UML – A Programmers Guide

The **Unified Modelling Language**, UML, is a software analysis and design modelling language that supports the concepts used in object-oriented software systems. As a modelling language, it comprises a set of (mainly) graphical forms of notation that allow us to depict the outcomes of the analysis, architectural design, detailed design, and deployment phases of the development of a software system. It also includes notation for depicting interactions and state changes within a software system at a conceptual level.

In this guide, we will concentrate on the notations used by software designers and programmers – mainly Class diagrams and Interaction diagrams. However, since many programmers working alone, in small companies or as consultants are also required to perform software analysis for small to medium sized systems, it will also include a short guide to Use-Case diagrams and notation.

UML Overview

UML is used to document and record all of the activities that surround programming in the development of software systems. Some UML tools (e.g. VISIO, Rational Rose) also provide some support to programming by allowing for a direct automatic transition from class diagrams to class, although UML notation is perfectly amenable to being done by hand. Using UML, a system development is conducted in several phases that mesh in with the Analysis, Specification, System Design and Detailed Design phases that are required of any formalized software project.

System Development Phase	Process Phase	UML Step/Notation
Feasibility Study	Inception	None
Requirements Analysis	Elaboration	* Use-Cases, * Use-Case Diagrams
Specification	Elaboration	* Class Diagrams
Architectural Design	Elaboration	* Class Diagrams, Package Diagrams
Detailed Design	Construction	* Class Diagrams, * Activity Diagrams, * Interaction Diagrams
Implementation	Construction	Programming
Deployment and Maintenance	Transition	Deployment Diagram

Table 1: Software Development and its correspondence with UML

Although we will look briefly at all of the UML steps involved in bringing a software system from concept to installation, only the steps marked * in the table above will be described in any detail.

Iteration

One of the key features recognised in UML is that software development is rarely if ever a linear process – i.e. we don't start at step 1 and progress through to step N without ever having to repeat a step, retrace our steps occasionally and, now and again, throw work away and begin it again having (hopefully) learned something from our initial attempts. UML fully embodies the idea of Iterative processes in software development.

In some stages of development, we will design or build a prototype of part or all of a system in order to gather more information necessary for us to design or build the final version. This could be considered as an evolutionary prototyping step where we use the process of development to help us to uncover the details of what functions the system should perform.

Iteration is also how we can describe the process of designing and building a *part* of a system from analysis through to production-quality software, and then repeating for other parts of the system. In this case, each iteration might contain all of the usual life-cycle phases of analysis, design, implementation and testing.

In either case, we can use UML as a modelling tool to govern the iterative process embodied in a Evolutionary Prototyping life-cycle. Recall that Evolutionary Prototyping is a process model that has already been considered as fitting well with object-oriented software development. Figure 1 shows a typical diagram of Evolutionary Prototyping and (loosely) how it corresponds with UML methods:

