Review of UML

The purpose of this chapter is to set the scene by reviewing the key UML concepts, the main diagram types, and the role of those diagrams within the software development process. If you're quite new to UML this will serve as a practical introduction that will help you make sense of the rest of the book, before you move on to further reading. If you're experienced with UML the chapter will serve as handy revision and you might just find some nuggets of information that have so far eluded you.

Either way we'll all be moving on from roughly the same starting point: with the same appreciation of UML notation, with an understanding of relevant software development processes, and with a common bias towards .NET and the Visio for Enterprise Architects tool.

The final point is quite important, and the raison d'être for this book. In recent years the body of UML literature has focused mainly on Java development and the use of modeling tools such as Rational Rose. In this book we're applying a .NET development perspective at the same time as demonstrating the so far under-documented Visio modeling tool that comes bundled with the Visual Studio .NET Enterprise Architect.

With all this in mind we can now press on with the introduction to – or revision of, depending on your background – the Unified Modeling Language.

What is the Unified Modeling Language?

When discussing UML, we need to establish one important point right up front.

> *The **Unified Modeling Language** is a **notation**; that is a set of diagrams and diagram elements that may be arranged to describe the design of a software system. UML is not a **process**, nor is it a **method** comprising a notation and a process.*

In theory you can apply aspects of the notation according to the steps prescribed by any process that you care to choose – traditional **waterfall**, **extreme programming**, **RAD** – but there are processes that have been developed specifically to complement the UML notation. You'll read more about the complementary process(es) later in this chapter.

Why use UML?

Hidden inside that specific question there's a more generic question, which is **"Why use a formal analysis and design notation, UML or otherwise?"** Let's start to answer that question by drawing an analogy.

Suppose you wanted to make a bridge across a small stream. You could just place a plank of wood across from one side to the other, and you could do so on your own. Even if it failed to hold your weight, the only downside would be wet feet.

Now suppose you wanted to make a bridge across a narrow river. You'd need to do some forward planning to estimate what materials you'd need – wood, brick, or metal – and how much of each. You'd need some help, and your helpers would want to know what kind of bridge you're building.

Finally, suppose you wanted to build a bridge across a very wide river. You'd need to do the same kind of forward planning as well a communicating your ideas to a much bigger team. This would be a commercial proposition with payback from fare-paying passengers, so you'd need to liaise with the relevant authorities and comply with health-and-safety requirements. You'd also be required to leave behind sufficient documentation to allow future generations to maintain the structure long into the future.

In a software context, this means that formal design becomes increasingly important as a function of the size and complexity of the project; in particular, as a function of the number of people involved. Based on that analogy, and wider project experience, we could conclude that a formal design notation is important in:

- Establishing a blueprint from the application
- Estimating and planning the time and materials
- Communicating between teams, and within a team
- Documenting the project

Of course, we've probably all encountered projects in which little or no formal design has been done up-front (corresponding with the first three bullet points in that list); in fact more projects than we care to mention! Even in those situations, UML notation has been found to be invaluable in documenting the end result (the last bullet point in that list). Though not recommended, if that's the extent of your commitment to UML you'll be most interested in the Reverse Engineering discussion in Chapter 5.

Now that we've answered the generic question, let's return to the specific question of **why use UML**?

Well it's become something of an **industry standard**, which means that there's a good chance of finding other people who understand it. That's very important in terms of the **communication** and **documentation** bullet points in our list. Also if you or anyone else in the team does not understand it, there's a good chance of finding relevant training courses, or books like this one.

That's very pragmatic reasoning and perhaps more convincing than a more academic (or even commercial) argument such as:

"The application of UML has a proven track record in improving the quality of software systems."

A Brief History of UML

Taking the phrase **Unified Modeling Language** as our starting point, we've discussed in the previous section the "language" (namely, notation) aspect. In the next section, we'll investigate the "modeling" aspect, which leaves us here with the word "unified". What, or who, preceded the UML and how did it all become **unified**? This will become clear as we step through a brief history of UML.

In the beginning although there was a plethora of object-oriented "methods", there were three principal methods:

- The **Booch** method devised by Grady Booch
- **Object Modeling Technique** (OMT) devised by Jim Rumbaugh
- **Object Oriented Software Engineering** (also known as Objectory) devised by Ivar Jacobson

These three methods have many ideas in common, yet different notation for expressing those ideas. Some of you may remember that in an OMT class diagram the classes were represented as rectangular boxes whereas in the Booch method they were represented as stylized cloud shapes. Also, each method placed emphasis on different aspects of object-oriented software development. For example Jacobson introduced the idea of use cases, not addressed by the other methods.

> **In simple terms, a use case is a unit of functionality provided by the system to an actor (such as a user). For example, in a word-processing application one of the use cases might be "Run spell checker".**

The unification of these three methods combined the best bits of each method with a common notation (UML) for the common concepts – the end result being an industry-standard notation for analysis and design. If you speak with anyone who claims to be doing **object modeling**, chances are they'll be using UML.

So how did this unification play out in time? The key dates are:

- OOPSLA '94 – Jim Rumbaugh leaves General Electric to join Grady Booch at Rational Software, so as to merge their methods and achieve standardization across the industry.
- OOPSLA '95 – Booch and Rumbaugh publish version 0.8 of the **Unified Method**. Rational Software buys Objectory and Ivar Jacobson joins the company.
- January 1997 – Booch, Rumbaugh, and Jacobson (**the three amigos**) release – through Rational – a proposal for the UML version 1.0.
- September 1997 – UML version 1.1 is adopted by the Object Management Group (OMG).

The Object Management Group, previously best known for the CORBA standard, is a non-profit organization – comprising many member companies – that encourages, standardizes, and supports the adoption of object technologies across the industry. You can find out more about the OMG at http://www.omg.org.

If we've given the impression that the Unified Modeling Language is the exclusive work of only three contributors, the three amigos, then let's set the record straight. Some of the concepts are based in the early work of other individuals – for example, David Harel's work on Statechart diagrams – and some further enhancements have come from other member organizations of the OMG; for example, the Object Constraint Language (OCL) devised by IBM.

> *OCL was devised so that additional rules could be added to a UML model in a language that less ambiguous than English. For example, the statement "Person.Employer=Person.Manager.Employer" may be less ambiguous than "a person and their manager must both work for the same company."*

More information on OCL can be found at
http://www-3.ibm.com/software/ad/library/standards/ocl.html.

At the time of writing, the UML specification is at version 1.4 and in mid-2001 the OMG members started work on a major upgrade to UML 2.0. Modeling tools – including Visio for Enterprise Architects – will always be one or two steps behind in their support for the specification, but that's not usually a big problem because the core concepts discussed in the next section are now quite mature and stable.

At the time of writing, the version of Visio for Enterprise Architects used in the construction of this chapter provides support for UML at least up to version 1.2 – this can be determined from the *About error checking in the UML model* section of the Microsoft Visio Help**:**

> *"Semantic error checking occurs automatically, noting errors in the design of UML model elements, based on the well-formedness rules in the UML 1.2 specification."*

End-to-End UML Modeling

Having looked at why UML is useful, and where it came from, we'll now look at the notation itself. To cover the complete notation in a single chapter would be impossible, so for a deeper coverage I'll refer you to some other works.

