Section V: Object Oriented Programming

Risk Management

In theory, there is no difference between theory and practice. But, in practice, there is.

- Jan van de Snepscheut

Chapter 21: Unified Modeling Language

As we hope you have been able to see over the last several chapters, object oriented programming allows us to break down computer programs into separate objects in a very intuitive way. If you were new to programming when you first opened this book, you may very well have started with Chapter 3, fired up VB.NET and started to code. While this approach might work for simple programs, it will certainly not work for larger ones. For example, what if you were asked to create a large Value-at-Risk system to monitor several automated trading systems. A project of this magnitude is too big to immediately start programming. Clearly, a good bit of planning would be required first.

In order to create larger applications, we should follow a detailed planning process for program design. This process must include a comprehensive analysis of the project requirements and result in a design, or blueprint of the objects to be used in the program, as well as a plan for project completion. As you will no doubt learn over your career as a financial engineer, quality time spent on planning will save countless hours of coding and may even prevent failure of projects.

Large software projects have large probabilities of failure. Very rarely, if ever, do large software applications meet all of the requirements as planned on time and within budget. Proper planning is the only way to ensure against failure before you start to program. Furthermore, it is not enough just to plan; be sure to have your designs and plans approved by management before you build anything.

As described in Chapter 2, the Kumiega-Van Vliet Trading System Development Methodology requires that we build an Objects and Program Document as well as gain management buy-in prior to programming. This document should lay out all the objects along with all their functionalities that will be needed to construct the system. Again, the process should include requirements analysis, to define the specifications of the software; object oriented analysis to provide a framework within which all the objects can cooperate to satisfy the requirements; and object oriented design, to lay out the class hierarchy. Fortunately, there is a graphical language created expressly for this purpose--the Unified Modeling Language (UML).

Unified Modeling Language

Although it's also possible to describe a software system and its design in words, most developers prefer to use pictures to help visualize the system's pieces and the relationships between them. UML is a way to represent object-oriented applications using a standard set of graphical notations. With UML we can create blueprints in the form of diagrams before we start to program. Planning with UML makes the entire software development process much more structured and makes it easier to communicate ideas about system architecture. UML is not, however, a project management tool. Project management tools and software coordinate the various parts of a software project into a timeline for completion. UML diagrams show, from an architectural perspective, the objects and the interrelationships between objects in a software application.

We can model just about any object oriented application using UML. By creating models first, we can assure ourselves not only that trading algorithms are completely and correctly formulated, but also that the thorny issues of object oriented implementation are worked out before construction begins and changes become expensive. Blueprints of classes and code modules, drawn either by hand or built in a UML software suite, are much easier to change than existing systems.

There are, in fact, dozens of products available that facilitate creation of UML diagrams. The most well-known UML design tool is Rational Rose (www.rational.com). Using these tools, we can build new applications or analyze existing code to reverse-engineer the UML diagrams. At the extreme end, some software will even go so far as to generate program code from UML diagrams, producing most of a production application.

Large automated trading systems must be structured in a way that facilitates error free execution and a clear architecture so that financial engineers can find and fix bugs quickly. UML helps us visualize trading system design from a technology standpoint and document the results of the modeling process. This visualization is enabled through the use of UML's twelve diagram types, which are defined in three categories—model management, structural and behavior diagrams.

High level, model management diagrams lay out the way we organize and manage the components of an application and consist of:

- Model Diagrams
- Subsystem Diagrams
- Package Diagrams

Structural diagrams are used to model the static structure of a software application and consist of:

- Class Diagrams
- Object Diagrams
- Component Diagrams
- Deployment Diagrams

Behavior diagrams show the different behaviors of objects in an application and consist of:

- Use Case Diagrams
- Sequence Diagrams
- Activity Diagrams
- Collaboration Diagrams
- Statechart Diagrams

Model Management Diagrams

The process of modeling a software application is a process of breaking down a large system into smaller and smaller subsystems, because as systems get larger, it becomes more and more difficult to understand how the pieces fit together.

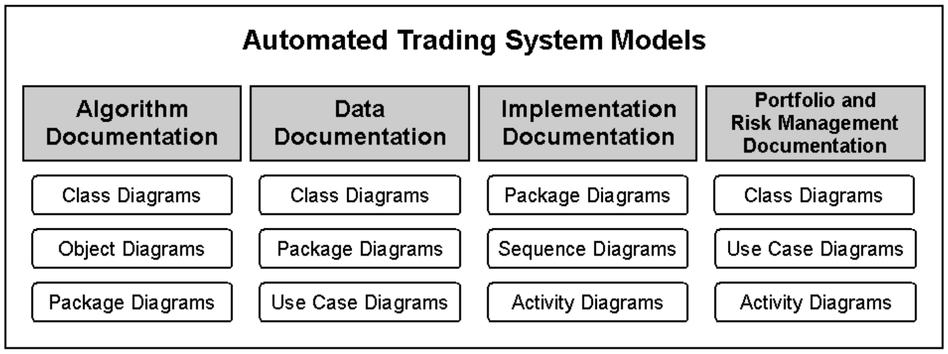
Model management diagrams are high-level designs to illustrate the organization and management of application components. Model management diagrams describe how the different pieces of a UML design will fit together. Subsequent diagrams will refine the details, but for now model management diagrams will incorporate all of the other diagrams we will look at in order to show how the system is structured.