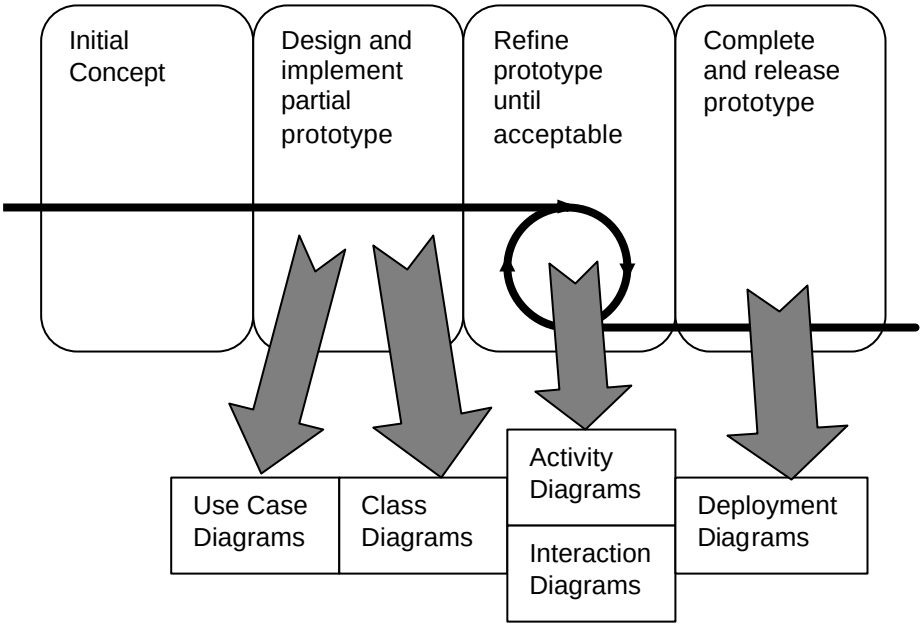

Figure 1: Evolutionary Prototyping

Use-Cases

A Use-Case is a scenario describing a typical interaction between a user (or another software system) and the software system being developed. The idea is simplicity itself – describe in human terms all of the typical interactions the system will be required to perform to produce clear, unambiguous statements of requirements. Typically, we talk of *capturing* a use-case, meaning that by having fully discussed a use-case/scenario for an interaction with the system with those who will use it to their satisfaction, we will be able to document it fully enough to make it a useful design component.

A use case describes a single interaction between use and computer system. For example, in a word processor, typical use-cases might be to save a file, make some text bold or paste a passage of text into a document. They are described in non-technical terms a user will understand. Key features are:

- A use-case captures some function that is apparent to the user (for example, if a program performed an automatic backup file invisibly every 10 minutes, this would not be a use-case since it is not visible to the user)
- A use-case can be small or large (e.g. in a word processor, edit a single character or print an entire document)
- A use-case achieves a discrete goal for the user (e.g. no use case would complete part of an operation or more than one operation – so Copy a passage of text is a use-case, Paste a passage of text is a use-case, but Copy and Paste a passage of text is not a use-case since it is better described as two discrete ones)

Generally speaking, the goals of a software system are implemented by implementing use-cases. A use-case is not a goal, but one way of implementing it. For example, the overall goal for one part of a system might be to "ensure the system data is persistent". Use cases that might implement this include "Save the system data to a file", "Add new data records to a database", "Stream the updates to an archive server" or any of a dozen other ways of keeping data from disappearing when the system is switched off. It is a good idea to distinguish between goals and use-cases: a first cut at the design of a system should concentrate of the general goals – the first refinement should be to write use-cases.

Use-Case Diagrams

Most use cases should be defined as a short (or, if necessary, long) passage of text, describing the required interaction, any assumptions made, the goal that it implements or partially implements etc. However, a better overview of a system or part of a system is achieved by drawing a use-case diagram. This is simply a diagram showing the Actors that will interact with a system and the use-cases they interact with. The use-cases are drawn as simple bubbles with short text descriptions in them. The term Actor is used to mean anyone or anything that interacts with a system, but is always drawn as a stick-man to indicate its autonomy from the system. Typically this is a user, but systems can also interact with other systems – for example, a web-server receives interactions from the Internet, an air-traffic control system receives interactions from the radar controllers and other sensors, an ATM banking machine controller receives data from the banks main data server and a signal from the cash hopper to tell it that there are no more £10 notes etc.

The Use-Case diagram is part of UML. Here is simple a diagram for a banking machine (it would probably not satisfy the designers of such a system who would have to consider a great many other factors – mainly for security reasons):