- **Instant UML** by Pierre-Alain Muller (Wrox Press, ISBN 1-86100-087-1).
- **The Unified Modeling Language User Guide** by Grady Booch, James Rumbaugh, and Ivar Jacobson (Addison Wesley, ISBN 0-201-57168-4).
- **UML Distilled** by Martin Fowler with Kendall Scott (Addison Wesley, ISBN 0-201-65783-X).

What we'll do here is cover the essential notation and core concepts that will allow us to progress through the rest of the book with a common understanding.

We'll also aim to address one of the problems of many UML courses and books. The problem being, that all too often the various diagrams are presented in isolation without a clear indication of how they relate to one another. To make matters worse, different examples are often used to demonstrate the different diagrams, not one of those examples being for a system that you might actually want to build. Think here of a **statechart diagram** that describes a motor car gearbox, or a **sequence diagram** that describes the operation of a hotel elevator.

So in the following section we'll have a single example, an Order Processing system, which you should be able to relate to even if you don't intend to build such a thing, and at the end, we'll pull it all together.

UML Essential Notation and Core Concepts

Now we'll step through the UML diagrams in turn, all the way from an **activity diagram** through to a **deployment diagram** in this order:

- Activity Diagram
- Use Case Diagram
- Sequence and Collaboration Diagram
- Statechart Diagram
- Static Structure Diagram
- Component Diagram
- Deployment Diagram

Each diagram is labeled in light gray with some of the names given to the UML elements that are shown, which – for the record – reflects the **UML metamodel**.

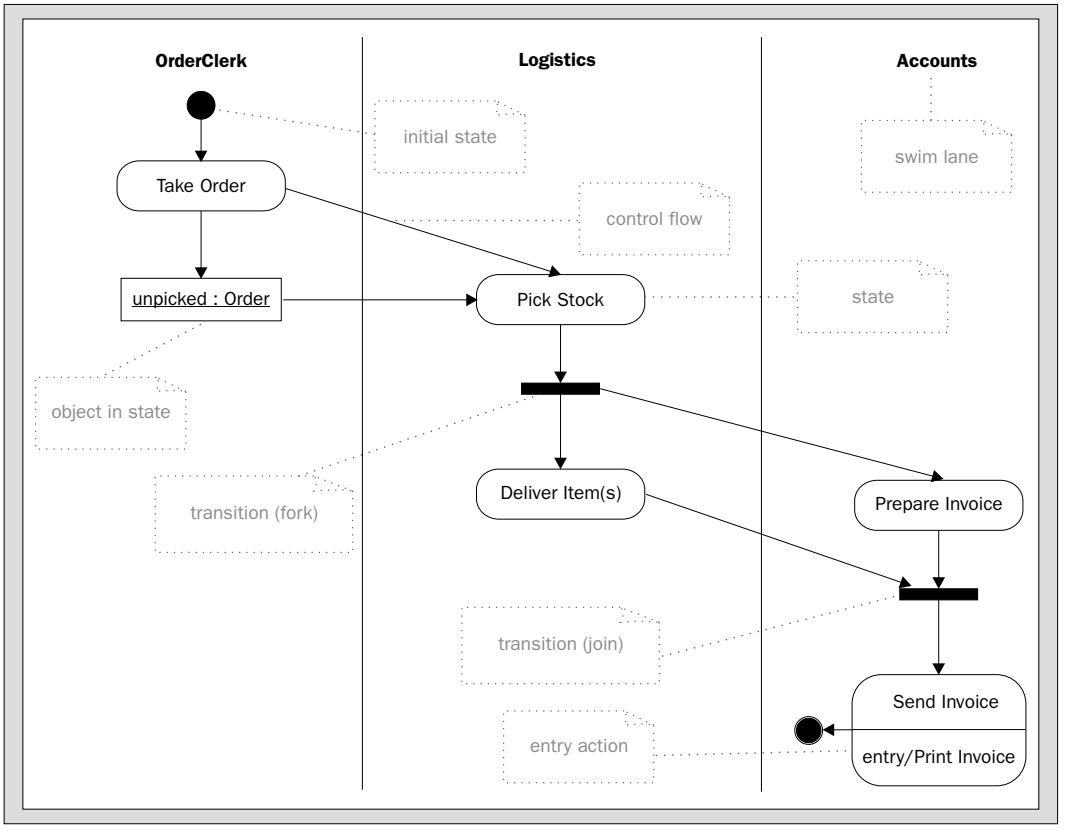
> **The UML Metamodel is itself a UML model, which defines the rules for constructing other UML models. Whereas in one of your own models you might state "Bank is associated with one or more Accounts", the metamodel would state a more generic relationship of "a Class may be associated with any Other Class".**

On the whole, the model elements have been labeled using Visio EA terminology so as to reduce the potential for confusion when you come to use the tool. Historically – and in other modeling tools – you may have encountered alternative UML terminology. The alternative terms have been tabulated towards the end of this chapter.

As you'll see later in this chapter, the software development process that you follow might well be described as **use-case driven**, which implies the **use case diagram** as an obvious starting point. But those use cases will doubtless fit into some kind of overall **business process**, perhaps modeled up-front by a business analyst. So we'll take a business process as our starting point and use this as a vehicle for demonstrating the most suitable diagram for that purpose; the **activity diagram**.

Activity Diagram

The activity diagram is the closest you'll get in UML to a **flow chart**, and the closest you'll get to a **business process diagram**. Here is a sample activity diagram with the important UML elements labeled, followed by a description of those elements.



- **Initial state** is where the diagram begins.
- **Control flow** shows a transfer of control from one activity to another.
- **State** represents a period of time during which a piece of work is carried out by person or team.
- **Transition (fork)** shows the point as which two or more parallel activities will commence.
- **Transition (join)** shows the point as which two or more parallel activities must synchronize and converge.
- **Swim lane** allows all of the activities carried out by a particular person or team arranged into a column.
- **Entry action** shows what must happen when the activity begins.
- **Object in state** shows an object that is produced or consumed in the course of an activity, with the production or consumption (object flow) being represented by the dashed line.

What the diagram shows

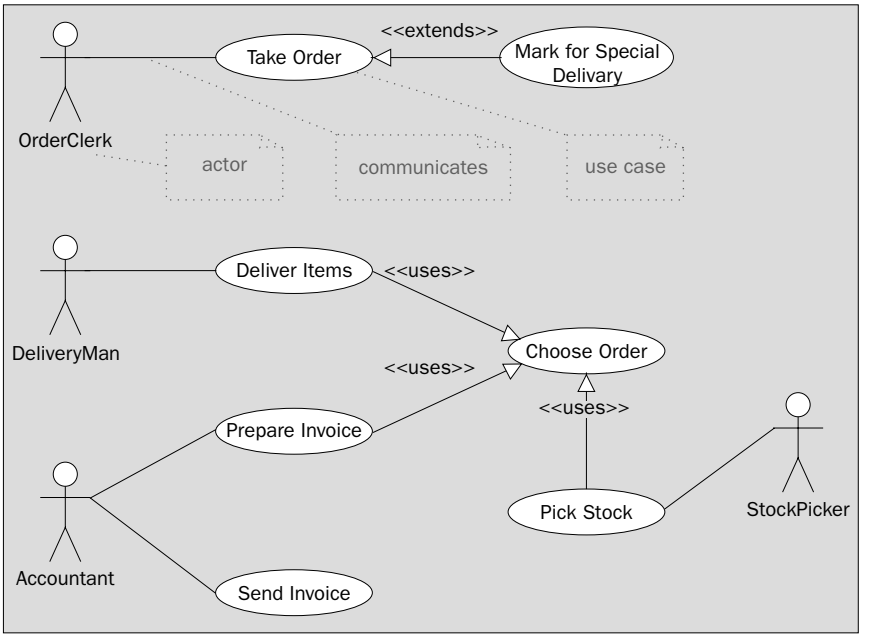
The Order Processing business process begins when an Order Clerk performs the Take Order activity. This activity results in an Order object being created in unpicked state. Next, the Pick Stock activity is performed (for the Order) by the Logistics team.