Model Diagrams

Our model diagram follows the Kumiega-Van Vliet paradigm presented in Chapter 2 of this book for developing automated trading systems. Thus, the whole trading system software design process is defined by model corresponding to each of the four steps along the waterfall.

For each of the four models, as an example, subsets of the twelve UML diagrams have been selected to show the relevant areas of communication. In each of the four models, instances of all of the diagram types may be required but nonetheless the focus will be on the diagrams listed.

[21fig01]



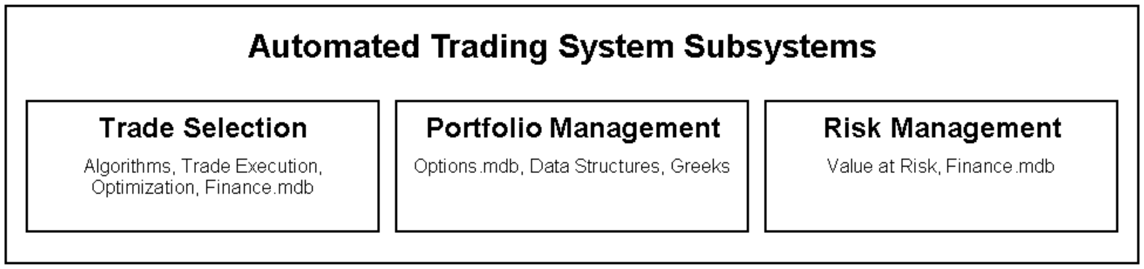
So, as shown in the diagram, modeling of the algorithms for trade selection will concentrate on structural diagrams. Data and implementation models will focus on package diagrams and behavior diagrams. Portfolio and risk management models will focus on structural class diagrams as well as behavior diagrams.

Subsystem Diagrams

Software systems are made up of subsystems. And subsystems are made of packages. A subsystem diagram breaks down a model diagram into the constituent subsystems of a software system and provides a hierarchical view on a system's overall structure.

As we saw in Chapter 2, the implementation of a trading system must manage three concurrent processes—trade selection, portfolio management and risk management. So, our subsystem diagram organizes the models into these logical components. Examples of the pieces of subsystems are shown.

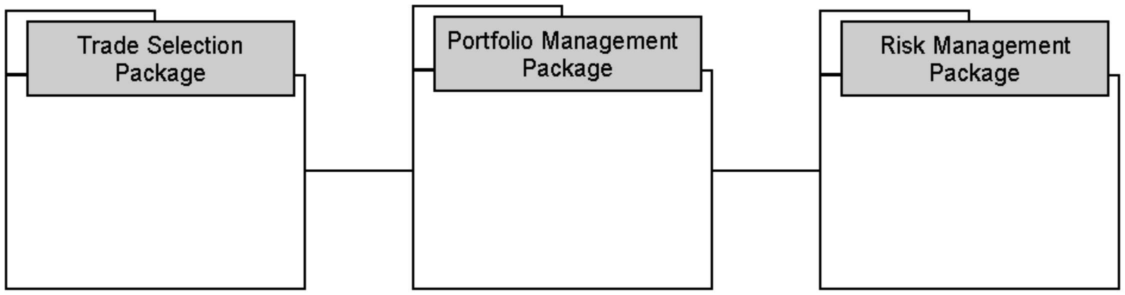
[21fig02]


Automated Trading System Subsystems
Trade Selection
Algorithms, Trade Execution, Optimization, Finance.mdb
Portfolio Management
Options.mdb, Data Structures, Greeks
Risk Management
Value at Risk, Finance.mdb

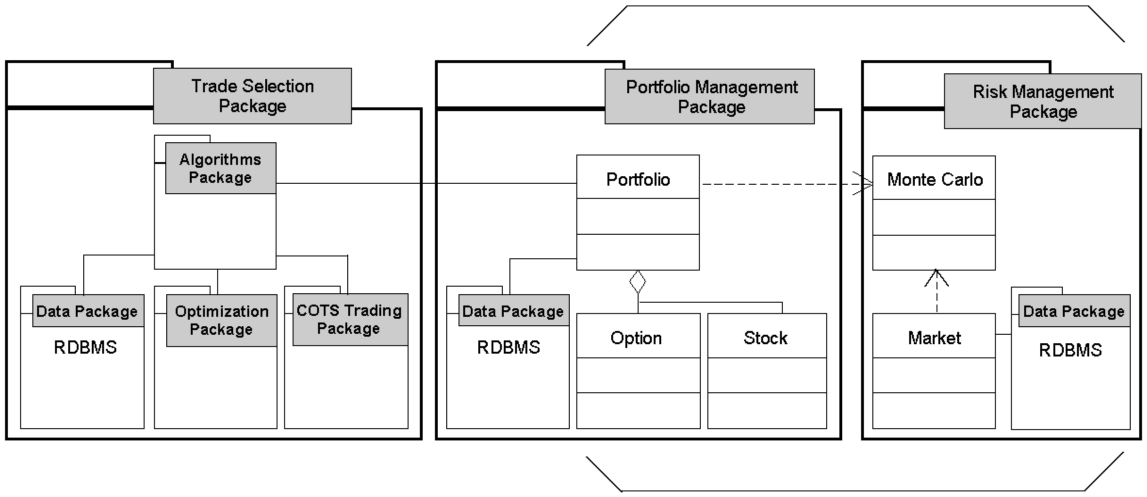
Packages Diagrams

Packages diagrams break subsystems into packages of classes and subpackages. Packages simplify complex class diagrams and group together logically related program elements.