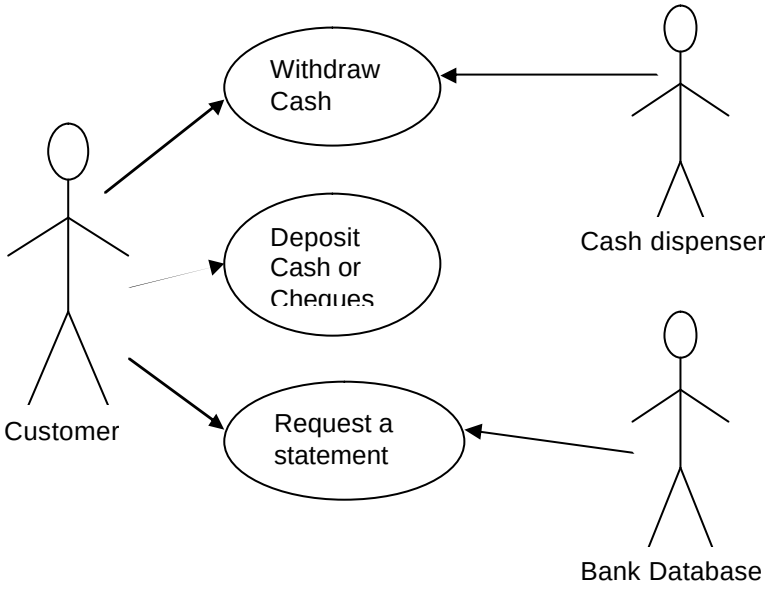

Figure 2: Use-Case diagram of an ATM (simplistic)

Some guidelines for drawing use-case diagrams:

1. Remember a use-case is a scenario – not (necessarily) a system function
2. A use-case should be a discrete thing – not part of a bigger operation and not several operations
3. Actors are entities that interact with the system. Some designers will only include an actor if it has a causal interaction with the system – i.e. the actor is the entity that needs the service of a particular use-case to do its job. The cash dispenser in figure 2 would not be considered to be an actor using this style, since it is simply performing a service for the banking machine, nor would the Bank Database, since it simply participates in the task of generating a statement. Certainly, the end-users of the data are the important drivers of use-cases, but it is often useful to include the other participants to make the overall picture clearer to the user.
4. Keep it simple and clear, using obvious names and unambiguous use-case text.

Analysing Use-Cases

A use-case is an analysis tool. A system may require one or many use-case diagrams according to its complexity. Ivor Jacobson suggests about 20 use cases will be enough to describe quite a large system (in his example, a 20 person-year project). Others suggest the use of many more. Certainly

too many use cases will simply make a confusing picture of a system, but too few will not provide enough detail to let the designers do their work.

When the use-cases have been drawn and fully described, they become an important tool for the software designers. Start by writing a short passage of text to fully describe each use-case. For the ATM example:

- Withdraw Cash: A customer (user) enters the system by providing a bank card and PIN. The card is checked for validity (is it out of date?) and, if it is valid, checked against the user's PIN. This identifies the user's account and their authorization. The user then selects Withdraw Cash from the menu, and is asked to enter the amount. The amount entered is checked – it must be no more than the daily limit and a multiple of the smallest value of note held in the dispenser. If this is valid, the card is returned. Then the cash is dispensed and details of the transaction are printed on a receipt, printed on the internal audit roll and the sent to the bank's transaction server. Finally, the customer's receipt is dispensed, and the machine returns to an idle state
- Deposit Cash or Cheques: A customer (user) enters the system by providing a bank card and PIN. The card is checked for validity (is it out of date?) and, if it is valid, checked against the user's PIN. This identifies the user's account and their authorization. The user selects Deposit Cash or Cheques from the menu, and is asked to enter the amount. This amount is displayed and the user is asked to confirm. Once confirmed, a deposit envelope is dispensed and the system waits until the envelope is returned to the machine. The date, time and transaction details are printed on the envelope and the customer's receipt, and the transaction data is sent to the bank's transaction server. The user's card is returned. Finally the customer's receipt is dispensed and the machine returns to an idle state.
- Request a Statement: A customer (user) enters the system by providing a bank card and PIN. The card is checked for validity (is it out of date?) and, if it is valid, checked against the user's PIN. This identifies the user's account and their authorization. The user selects Request a Statement, and is given the options Print Statement or Display Statement on a menu. When the selection has been made, a request for a statement is sent to the bank's database server. The user's card is returned. When the statement details are returned from the bank's database they are displayed or printed as requested.