At this point some parallel behavior occurs – the Logistics team Deliver Item(s) around the same time that the Accounts department Prepare Invoice. Only when the items have been delivered and the invoice has been prepared can the Accounts department then Send Invoice. Immediately prior to sending the invoice they must Print Invoice.

Those are the essential points of an activity diagram, but not a complete coverage. In particular you will see some additional syntax in the description of a Statechart diagram.

Use Case Diagram

Here is a sample use case diagram with the important UML elements labeled, followed by a description of those elements.



- **Actor** represents a person or external system that initiates a use case in order to get some value from the application.
- **Use case** is a well-defined unit of functionality that the system provides to one or more actors.
- **Communicates** shows that a particular actor makes use of a particular use case.
- A <<uses>> relationship shows where a piece of potentially-reusable functionality has been factored out into a separate use case.

- An <<extends>> relationship shows where some additional functionality may be provided in support of a use case, that extended functionality having been factored out into a separate use case.

A Note about Stereotypes

You might wonder why the words <<uses>> and <<extends>> are enclosed within angled brackets << like this >>. It's because they are **stereotypes**; these allow a single UML element (in this case a generalization line with a triangular head) to represent slightly different concepts.

> **Any UML element may be stereotyped and later you will see components stereotyped as <<executable>> or <<library>> in a component diagram.**

What the Diagram Shows

Taking the original activity diagram as a starting point, each of the activities – Take Order, Pick Stock, Deliver Items, Prepare Invoice, and Send Invoice – has been represented as a use case. A one-to-one correspondence between activities and use cases is not mandatory, but here it shows the potential for traceability between the diagrams.

You will also see a correspondence between the actors in this diagram and the swim lanes from the original activity diagram. The Order Clerk swim lane is represented as an Order Clerk actor, the Logistics swim lane is represented by DeliveryMan and StockPicker actors, and the Accounts swim lane is represented as an Accountant actor.

Choose Order represents functionality that is common to (used by) the Deliver Items, Prepare Invoice, and Pick Stock use cases. To deliver items, prepare an invoice, or pick stock the actor must first choose an order, but to take a new order the actor does not need to first choose an order (obviously) and to send an invoice the actor need not chose an order (because they will choose an invoice).

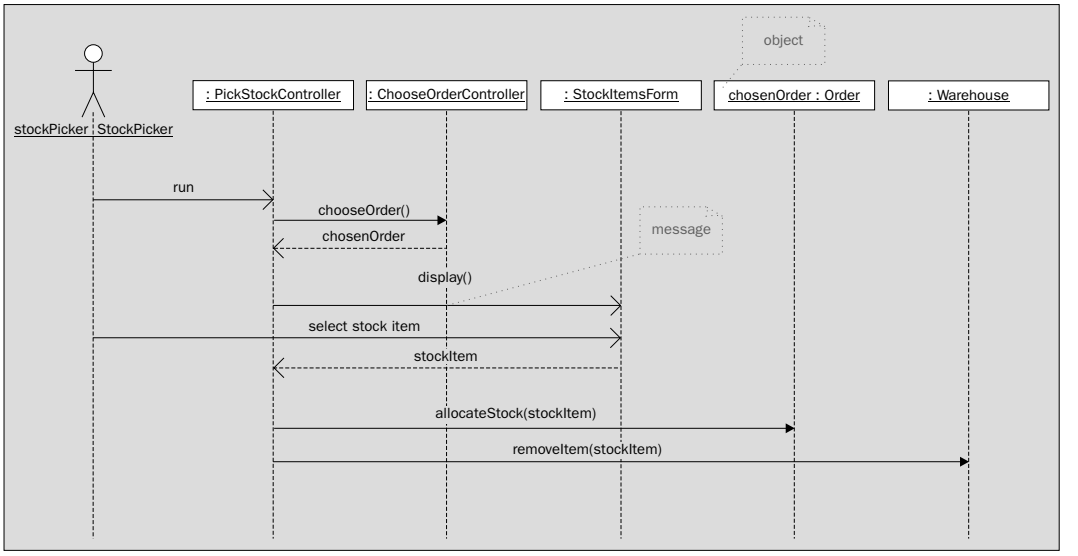
In the course of taking an order, the Take Order use case may be extended by functionality to Mark for Special Delivery. This has been modeled separately as an extension so that the extended behavior may be changed with no impact on the main use case; for example, this extension may bypass the standard procedure and instead send an instant message to the Stock Picker and Delivery Man.

Sequence and Collaboration Diagram

Use cases are realized (that is, described in terms of interactions between collaborating objects) using interaction diagrams, of which there are two types:

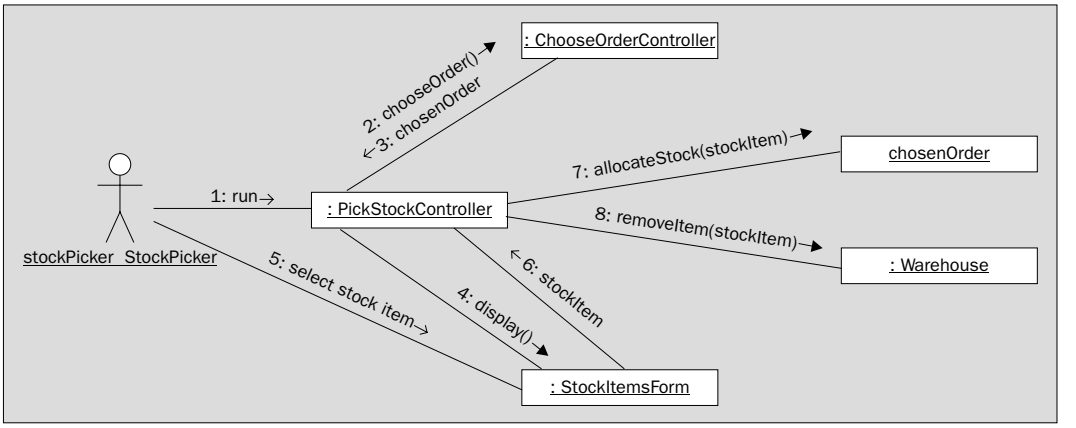
- Sequence diagrams
- Collaboration diagrams

Here is a sample sequence diagram with the important UML elements labeled, followed by a description of those elements.



- **Object** refers to an object instance that sends messages to, or receives messages from, other object instances. The objects are labeled as instanceName : ClassName where the instance name is commonly omitted to show no particular instance of the class in question.
- **Message** shows an interaction between two objects, which may be labeled using descriptive text (such as select stock item above) or may be labeled with the name of an operation on the receiving class, such as allocateStock above. A return message may be shown as a dotted line.