We draw packages as rectangles with small tabs on the top right-hand side. As we will see, lines of different types show relationships between packages. We might say for example that one package has a relationship with another package if changes in one cause changes in the other.



[21fig03]
The three subsystems in our trading system are packages. We can break down these subsystems further, into other subsystems and classes.



[21fig04]
The classes and subpackages in this diagram are connected by relationships to illustrate the fact that classes send messages to one another. We will look at these relationships in greater detail when we examine structural diagrams. One of the arts of UML design is to

minimize the dependencies between classes, which will have the result of reducing the impact of changing a class or package definition.

Over the remainder of this chapter, we will not be able to diagram all of the aspects of a trading system in detail. From the package diagram above, however, and the knowledge gained over the past chapters, you should be able to piece together the elements and subsystems of a full trading system. For this chapter, though, we will focus on a simplified project implementing portfolio and risk management using Value at Risk and UML. The project will encompass the class and packages defined by the brackets in the previous graphic. This project will require that we define the packages and objects we will need to build the application as well as specifications as to how these objects will interact with each other. The Value-at-Risk application will present a subset of the features of UML, but will give you an understanding of the steps necessary to create an Objects and Program Document with UML.

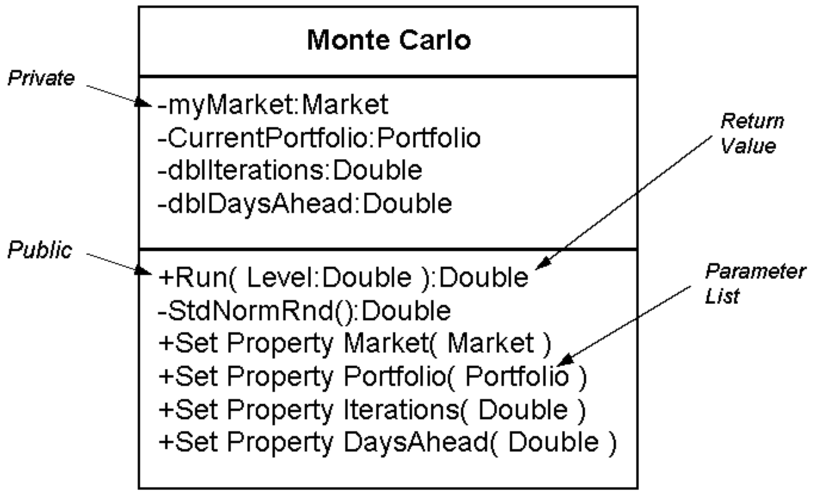
Structural Diagrams

Structural diagrams show the static architecture of a software project.

Class Diagram

A full class diagram displays an overview of an entire system including the constituent classes and the relationships between them. However, class diagrams are static and only show what relationships exist, but not when they happen.

UML notation for a class is a rectangle with three parts, one each for the class name, the attributes, and the member methods or functions. An individual class is represented in this fashion:


Monte Carlo
Private
-myMarket:Market
-CurrentPortfolio:Portfolio
-dblIterations:Double
-dblDaysAhead:Double
Return Value
Public
+Run(Level:Double):Double
-StdNormRnd():Double
+Set Property Market(Market)
+Set Property Portfolio(Portfolio)
+Set Property Iterations(Double)
+Set Property DaysAhead(Double)
Parameter List

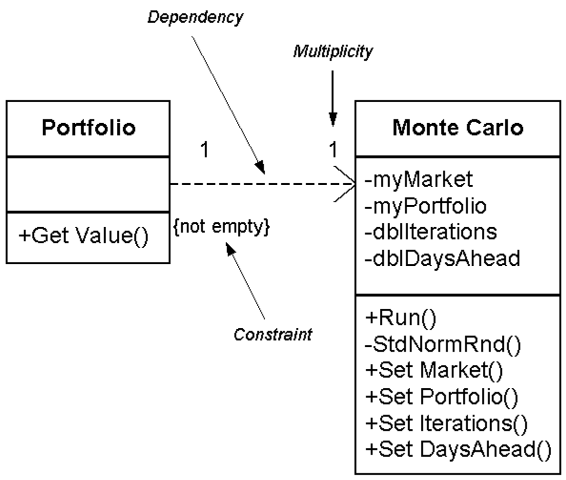
[21fig05]

Here, Monte Carlo is the name of the class and it represents the definition of a Monte Carlo simulation object. The – and + signs define the private or public visibility of the attributes and methods. Although not shown # would define protected visibility. MyMarket, CurrentPortfolio, dblIterations, and dblDaysAhead are all private attributes of the Monte Carlo class. The dblDaysAhead attribute, for example, will hold the time horizon of the simulation in terms of the number of days. DblIterations will hold the number of times the simulation will run.

The member functions are listed in the bottom box and include the property gets and sets.   The signatures of the respective methods are also shown outlining the input and output argument types. The New() method of course is the constructor and StdNormRnd() is the function described in Chapter 5 that returns a standard normal deviate. In this case, the Properties are all WriteOnly and so only Sets are listed.

