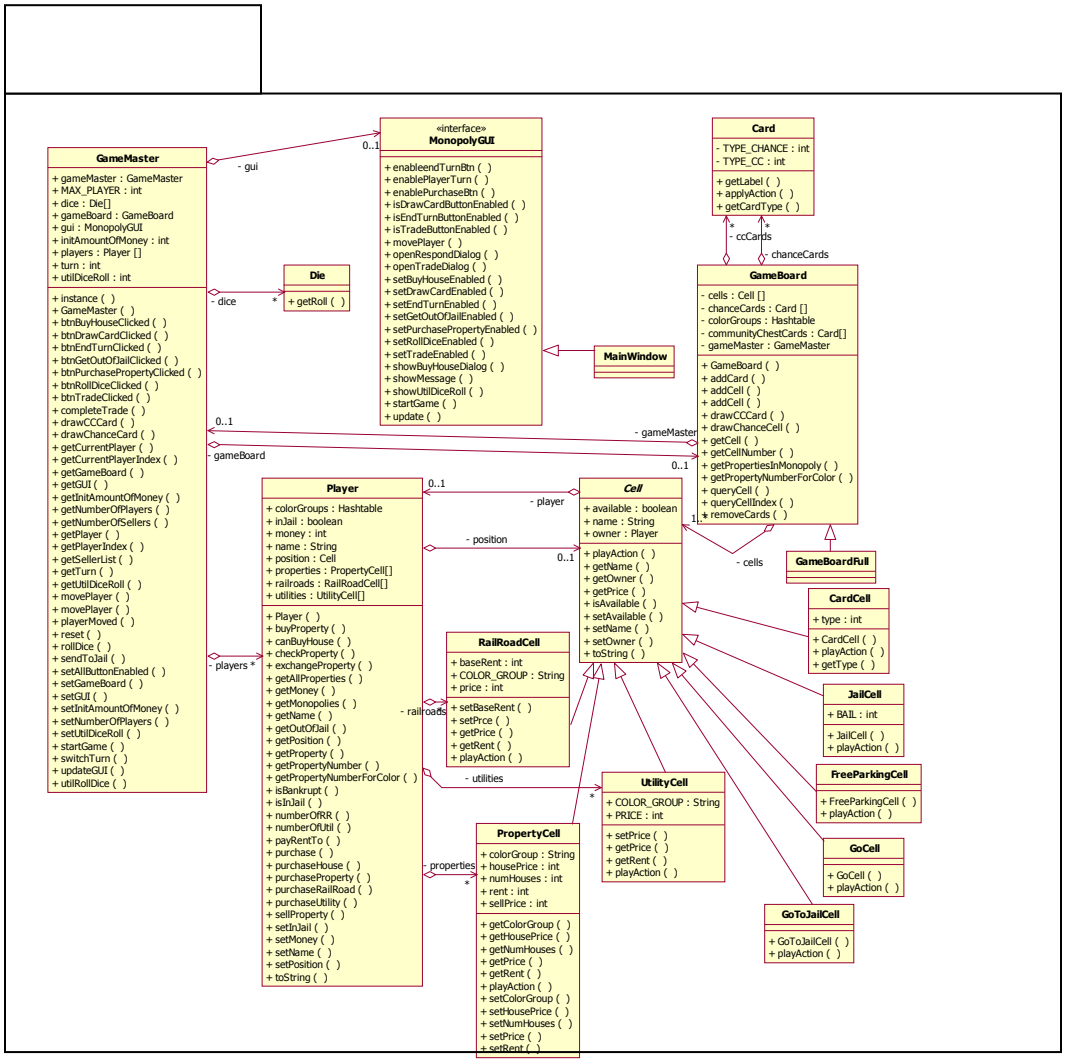

Figure 14: A (non-collapsed) package diagram

2.4.3 Stereotypes

Stereotypes are a high-level classification of an object that gives you some indication of the kind of object it is. Classes can be grouped under *stereotypes*, whose name is written between matched guillemots (« »), over the class name. Stereotypes can also be shown with specific icons. All model elements can have stereotypes. For example, common class stereotypes are:

- «control», a class, an object of which denotes an entity that controls interactions between a collection of objects;
- «entity», a class that represents a domain-specific situation or a real-world object and that does not initiate interactions; and
- «boundary», a class that lies on the periphery of a system but within it.