Here is an equivalent **collaboration diagram** showing the same set of interactions. Whereas a sequence diagram has a top-to-bottom time line to show the order of events, a collaboration diagram uses a numbering scheme. Apart from the visualization style, sequence diagrams and collaboration diagrams may be thought of as equivalent to the extent that some modeling tools, such as Rational Rose, allow automatic switching between the visualization styles.



What these Diagrams Show

Both diagrams show the sequence of object interactions that support the PickStock use case. The sequence is:

1. The StockPicker actor runs the PickStockController (which in this design is a control class responsible for the use case).

2. The PickStockController calls the chooseOrder() operation on the ChooseOrderController which results in...

3. a chosenOrder being returned. This interaction represents the fact that the Pick Stock use case <<uses>> the Choose Order use case.

4. The PickStockController calls the display() operation of a StockItemsForm.

5. The StockPicker actor selects a stock item on the StockItemsForm.

6. The selected stockItem is returned to the controller.

7. To allocate the stock to the order, the PickStockController calls the allocateStock() operation of the Order – specifically the chosenOrder that was retrieved in Step 3. The stockItem from Step 6 is passed as a parameter.

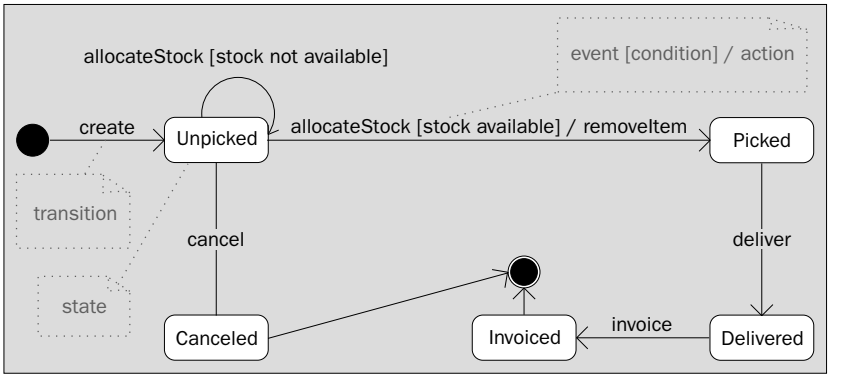
8. To remove the item from stock, the PickStockController calls the removeItem() operation of the Warehouse passing stockItem as a parameter.

Statechart Diagram

The inclusion of Order as an object in state in the original activity diagram hints at the fact that this will be a state-full class. We could have included the object multiple times on that diagram to show the state changes of an Order that result from the various activities, but for clarity we didn't.

To show the complete set of states for an Order, and – most importantly – to show the circumstances in which an Order will transition from one state to another, we draw a Statechart diagram.

Here is a sample Statechart diagram with the important UML elements labeled, followed by a description of those elements.



- **State** represents the status of an object (a function of its attributes and links) over a period of time between transitions.
- **Transition** is a movement of the object from one state to another, triggered by the object receiving an event.
- Each transition is triggered by an event, and the transition occurs only if the [*condition*] is met and the action is successful.

What the Diagram Shows

Upon receiving the create event, the Order transitions from the initial state into state Unpicked.

When an allocateStock event occurs the Order will return to the Unpicked state if the [stock not available] condition is true, otherwise – if the [stock available] condition is true and the removeItem action completes – then it will transition to state Picked. Alternatively the transition from state Unpicked may be to state Canceled if a cancel event is received.

The order may transition to state Delivered, from state Picked, upon receiving a deliver event. It may transition to state Invoiced, from state Delivered, upon receiving an invoice event.

When Canceled or Invoiced, that's the end of the line for this Order, so there is a notional transition to the end state.

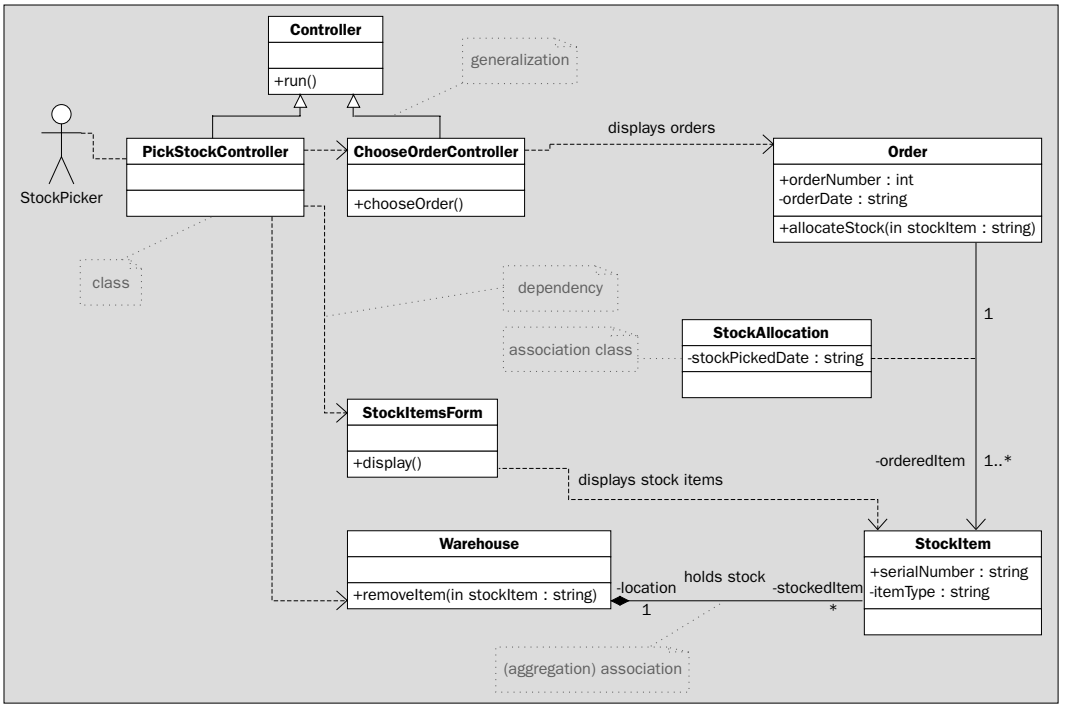
> *Only those transitions shown on the diagram are allowed. Thus for example it is impossible to move an order to state Canceled once it has been picked, because none of the states Picked, Delivered, or Invoiced has an outward transition to state Canceled.*

Static Structure Diagram

The sequence diagrams, collaboration diagrams, and statechart diagrams that we've encountered in the previous two sections are termed **dynamic diagrams**. They represent the dynamic model, which is the model showing how our system will behave over time.

We also need a static model, showing the persistent relationships and dependencies between classes and components. Out first static model will be the **Static Structure diagram** (Visio terminology), which is otherwise more commonly known as the **class diagram**.

Here is a sample class diagram with the important UML elements labeled, followed by a description of those elements.



- **Class** represents an object class as a rectangle comprising three segments that show the **class name**, the **member attributes**, and the **member operations**.
- **Generalization** is an inheritance relationship between a super-class and one or more subclasses.
- **Dependency** shows that one class depends (maybe temporarily) on the functionality of another class, but is otherwise not linked up to instances of the other class in any persistent sense.
- **Composition** shows that instances of one class may be linked to instances of another class persistently in an owner-owned kind of relationship. The composition, which is a special form of association, may be adorned with **roles** and **multiplicities**.