In addition to the classes themselves, we can also represent in UML the class relationships. Relationships between classes are shown as connecting links and come in five varieties—dependencies, associations, composition, generalization and aggregation. These links should also define the relationship's multiplicity rules, which we will discuss shortly.

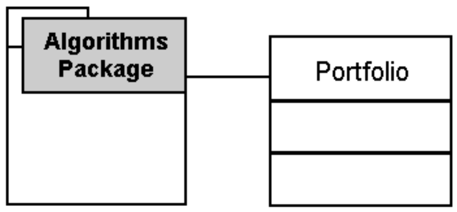
When a class has as a member another class, we say that it depends on that class. This is then a dependency relationship and is drawn as a dotted line with an arrow pointing to the containing class. In our example, the Monte Carlo class depends on the Portfolio class and has a constraint that relationship not be empty. Of course, if there is no portfolio, there is no value at risk to calculate. A constraint, written in braces {}, requires that every implementation satisfy a condition.



[21fig06]

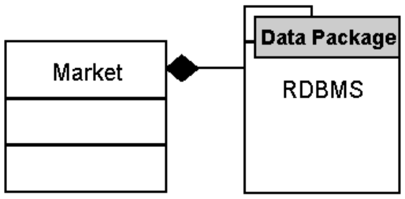
As you can see from this graphic, as we begin to move outward, and take a look at the bigger picture, we may start to abbreviate or even omit details at lower levels.

An association is the most basic relationship and in UML is drawn as a line connecting the two classes.   An association relationship exists between the Portfolio class and the algorithms package.

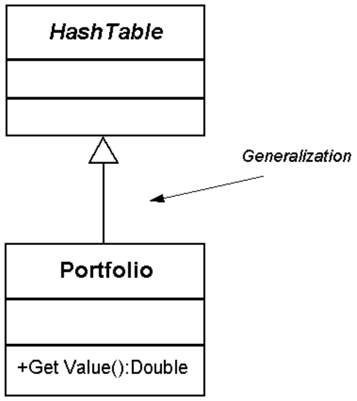


[21fig07]

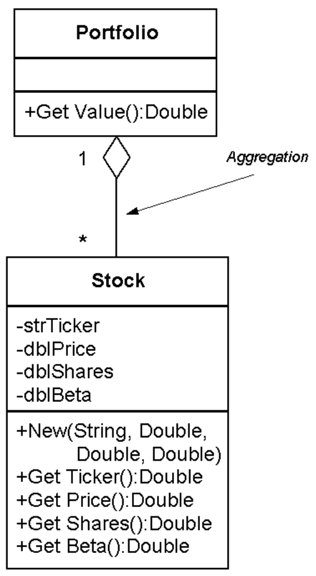
If class exists only as a member of another class, then the relationship is referred to as a composition within the containing class. A composition is drawn as a line with a solid diamond at the containing class end. In our trading system example, the OleDbConnection, OleDbDataAdapter, and DataSet classes, collectively referred to as a Data Package, will exist only as members of the market class.



[21fig08]
A generalization is equivalent to an inheritance relationship and is drawn as a line with a hollow arrow pointing to the base class. Inheritance, or in UML speak generalization, shows that Portfolio is a derived class of Hashtable, of course inheriting all of the attributes and methods of the parent. The Value property has also been added to the class Portfolio.



[21fig09]
An aggregation is a relationship in which several instances of a class belong to a collection class. An aggregation is drawn as a line with a hollow diamond pointing to the collection. In our diagram, an aggregation exists between a Portfolio and Stocks.
[21fig10]

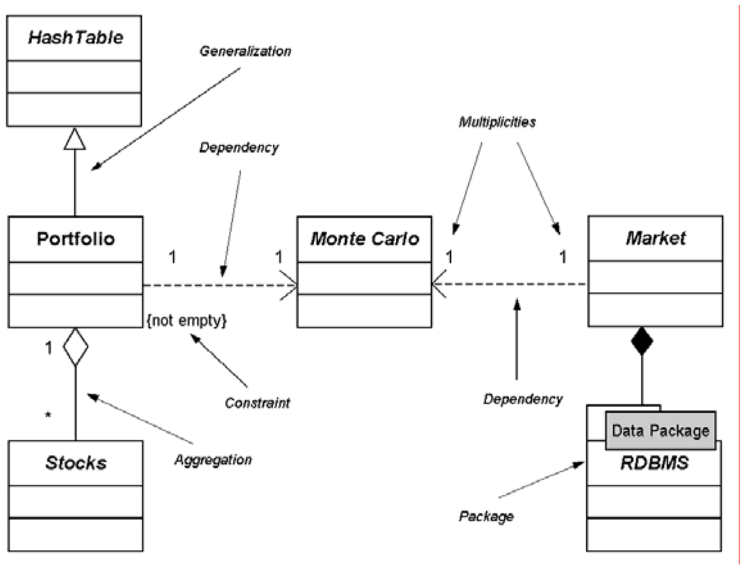


The asterisk near the Stock class and the 1 near Portfolio class represent the multiplicities. A single portfolio can have many stocks. Thus, there is a one-to-many relationship between Portfolio and Stock. Since a Portfolio has Stocks as elements, the diamond is positioned near the Portfolio box. We could also add a StockOption to represent another type of element in a Portfolio.