Other stereotypes can be defined by the team within the context of the system to be developed.

2.4.4 Notes

The class diagram may also include a *note* which is represented as a rectangle with a "bent corner" in the upper right corner. Notes are used to "attach" comments and constraints to the model elements. Notes may appear on any UML diagram and may be attached to zero or more modeling elements by dashed lines. Notes have no impact on the model.

2.5 Object Diagrams

UML class diagrams show the classes of the system, their data structure, their relationships and their interfaces. Ideally, a full UML class diagram show all system classes, although for practical reasons they are usually partitioned in many class diagrams, referring to various packages. A UML object diagram, on the other hand, shows a snapshot of the detailed state of a system at a point in time. A UML object diagram shows some specific instance of the classes of the system. While there is only class diagram of the system, there may be hundreds of different object diagrams. In an object diagram, many different instances of the same class, and no instance of other classes, may be shown.

Figure 15 shows UML notation for an object. The notation is similar to the class notation, with three key differences:

- The name of the object is underlined, and is followed by its class name, separated by a colon. Often, there is no need to explicitly name a class. In this case, only the colon and the object name are written in the rectangle.
- The attribute compartment may hold a list with the values of relevant attributes of the object.
- There is no operation compartment.

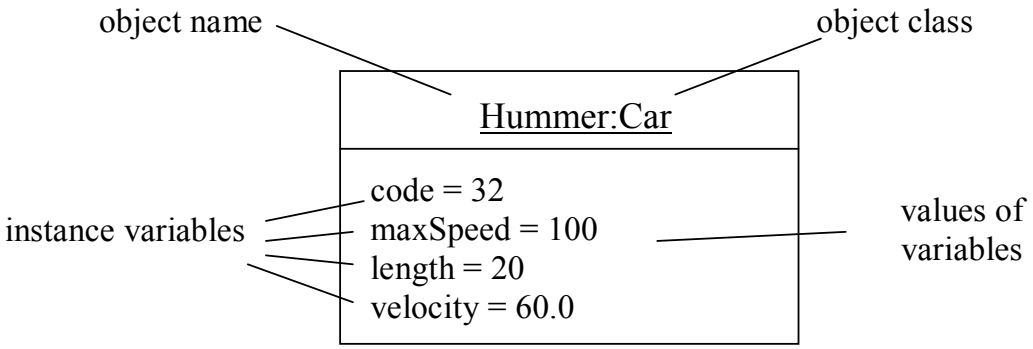

Figure 15: UML notation for an object

In object diagrams, the associations among objects are shown as *links*. A binary link is shown as a path between two objects. In the case of a reflexive association, it may involve a loop with a single object.

A role name may be shown at each end of the link. An association name may be shown near the path; if present, it is underlined to indicate an instance. Multiplicity is not shown for links because they are instances. Other association adornments (aggregation, composition, navigation) may be shown on the link roles. A sample object diagram is shown in Figure 16.