- **Association** is a more general form of linkage between class instances, which does not imply an owner-owned relationship. The association may be adorned with an **association class**, instances of which occur only for each link between the two associated classes.

What the Diagram Shows

PickStockController and ChooseOrderController are specializations of a generalized class called Controller. In our design, all controller classes will be subclasses of the Controller super-class.

The PickStockController class depends on the ChooseOrderClass class, the StockItemsForm class, and the Warehouse class. Look back at the sample sequence diagram to see how the PickStockController calls operations of those other classes.

The ChooseOrderController depends on the Order class by virtue of the fact that it displays orders.

A Warehouse instance owns aggregate StockItem instances according to the following aggregation:

1 (multiplicity) Warehouse (class) location (role) holds stock (association name) of * (many multiplicity) stockedItem (role) StockItems (class).

Each Order instance is associated with one or more (1..*) StockItems, with linked stock items taking the role of orderedItem. For each such link there arises a StockAllocation instance that holds the date that the stock was picked and allocated.

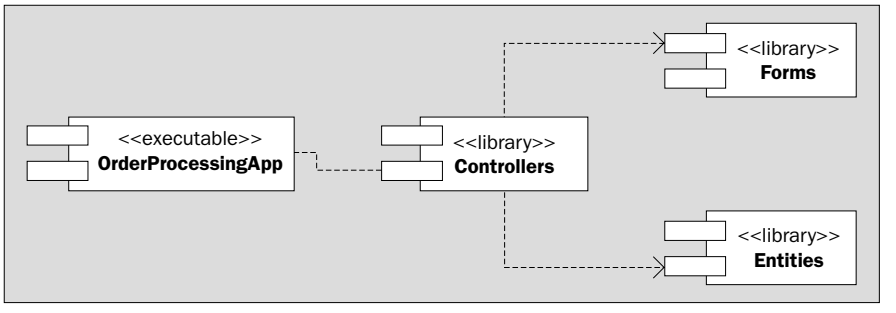
The static structure diagram (class diagram) shown here as an example does not contain the complete set of classes for the entire application, but rather those classes that are relevant to the Pick Stock use case. As such, it represents View of Participating Classes (VOPC) for that use case.

Component Diagram

An application will be delivered or deployed typically not as individual classes or even a package of classes, but rather as one or more deployable components – executables or libraries – into which the classes or packages have been collected.

> *For a Java application, these components would likely be JAR files. For a .NET application the components will be executables (*`.EXE`*) and libraries (*`.DLL`*) corresponding to the solution structure within Visual Studio .NET.*

Here is a sample component diagram with the two component types described overleaf:



- <<executable>> represents a `.EXE` file, a program that you would actually run.
- <<library>> represents a `.DLL` file, a collection of classes that you might reference in a project.

The component diagram is a static diagram, just like the class diagram, because it shows how the application is organized rather than how it will behave over time.

What the Diagram Shows

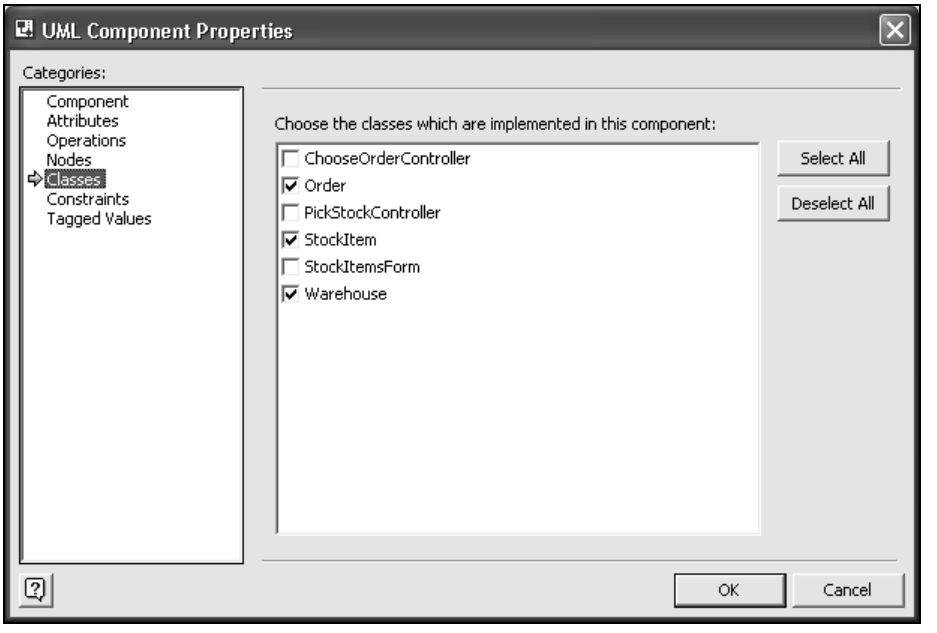
Our application will comprise a main executable program, the OrderProcessingApp deployed as file `OrderProcessingApp.exe`. This program will depend on a library of control classes deployed as file `Controllers.dll`, which in turn will depend on two more component libraries: `Forms.dll` and `Entities.dll`.

We've hinted at the fact that components represent deployable collections of classes. For this example, the mapping of classes onto the <<library>> components will be:

- The Controllers <<library>> component realizes classes PickStockController and ChooseOrderController.
- The Forms <<library>> component realizes the StockItemsForm class.
- The Entities <<library>> component realizes classes Order, Warehouse, StockAllocation and StockItem.

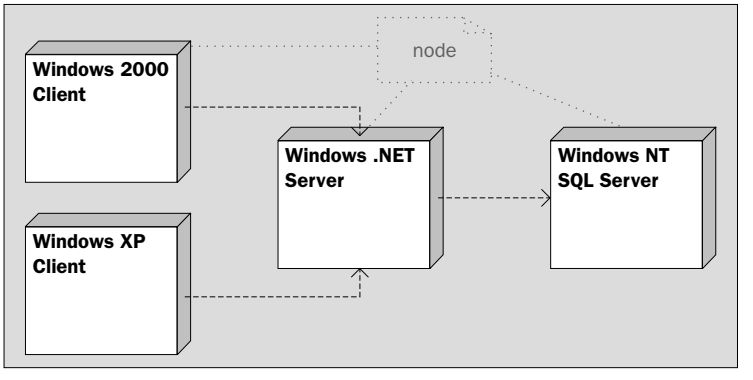
Note that this is just one way in which classes may be mapped on to components, in this case according to the types of each class: form, control, or entity. You might instead decide to package classes onto components according to application subsystems, for example StockControl.dll (containing forms, controls, and entities relating to stock control functionality) and OrderHandling.dll (containing forms, controls, and entities relating to order handling functionality).

Visio for Enterprise Architects note – although you can't see in this diagram the classes that are mapped to each component, you can double-click any component in Visio EA to view and set the list of mapped classes as shown in the following figure.