The multiplicity is the number of instances of a class that may be associated with a single instance of the class at the other end. The following table describes the most common multiplicities.

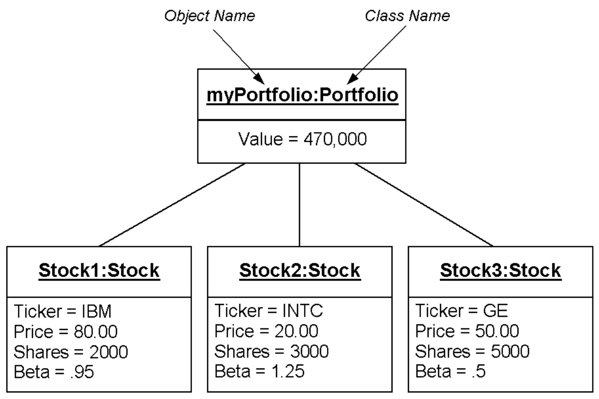
Multiplicity	**Description**
0..1	Zero or one I nstance.
0..* or *	Zero or more instances.
1	One instance.
1..*	One or more instance

The class diagram below models the entire Monte Carlo simulation application we will create later in the chapter. As you can see, the central class is the MonteCarlo class.
[21fig11]



Object Diagram

An object diagram is simply a snapshot of all of the objects at any given point in time. Object diagrams show instances of classes and objects come and go, sometimes rapidly. So object diagrams are useful for explaining very small project pieces with highly complicated relationships, especially recursive ones. The object diagram below instantiates the class diagram, replacing it with a concrete example.
[21fig12]



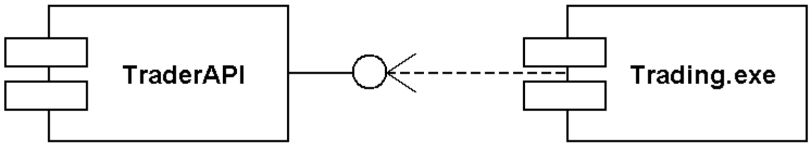
Each rectangle in the object diagram corresponds to a single instance of a class. Instance names are underlined in UML diagrams. Class names are often omitted from object diagrams since the meanings are usually clear.

Component Diagrams

A component diagram describes the physical units of a software system and the dependencies between them. Software components, such as the executable files and library files, are often combined into a single system and as a result have relationships and dependencies between them.

In UML, components are drawn as rectangular boxes, with two smaller rectangles sticking out the left side. Dependencies are dashed lines with arrows pointing from the client component to the supplier component upon which it depends.

[21fig13]

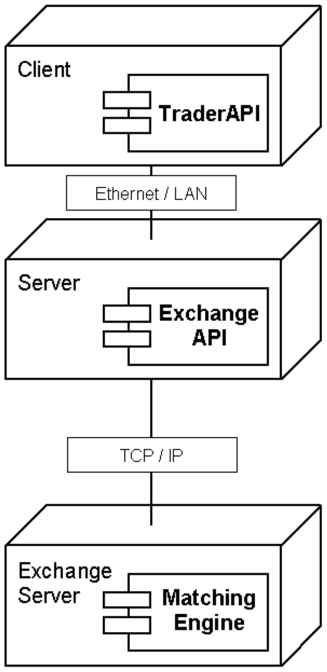


The TraderAPI component contains an interface, shown here as a "lollipop." The dependency relationship within this diagram indicates that the .exe file component refers to services offered by the TraderAPI component via its public interface.

Deployment Diagram

A deployment diagram illustrates the physical organization of hardware in a system. Each node on a deployment diagram represents a hardware unit and communication relationships exist between nodes. Nodes are drawn as three dimensional boxes and contain software components.

Since the VaR model we have been following does not require any Internet or even LAN communication, we will show the hardware structure of an automated order routing system. The deployment diagram shown lays out the communication relationships between the hardware components involved in automated trade entry.



[21fig14]

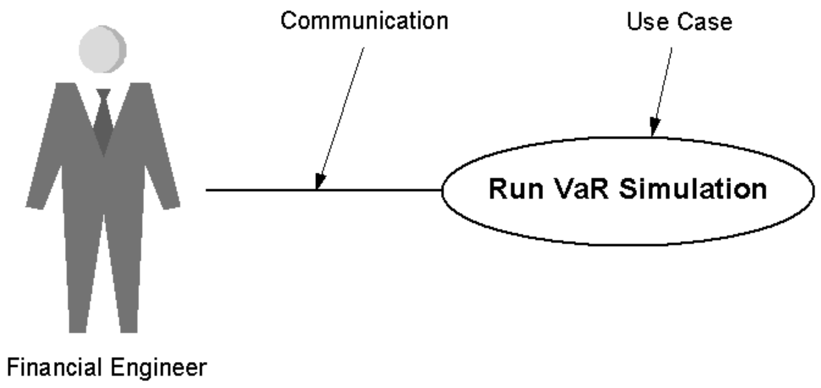
Behavior Diagrams

A behavior diagram represents the different aspects of a system's behavior.