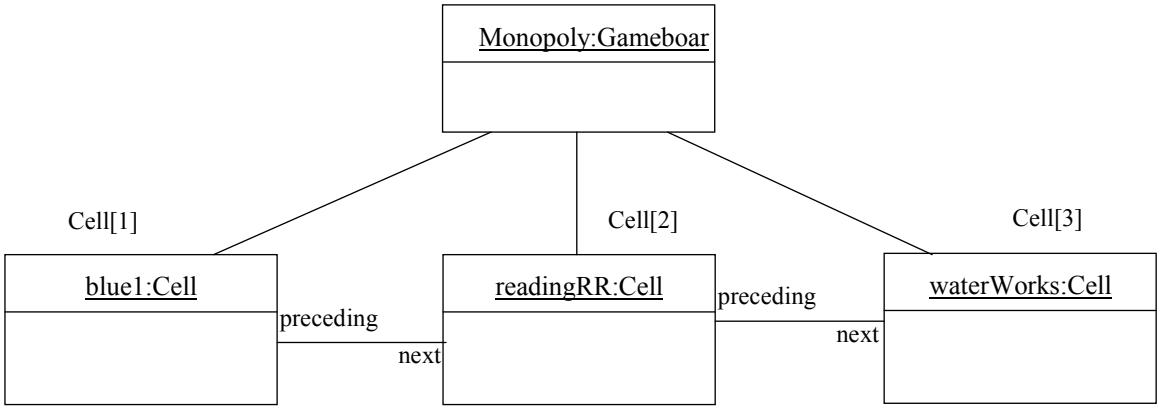

Figure 16: Object diagram with a Monopoly Gameboard and some specific Cells

3 Sequence Diagrams

Sequence diagrams are used in the **analysis** and **design** phases. Sequence diagrams are often used to depict the chronologically-structured event flow through a use case. By creating a sequence diagram, the objects that participate in the use case are identified. Additionally, pieces of the use case behavior are assigned to objects in the form of services. The process of creating a sequence diagram often results in the refinement of the use case, potentially identifying missing but desired behaviors.

Sequence diagrams represent a system behavior based upon the needed interactions among a set of objects in terms of the messages that exchange among them to produce the desired result. Sequence diagrams highlight the sequence of messages through time. However, they do not show how objects are linked and may send messages to each other.

In a sequence diagram, objects are shown in columns, with their object symbol on the top of the line. Similar to the class diagram, the object name appears in a rectangle. If a class name is specified, it appears before the colon. The object name always appears after a colon (even if no class name is specified). If an external actor (see the preceding Use Case Diagram section above) initiates any interaction, the stick figure can be used rather than a rectangle.

A sequence diagram has two dimensions: the vertical dimension represents time; the horizontal dimension represents different objects. Initiation of the sequence starts in the top-left corner, and time proceeds down the page (from top to bottom). The vertical line is called the object's lifeline. There is no significance to the horizontal ordering of the objects.

A message sent from one object to another is shown as an arrow from the line of the sender to the line of the receiver. Each message is labeled at a minimum with message name. You can optionally include the arguments containing information that needs to be passed with the message. The reception of a message triggers a corresponding operation to execute. During this execution, other messages may be sent to other objects, and eventually the methods end. An object may send a message to itself. This is shown by an arrow from the object line to the same line. The method execution is represented in the sequence diagram by a thickening of the object line.

Figure 17 shows an example of a player taking a turn in Monopoly. Most sequence diagrams are concrete and represent one scenario. A scenario is a sequence of actions that illustrates behavior. A scenario may be used to illustrate an interaction or the execution of a use case instance. (Rumbaugh, Jacobson et al., 1999)