Deployment Diagram

The final UML diagram we'll look at is the deployment diagram, the purpose of which is to show the physical nodes on which the software components will actually be installed. Here is a sample deployment diagram for our hypothetical deployment platform:



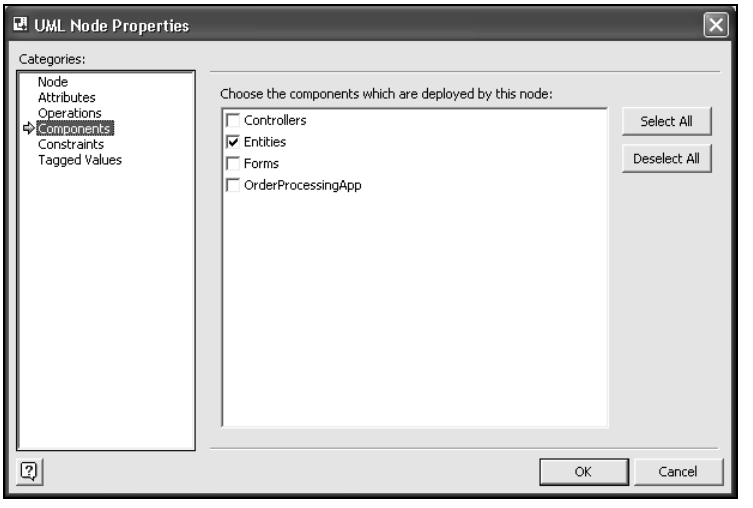
- **Node** is a run-time computational or physical resource – a software or hardware device on which software components may be deployed.

What the diagram shows

Our deployment platform will comprise a backend Windows NT server running a SQL Server database, with the `Entities.dll` component deployed to this node. There will also be a Windows .NET application server that services Windows 2000 and Windows XP clients.

Whether the `Forms.dll` component and the `Controllers.dll` component are deployed on the Windows .NET Server or on the clients themselves will depend on our choice of a thin- or fat-client architecture. For the sake of argument, we'll assume these components to be deployed to the Windows .NET Server. In either case, we'll deploy the main `OrderProcessingApp.exe` executable program directly on the client nodes.

Visio for Enterprise Architects note – although you can't see in this diagram the components that are deployed to each node, you can double-click any node in Visio EA to view and set the list of deployed components as shown in the following figure.



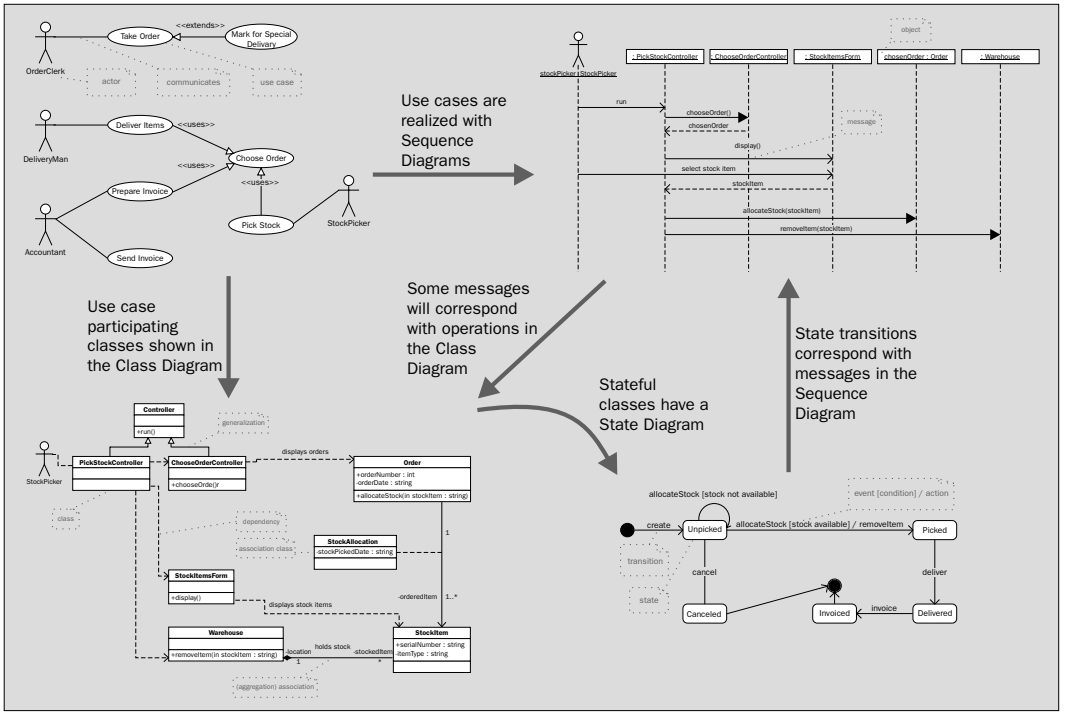
Fitting the Pieces into the UML Jigsaw

As stated earlier, the problem with many UML books and training courses is that they often present the various diagram types in isolation. To make things worse, the examples are often disjointed and not relevant to any system that you're ever likely to build: a vehicle gearbox as the state diagram example, a telephone handset as the sequence diagram example, an insect classification for the class diagram. All of which leaves you wondering about the relevance of these modeling techniques and the relationships between the various techniques.

To address the issues of relevance and consistency of examples you will notice that all of the diagrams in the previous section relate to a common application, the **order processing application**, which is one that should be familiar to you, whatever your background.

Now, what of the relationships between the diagrams that we've alluded to? Well, each diagram shows a different aspect of the same application design so they should be taken, not individually in isolation, but as a **consistent** whole. The word consistent in that sentence is important because you affect the correctness and completeness of your design significantly by ensuring consistency between the diagrams.

The following figure shows how the various diagrams relate to each other at the macroscopic level. The Use Case Diagram represents the functionality requirements of the system from a user's – or a least a system analyst's – perspective. These use cases are realized as the object interactions of a Sequence (or Collaboration) Diagram and the use case participating classes may be represented as a Class (Static Structure) Diagram. For each class that is stateful in nature there may be a State diagram.



Don't worry if you can't read the detail in these four diagrams. You've seen them all before in this chapter and you can refer back to them. What is important is that you understand the significance of the arrows that show how the diagrams fit together.

Here's the consistency bit, which we've distilled into the following set of rules:

- Does every use case have at least one Sequence or Collaboration Diagram describing its realization? If not, the design is incomplete.
- Is every one of the classes of the Static Structure Diagram present on at least one Sequence (or Collaboration) diagram? If not, you might be missing a use case realization or even a whole use case because, effectively, the class is never used.

- For all stateful classes is there a corresponding Statechart Diagram? If not, the rules for allowable state changes will be ambiguous or unknown.
- For each event in a Statechart Diagram is there a corresponding message in a Sequence Diagram that provides a context in which the event actually occurs? If not, the state transition may never occur.

Depending on your approach to analysis and design, and the kind of application you're developing, those rules may be more or less important and you're unlikely ever to achieve 100% mutual consistency. So treat them not as revealed truth but as rules of thumb – I've found that they've certainly served me well in my development work.

The fact that the previous figure has arrows emitting from the Use Case Diagram – with none going in – suggests that as the starting point. That's true unless you draw some activity diagrams up-front, and it makes perfect sense to start with the diagram that represents the functional requirements doesn't it? However, the diagram to start with is not really a question of UML but a question of **process**.

We'll conclude this chapter by looking at the process side of things, just after a brief mention of the tools that support the modeling effort.

UML Modeling Tools

It's doubtful that anyone would be working with UML these days without the aid of a modeling tool, because these tools are to software design what a word processor is to writing.