Use Case Diagram

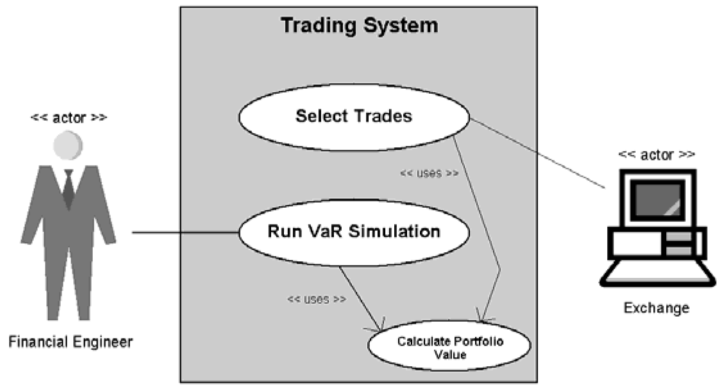
A use case diagram describes from an outside observer's the point of view what a system does, but not how it does it. A use case explains what happens when a hypothetical user or actor interacts with the system. An actor is someone or something that initiates an interaction with the system. Actually, a use case is very much like a scenario or a simple case study where an actor interacts with a system and is provided services by it.

The picture below is a simplified Run VaR Simulation use case. The actor is a financial engineer. The connection between actor and use case is a communication.
[21fig15]



Use case diagrams are helpful in determining system requirements. In fact, new use cases often bring to light new requirements as the system undergoes an evolutionary design cycle and changes are made. Further, their simple, graphical notation facilitates communication.

A simple use case diagram can be expanded with additional features to display more information. The following use case diagram expands the original VaR simulation diagram with additional features for a simplified trading system. In this expanded design, we could include the ability to place trades and populate a portfolio.
[21fig16]



Note again that the Use Case diagram does not represent any sequence, it simply shows the list of scenarios. A system boundary rectangle separates the system from the external actors—the financial engineer and the exchange. The << uses >> relationship links use cases to additional ones, such as in this case Calculate Portfolio Value. Uses relationships like the one shown are especially helpful when the same subtask can be factored out of other use cases. Both Select Trades and Run VaR Simulation use

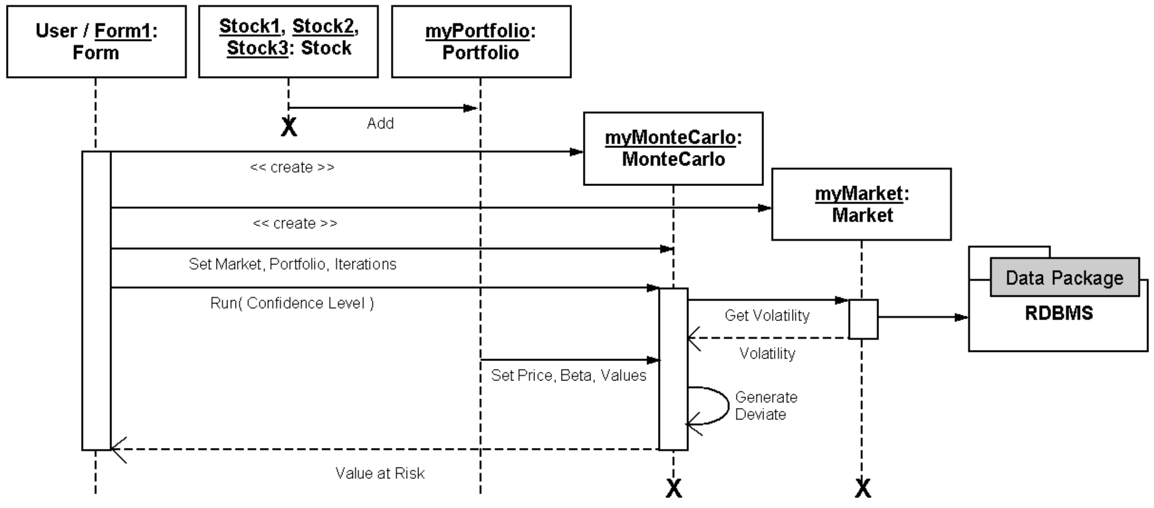
Calculate Portfolio Value as a subtask. In the diagram, the uses relationship is drawn as a line from base use case to the used use case. Calculating the portfolio value is not of the type Run VaR Simulation, but is a task that comprises a piece of the overall run simulation use case.

Although not shown, extend relationships are also possible. Extends indicate that one use case is a version or variation of another use case. Extends are also drawn as lines with an << extend >> label. An extended case can be thought of as a subtype of a use case.

Sequence Diagram

A sequence diagram describes the flow of messages as they are passed from object to object. Whereas class diagrams describes a static structure, sequence diagrams illustrate the nature and timing of interaction between classes.

Below is a sequence diagram for running a MonteCarlo simulation. The object initiating the sequence of messages is a Form1 GUI window. The sequence of events proceeds as we move down the diagram and the objects are displayed from left to right according to when they become part of the sequence. The dotted lines, called lifelines, show that the portfolio exists before the Monte Carlo is run and continues to exist afterwards. On the other had the Monte Carlo object itself and the Market object cease to exist after the simulation is completed, as denoted by the large Xs.