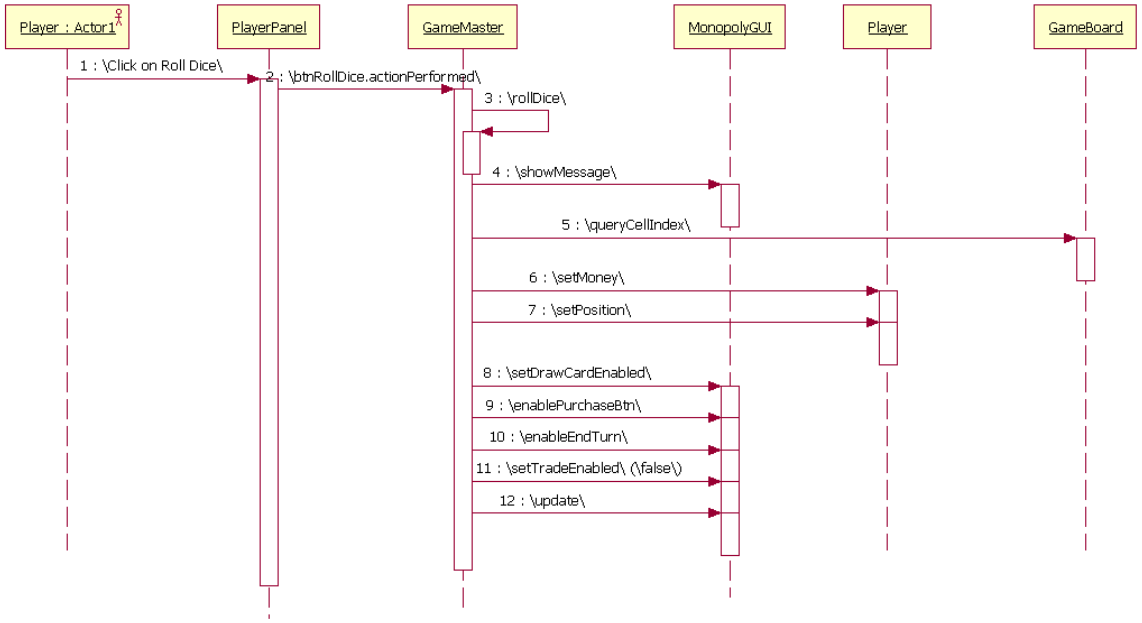

Figure 17: A sequence diagram representing a player taking a turn

4 State Diagrams

State diagrams are created during the **analysis** and **design** phase to describe the behavior of nontrivial objects. State diagrams are good for describing the behavior of one object across several use cases and are used to identify object attributes and to refine the behavior description of an object.

A state is a condition in which an object can be at some point during its lifetime, for some finite period of time (Scott, 2001). State diagrams describe all the possible states a particular object can get into and how the objects state changes as a result of external events that reach the object. (Fowler, 2000) In this section, we'll present instead the notation for state diagrams that was first introduced by Harel (Harel, 1987), and then adopted by UML. In a state diagram:

- A state is represented by a rounded rectangle.
- A start state is represented by a solid circle.
- A final state is represented by a solid circle with another open circle around it.
- A transition is a change of an object from one state (the source state) to another (the target state) triggered by events, conditions, or time. Transitions are represented by an arrow connecting two states.

Figure 18 shows the state diagram of a turn in Monopoly. In a state diagram, when a transition has no event within its label (such as leading out of InJail and into EndTurn), it means the event is triggered as soon as any activity associated with the state is complete. This is called a triggerless transition. Transitions can also be labeled with guards (a Boolean expression which evaluates to true or false) inside square brackets, such as [trade accepted]. A guarded transition occurs only if the guard resolves to true. Only one transition can be taken out of a given state. If more than one guard condition is true, only one transition will fire. The choice of transition to fire is nondeterministic if no priority rule is given (Rumbaugh, Jacobson et al., 1999).