In creating a chapter like this one, few authors would ever dream of writing out the words long-hand with pencil and paper. How would they delete unwanted paragraphs, rephrase sentences and insert the pictures without making the first draft a complete mess? – and how time-consuming would it be to write it all out again for the final draft?

Now make the analogy with UML diagrams and a modeling tool. How would you add an operation to a class on a static structure diagram, or change the order of events in a collaboration diagram, or change a relationship between an actor and a use case in a use case diagram without a significant amount of redrawing?

At this point, the following question might occur to you:

> *"OK, but we could just use a good drawing package to solve those problems. They're only diagrams, right?"*

Wrong! The whole point about a modeling tool is that besides allowing you to draw the diagrams, it actually understands the model you're creating. It knows that a line between two classes is an association or aggregation so an instance of one class must be linked to an instance of another class, perhaps via a member variable. It's this understanding of the model that allows the modeling tool to provide added value to your software development effort through code generation, documentation production, and model semantic checking.

Before the arrival of Visual Studio .NET on the software development scene, your choice of modeling tool would most likely have been one of **Rational Rose**, **Select Enterprise**, or **Together Control Center** – none of which cater specifically for UML in the context of .NET. The main contenders in the .NET modeling space are **Rational XDE** and **Visio for Enterprise Architects**.

Rational XDE has the Rational pedigree, some impressive .NET-related features, and tight integration with Visual Studio .NET; so it's well worth a look if you're from a Rational tools background. The main problem is that you may have to shell out on an expensive license on top of what you've already paid for Visual Studio .NET, and – on that subject – it actually won't run without the IDE.

Visio for Enterprise Architects comes bundled with the Visual Studio .NET (Enterprise Architect version) and/or an MSDN Universal Subscription, so you may already have it at no extra cost. It supports UML notation as well as the back-catalogue of other Visio diagram types. Code generation, reverse engineering, model semantic checking, and document production are supported, plus integration with the Visual Studio .NET IDE. All of which make this *not just a drawing tool*, or – more accurately – *no longer just a drawing tool.*

Process Essentials

UML is a notation not a process, but invariably you will use UML in the context of a software development process; so which one to choose?

As indicated already, you are not compelled to use any particular process. You could adopt a pure **Rapid Application Development** (RAD) approach, or join the growing band of practitioners adopting the **eXtreme Programming** approach. Just a few years ago you might have been tempted by the **Select Perspective** method, which was – and still is – biased towards component-based development and based on an **iterative-incremental** approach. In all cases there would be nothing to stop you using UML as the analysis and design notation.

To round off this chapter we'll focus in on two processes in particular, the **(Rational) Unified Process** and the **Microsoft Solutions Framework**; the former because has been devised by the authors of UML as the preferred partner process, and the latter because in this book we're interested in designing software for the Microsoft environment.

(Rational) Unified Process

The Unified Process has its roots in the Objectory method devised originally by Ivar Jacobson. It represents the unification of the process ideas of the three amigos, thus it is complementary to the Unified Modeling Language and is marketed by Rational Software as the Rational Unified Process (RUP).

In practical terms what you get when you purchase this product is a set of HTML pages describing the process, its roles, activities, and artifacts, along with a set of Microsoft Word templates that provide a starting point for those artifacts – not to mention a great deal of encouragement to buy the Rational Suite.

Three of the essential points of this process are that it is:

- **Use-case driven** – this ensures that the system we build will actually meet the requirements of the business.
- **Architecture-centric** – so we won't complete the analysis under the blind assumption that we can build the application on our chosen technical platform. Early on we'll do some technical prototyping.

- **Risk managed** – with an emphasis on tackling the tricky parts of the system – the architecturally significant use cases – at the beginning rather than at the end to reduce the likelihood of nasty surprises later on.

The fact that the process is use-case driven suggests that we start with the use case diagram(s). That's true to an extent, but not the whole story. We've already suggested that activity diagrams may add value early on by describing the workflow of the business, in essence the ordering of the use cases. There's a lot to be gained by producing a static structure diagram (class diagram) of fundamental business entities up front, to be called the **domain model**.

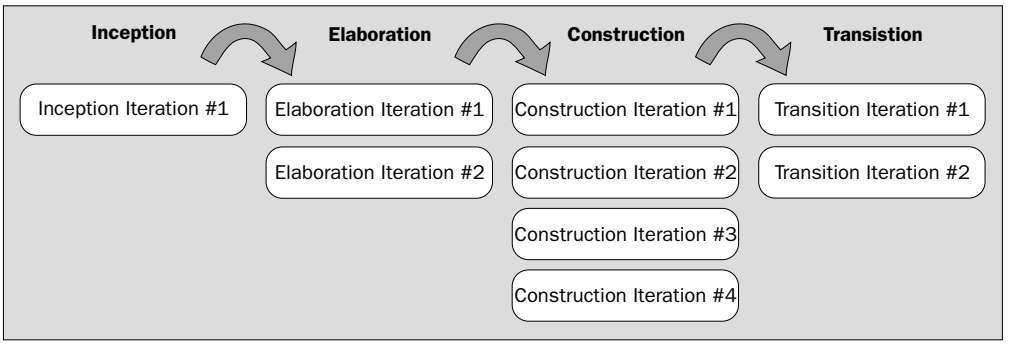
The point is that although the various diagrams will each have a lesser, or greater, impact as we move through various stages of the process, we won't simply be stepping through the diagrams in waterfall fashion. Rather, we'll analyze, design, and build the software as a series of **increments** via a set of **iterations**.

Within each iteration, any combination of diagrams may be valuable, those diagrams becoming more detailed as we go along. However, it goes without saying that as we approach implementation we'll have much greater need for component diagrams than for use case diagrams!

Though the Unified Process is iterative, not waterfall in nature, these iterations do fit into a series of distinct phases called **Inception**, **Elaboration**, **Construction,** and **Transition**.

- Inception is the initial phase in which you establish the business case for the project and determine the project scope.
- Elaboration is the phase in which you gather detailed requirements, undertake analysis and high-level design, define the architecture, and plan for construction.
- Construction is the phase in which you undertake the detailed design and build the software components themselves.
- Transition is the final phase in which you test the software, tune for performance, and train the users in preparation for going live.

The relationship between phases and iterations is shown in the following figure.



RUP .NET Developer's Configuration

Since we're dealing specifically with .NET application design in this book it's worth mentioning that the Rational Software web site (http://www.rational.com/products/rup/sample.jsp) describes a variant of the Rational Unified Process called RNDC, which is defined as the following:

> *"The RUP .NET Developers' Configuration (RNDC) is a straightforward, lightweight process configuration of the Rational Unified Process® that has been specifically customized to address the needs of the .NET software developer."*

There are two important aspects here.

Firstly, it's a customization of the Rational Unified Process specifically for the .NET development environment. Historically RUP has been biased towards Java software development and tools, with .NET now presenting some new technical challenges – and marketing opportunities – for customized version of the process.

Secondly, it's aimed specifically at software developers rather than all the team members defined by RUP. Presumably, no customization was required for technology-independent business analysts, but this also seems to reflect Rational's positioning of UML in the context of .NET. Experimentation with the new **Rational XDE** UML modeling tool shows this to be much more of a **developer tool** than Rational Rose ever was.