[21fig17]

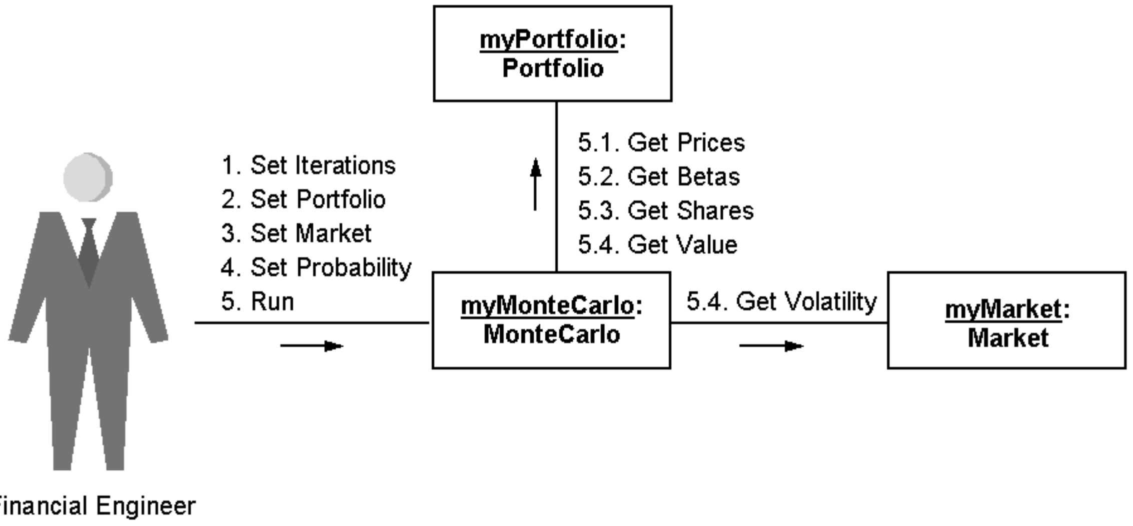
Message calls are represented by arrows from the sender to the receiver's lifeline. The activation bars, the hollow rectangles, represent the length of time of the execution of the message. These bars indicate the scope of a method occurring in a particular object. The dotted lines show return values coming back to the calling object. Notice that myMonteCarlo issues a self call to generate a new random number.

So, to populate the portfolio, Form1 creates stocks and adds them to myPortfolio. A user, presumably a financial engineer, inputs data into the GUI. Form1 creates myMonteCarlo and myMarket. The GUI sends messages to myMonteCarlo pertaining to the parameters of the simulation. Then myMonteCarlo gets the volatility from the market and finally the simulation runs and the Value at Risk number is returned to the GUI.

Collaboration Diagram

In a large software system, objects have to collaborate and so, we have collaboration diagrams. A collaboration in UML-speak is an interaction between two classes. Collaboration diagrams, while conveying the same information as the previous, sequence diagrams, focus on the roles that objects play in the overall scheme, as opposed to the sequence of messages being sent.

Each message in a collaboration diagram has a sequence number. The top-level message is numbered 1. Messages at the same level have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.
[21fig18]



The financial engineer, through the GUI, collaborates with myMonteCarlo by means of a button click and some property sets and the run method. myMonteCarlo collaborates with myPortfolio via three property gets and the value method. And myMonteCarlo collaborates with myMarket by means of the volatility get method.

Statechart Diagram

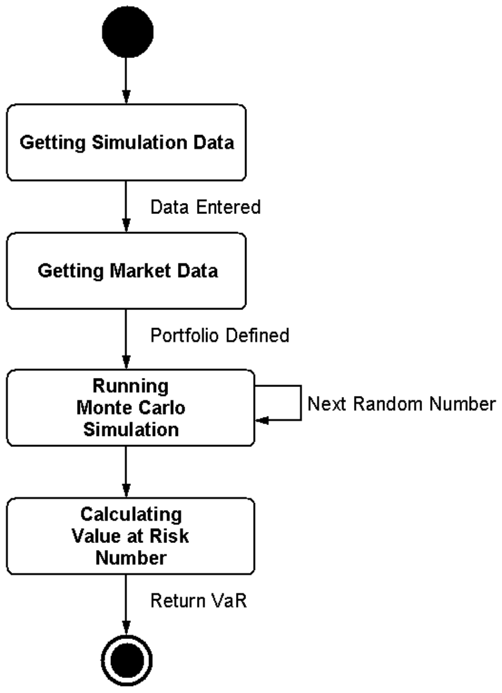
Statechart diagrams allow us to picture the lifecycle of an instance of a class and the timing of external events affecting it. State diagrams consist mainly of two things--states and transitions. Objects have states that depend upon its current activity or condition and transitions, which describe how the object responds to outside influences.

An object performs an activity while in a particular state. Whereas actions are usually though of as processes that are performed quickly, activities take much longer to process and may be interrupted by external events. An event, which could be a button click, a system generated event or even be internally generated, causes a transition or change in the state of an object.

An objects initial state is shown as a black circle. Intermediate states are rounded rectangles, and end state is shown as a black circle with another hollow circle around it.

Transitions are arrows from one state to the next. A description of the event that triggers a transition is usually written beside the transition arrow.

Our example statechart diagram illustrates the states and transitions of the MonteCarlo object. After creation, the object waits while the user enters a valid portfolio and market, a number of iterations and a confidence level. Then the simulation executes. The set up and exeuction can be factored into four non-overlapping states: getting simulation data, getting market volatility data, running the simulation, and calculating Value at Risk.