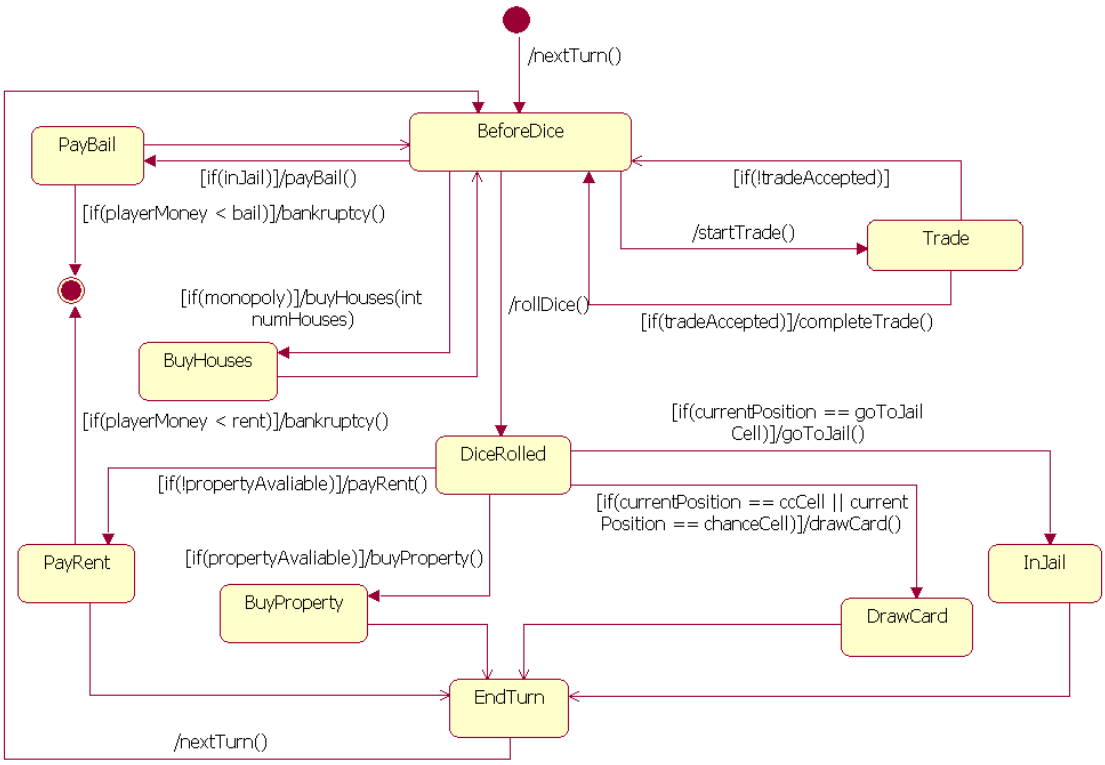

Figure 18: UML State Diagram for a Turn in Monopoly

5 Activity Diagrams

Activity diagrams are used during the **design** phase of complex methods. Alternately, the activity diagram can also be used during **analysis** to break down the complex flow of a use case. Through an activity diagram, the designer/analyst specifies the essential sequencing rules the method or use case has to follow.

UML activity diagrams are an updated and enhanced form of flowcharts; the main enhancement over flowcharts is the ability to handle parallelism, as will be discussed. An activity diagram is a variation of a state chart, discussed in the prior section, in which the states are activities representing the performance of operations and the transitions are triggered by the unconditional completion of the operations. An activity is a single step that needs to be done, whether by a human or a computer (Fowler, 2000). Incoming transitions (an incoming arrow) trigger the activity. If there are several incoming transitions, any of these can trigger the activity independent of the others. (Oestereich, 2001)

Figure 19 shows an activity diagram for preparing corn on the cob. The symbols used in the diagram are the same as those used in state diagrams with the addition of the decision symbol and the synchronization bar. The symbol for a decision is the diamond shape,

with one or more incoming arrows and with two or more outgoing arrows, each labeled by a distinct guard condition. A guard is a Boolean, logical expression that evaluates to "true" or "false." All possible outcomes should appear on one of the outgoing transitions.

The synchronization bar indicates that progress cannot proceed past the bar until all activities leading up to the bar have completed (the outbound trigger occurs only when all inbound triggers have occurred). The synchronization bar allows the activity diagram to be able to be used for concurrent programs. The designer can lay out the threads and when they need to synchronize.

Additionally, activity diagrams allow for parallelism, when the order of the ensuing activities is irrelevant (they can run consecutively, simultaneously, or alternately). For example in Figure 19, after the corn is boiled and the butter is melted, two things happen in parallel (the salt and the butter are put on the corn).

In the case that there is more than one possible final states, the various final states should be labeled with a name.