At the URL listed above is the RNDC roadmap, which provides a somewhat disappointing overview of the customized process. Under the headings Requirements Activities and Analysis Activities it simply states the following, which is at least consistent with the presumption that certain aspects of the process require no customization:

> *"Requirements activities are technology independent."*
>
> *"Analysis activities are technology independent."*

Under the heading Define an Initial Architecture, the reader is encouraged to use the .NET Framework – in particular Enterprise Templates – "*to create reusable reference architecture templates for .NET applications that can be tailored to support a certain application structure or a specific application domain*"

Finally, the RNDC roadmap references several concepts and guidelines such as *"Concepts: Microsoft .NET Architectural Mechanisms"* and *"Guidelines: Partitioning Strategies in Microsoft .NET"*. Unfortunately these additional references are not hyperlinked in the RNDC, which renders it not too useful in itself. For a complete picture – with hyperlinks to all the required content – we need to look into the RUP .NET Plug-in.

RUP .NET Plug-in

The vanilla Rational Unified Process may be enhanced by applying various plug-ins for:

- Compatibility with alternative approaches, such as eXtreme Programming
- Technologies such as IBM Websphere and, of course, .NET

You can find general information about the .NET Plug-in at URL http://www.rational.com/tryit/rup/seeit.jsp and, more usefully, you can step through a slide-show presentation at URL http://www.rational.com/demos/viewlets/rup/msnet/MSNET_Tour_viewlet.html.

In that presentation you will see that this plug-in contains detailed information in the form of workflows, roadmaps, guidelines, and links to relevant information on the **Microsoft Developer Network** (MSDN) and **Rational Developer Connection** web sites.

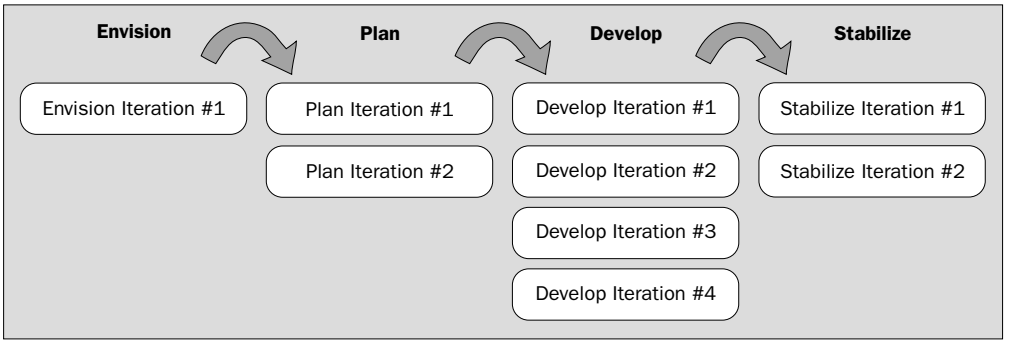
Microsoft Solutions Framework

The **Microsoft Solutions Framework (MSF)** is a process-methodology for development in a Microsoft environment. In effect, we can view MSF to be a potential substitute for the Rational Unified Process, perhaps one that is more relevant to the Microsoft tools we'll be working with.

A Framework not a Process

We've referred to the MSF as a process, to justify a comparison with RUP. In fact, it's a framework incorporating a **Process Model** (the process), a **Team Model**, and a **Risk Management Model**. Let's start with the process model.

The process model is described as **phase-based, milestone-driven, and iterative**. We've taken the liberty of incorporating iterations within the four phases of the core process – **Envision**, **Plan**, **Develop**, and **Stabilize** – to come up with the following figure.



You should be experiencing some déjà vu now, and if you're not you should look back at the RUP process described previously. For Envision (MSF) read Inception (RUP), for Plan (MSF) read Elaboration (RUP), for Develop (MSF) read Construction (RUP), and for Stabilize (MSF) read Transition (RUP).

The process model is not the only area of similarity between the MSF and RUP. As mentioned above, the MSF includes a Risk Management Model and earlier we described RUP as being a risk-managed process, and where the MSF incorporates a Team Model this corresponds with RUP roles and activities.

A sensible conclusion then is that whichever process you start off with – RUP or MSF – the underlying concepts and approach are similar enough to allow a degree of compatibility or a change of mind later on. Indeed the MSF datasheet proclaims the following:

> *"[MSF] ... can easily coexist with virtually any other process framework or provide sufficient structure where no methodologies are in place."*

Summary

In this chapter we've introduced the Unified Modeling Language in terms of what it is (an analysis / design notation) and what it is not (a software development process). We said that the notation represents a synthesis of three predecessor methods – Object Modeling Technique (OMT), the Booch Method, and Object-Oriented Software Engineering (OOSE) – with contributions from some others.

In terms of why you might use UML at all, we offer four main reasons:

- Establishing a blueprint from the application
- Estimating and planning the time and materials
- Communicating between teams, and within the team
- Documenting the project

The remainder of the chapter was divided into two main sections, *End-to-End UML Modeling* dealing with the UML notation and *Process Essentials* dealing with the companion process(es). Let's now review the modeling and process sections.

Modeling Summary

In this section, we looked at seven UML diagram types:

- Activity diagrams
- Use Case diagrams
- Sequence diagrams
- Collaboration diagrams
- Statechart diagrams
- Component diagrams
- Deployment diagrams

Each kind of diagram was annotated with the UML metatypes such as **actor**, **use case**, **class**, **dependency**, **association,** and so on.

Each diagram represented a different view of exactly the same application, so that you could relate the diagrams to each other with the help of the *What this diagram shows* sections. We consider the relationships between the diagrams to be so important – and all too often ignored – that we placed further emphasis on this point in the *Fitting the Pieces into the UML Jigsaw* section.

Finally, we suggested that you will almost certainly be doing UML modeling with a dedicated modeling tool, and that doesn't just mean a good drawing tool. Visio for Enterprise Architects represents such a modeling tool, no longer just a drawing tool, that we set out as the preferred tool on which the rest of this book has been based.

In the main, Visio terminology has been used *in this chapter* so as to avoid confusion when you come to use the tool. Other modeling tools may use slightly different terminology and, in fact, the UML terms themselves have changed slightly over the years. To help with the transition to – or from – other books and tools, here is a summary of this chapter's Visio UML terms and the alternative terminology that you may encounter:

Visio Terminology	Other Terminologies
Static Structure Diagram	Class Diagram
Package	Category
<<uses>>	<<import>>
State	Activity (on Activity Diagram), State (on Statechart)
Statechart Diagram	State Transition Diagram
Transition (fork)	Synchronization (start)
Transition (join)	Synchronization (end)

Process Summary

As to which software development process you should adopt, two were picked out two for discussion. The Unified Process, because it's the natural companion for the Unified Modeling Language, and the Microsoft Solutions Framework, because it's the Microsoft process offering. What these have in common with each other – and with other good object oriented software processes, such as the Select Perspective – is that they are:

- Iterative and incremental
- Use-case driven
- Focused on Risk Management

You also have a choice of **eXtreme Programming**, traditional **waterfall**, or **RAD**, and as the UML notation is independent of the process, ultimately the choice is yours.

In the next chapter, to complete our foundations for working with Visio for Enterprise Architects, we'll take a tour of the Visio environment and look at some of the available diagram features relevant to the software developer.