[21fig19]

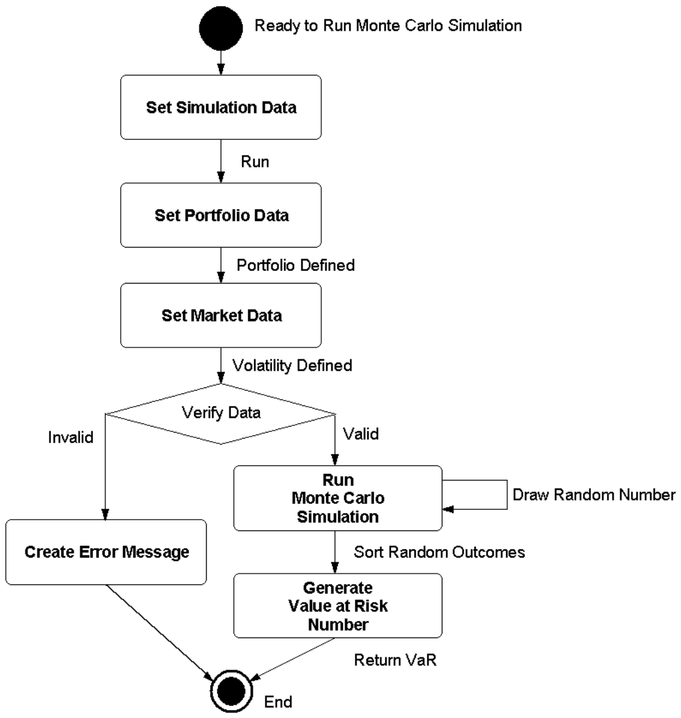
While in its running the simulation state, the Monte Carlo object does not wait for outside events to trigger a transition to the next state. The completion of the running simulation activity causes its transition to the subsequent state.

Activity Diagram

An activity diagram is very much like a flowchart. An activity diagram follows the flow of activities in the order that they occur. Being in an activity state means that an object is doing something. That something could be an event, such as a button click or form load, or the execution of a class method. Unlike a state chart diagram, which concentrates on a single object and its processes, an activity diagram focuses on the process and the flow of activities from object to object. In brief, an activity diagram states the basic sequencing convention the system should follow.

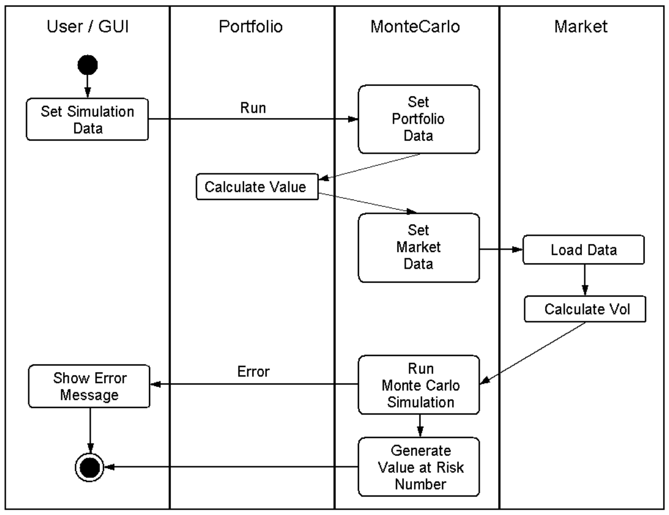
The activity diagram describes the sequence of activities including any conditional or parallel behavior. A condition is shown as a branching in the activity flow. A branch separates a single transition into multiple outgoing transitions. So, if bad data is entered, program flow will proceed down one branch. If the data is good, the other branch will be followed. Either way, the program activity flow merges again later on.

Obviously, since only one of the outgoing transitions can be taken, the conditions are mutually exclusive and a merge marks the end of conditional behavior.
[21fig20]



Some activities can occur at the same time, or in parallel. Although not shown, parallel behaviors are drawn as forks and joins. As with a branch, a fork has one incoming transition and several outgoing transitions. In a fork, however, when the incoming transition is encountered, all of the outgoing streams are taken at the same time. In the end a join occurs when all of the incoming transitions have completed their individual activities.

Activity diagrams are sometimes shown with object lanes, sometimes called swim lanes. Lanes define which object is responsible for which activity.
[21fig21]



Summary

In this chapter, we covered each of the twelve diagrams in UML, the Unified Modeling Language, and applied them to a Monte Carlo simulation for a portfolio of stocks. The chapter example program was built from these diagrams. According to the Kumiega-Van Vliet Trading System Development Methodology, we should build an Objects and Program Document before programming. This document should lay out in UML all the classes, with their attributes and functionalities as well system design and behavior.

Chapter Problems

1.) Describe each of the twelve UML diagrams in your own words.
2.) Explain the three methods described for calculating Value at Risk.
3.) How would you create an Objects and Program Document using UML?
4.) Describe each of the three categories of diagrams.
5.) What is a package?

Project One

The Beta of stock is the covariance of the stock with the market divided by the standard deviation of the market according to the following formula:

$$\beta = \frac{\sigma_{s,m}}{\sigma_m}$$

Create a historical simulation program that uses data in the Finance.mdb database to calculate the Betas. The program should select market returns at random from its distribution of historical ones.

Project Two

Add a connection to TraderAPI.dll and or OptionsAPI.dll so that the user can buy and sell assets and build a portfolio of stocks and options and calculate Value at Risk using Monte Carlo simulation.