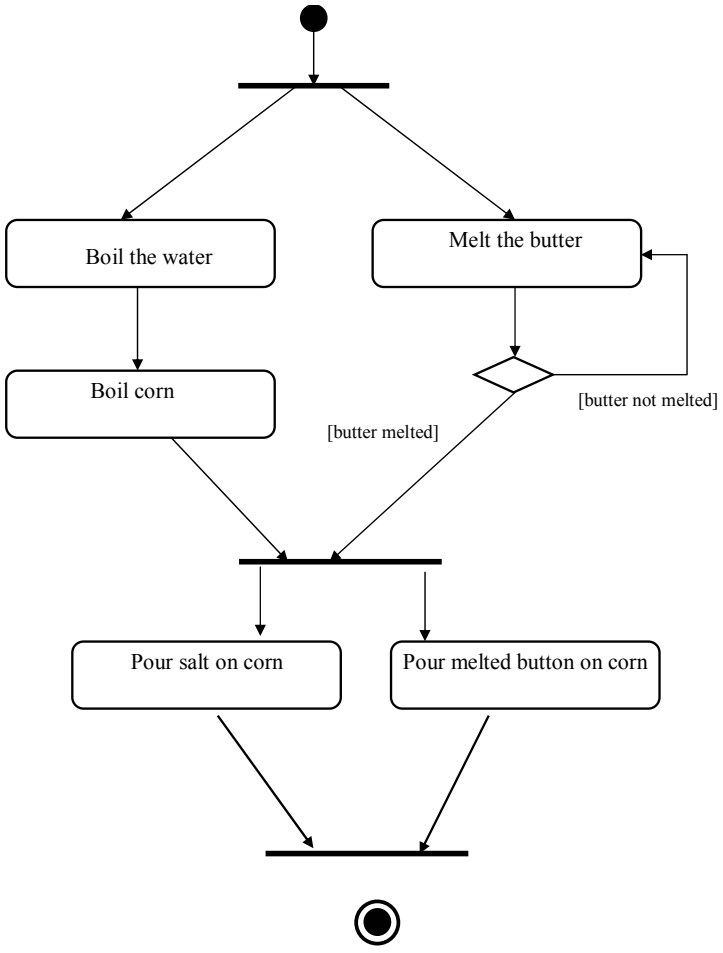

Figure 19: An activity diagram explaining how to prepare corn on the cob

We can use swimlanes in activity diagrams to specify "who" does what (where the "who" could be a particular role or a particular class). To use swimlanes, you must arrange your activity diagrams into vertical zones separated by dashed lines, as shown in Figure 20. The swimlanes indicate that "corn operator" is in charge of preparing the corn and putting the salt on. "Butter expert" melts the butter and pours it on the corn. Swimlanes are good in that they combine the activity diagram's depiction of logic and assign responsibility, as does the sequence diagram.

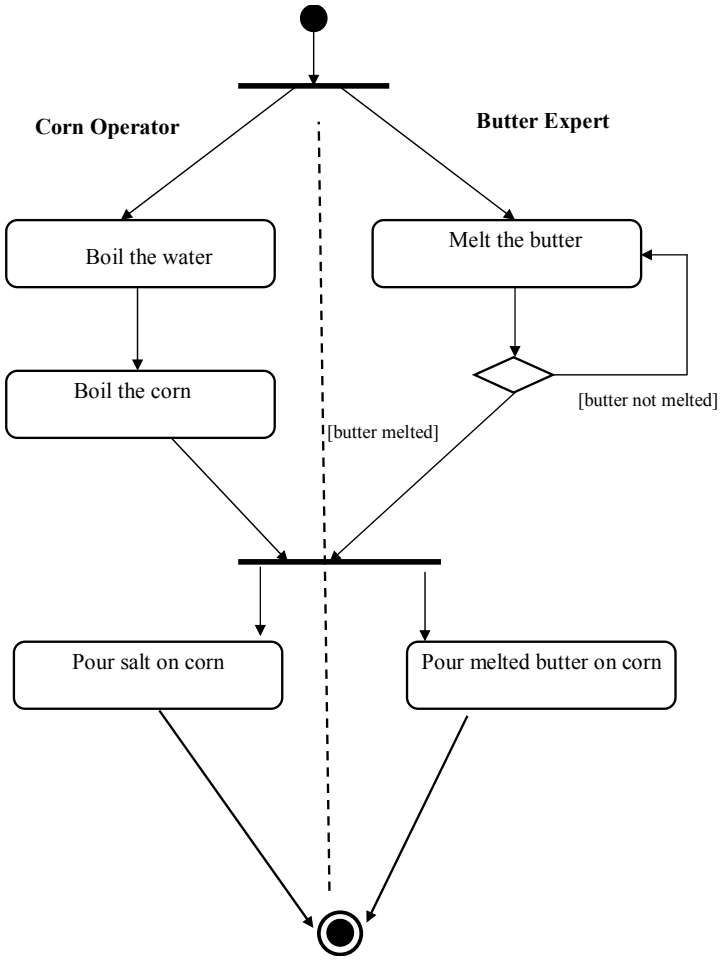

Figure 20: An activity diagram with swimlanes

References

Bruegge, B. and A. H. Dutoit (2000). Object-Oriented Software Engineering: Conquering Complex and Changing Systems. Upper Saddle River, NJ, Prentice Hall.

Fowler, M. (2000). UML Distilled. Reading, Massachusetts, Addison Wesley.

Harel, D. (1987). "Statecharts: A visual formalism for complex systems." Science of Computer Programming: 231-274.

Jacobson, I., M. Christerson, et al. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach. Wokingham, England, Addison-Wesley.

Oestereich, B. (2001). Developing Software with UML: Object-Oriented Analysis and Design in Practice. London, Person Education.

Rosenberg, D. and K. Scott (1999). Use Case Driven Object Modeling with UML: A Practical Approach. Reading, Massachusetts, Addison-Wesley.

Rumbaugh, J., I. Jacobson, et al. (1999). The Unified Modeling Language Reference Manual. Boston, Addison Wesley.

Scott, K. (2001). UML Explained. Boston, Massachusetts, Addison-Wesley.