Note that all three of these use cases provide nominal description of the operations. No errors or exceptions are considered, although these will need to be considered by the system's designers. For example, what if the machine is out of money, or paper for the printed receipts. What if the user simply walks away half way through an operation, or changes their mind half way through a transaction. These are not considered too carefully as features of use-cases – it is more important to get the nominal flow of each scenario nailed down.

We need to make the transition from Use-Cases to class diagrams, and this can be a difficult area, since we need to partition a whole system into individual components and describe each of these in terms of their responsibilities and collaborations. However, the general principle is to examine each use-case in turn, and identify the actors and components that are involved in it (the nouns used in the use-case description) and the responsibilities that each has (the verbs). Take the Withdraw Cash use-case as an example:

> Withdraw Cash: A **customer** (**user**) *enters the system* by providing a **bank card** and **PIN**. The **card** is *checked for validity* (is it out of date?) and, if it is valid, *checked against the user's PIN*. This identifies the user's **account** and their **authorization**. The **user** then *selects Withdraw Cash* from the **menu**, and *is asked to enter the amount*. The **amount** entered is *checked* – it must be no more than the **daily limit** and a multiple of the smallest value of note held in the **dispenser**. If this is valid, the **card** *is returned*. Then the **cash** *is dispensed* and **details** of the **transaction** are *printed* on a **receipt**, *printed* on the **internal audit roll** and the *sent* to the **bank's transaction server**. Finally, the customer's **receipt** is *dispensed*, and the **machine** *returns to an idle state*

Throughout the above use-case description, I've made the nouns, which will possibly be objects, **bold** and put the verbs, potential methods, *in italics*. From this we could make a first cut of the classes used in the scenario:

Customer	**Daily Limit**
User	**Dispenser**
Bank Card	**Cash**
PIN	**Details**
Card	**Transaction**
Account	**Internal Audit Roll**
Authorization	**Bank's Transaction Server**
Menu	**Machine**
Amount	

This list is of all the unique nouns or noun-like phrases in the description – no duplicates. We can also cull synonyms (e.g. **Bank Card** and **Card**, **Customer** and **User**) etc. Beyond this, we need to consider what is actually being done and its significance to the overall task. For example, we might consider Daily Limit since it is noun-ish, but a moment's thought will tell you that this is in fact a property of one of the other objects in the system (the Account probably). Going through the rest of them we can cull *Bank Card* (not a software object but a real entity, that will probably take part in the method of the account used to validate a customer) *PIN* (a property of the Account), *Authorization* (similar to checking for validity, and therefore likely to be a method of the Customer, Card or Account class), *Menu* (which is likely to be an operation of ATM), *Amount* (a property of Transaction), *Dispenser* (a non-software object that will be handled by a method of the Machine object), *Cash* (not a software object, although some verb, to Dispense Cash, will be required), *Internal Audit Roll* (again, not software, and again will require a verb to print on it), *Receipt* (same thing – a non-software object that a method will create) and *Details* (a property of Transaction). A better name for Machine is **ATM**. This leaves us with:

Customer	**Bank's Transaction Server**
Account	**ATM**
Transaction	

From this list, we can drop **Bank's Transaction Server**, since this will be an external system, leaving ourselves with a much shorter list of class-candidates. We can also identify **ATM** as the top-level object in this scenario, and therefore likely to interact with all of the other objects.

Class Diagrams

Now that we have decided on a set of class names, the next step would be to create a first-cut of class designs, using class diagrams. These are very simple, indicating for each class only the name, a list of its methods and a list of its properties. Class diagrams are also called Static Diagrams, since they depict the static relationships between classes based on associations and inheritance. These two forms of relationship are designed into classes with no consideration of the dynamic features of a system. For example, a typical association for the banking system is that an Account has a number of Transactions. We don't specify what the number is in this case because it changes through the life of the account object. These changes can not be shown in a class diagram – only the more general notion that an Account has a number of Transactions. Other associations of this type might involve a specific number of one end of the association – for example, a Car has four Wheels.

A class diagram showing the relationship between Account and Transaction is shown below:

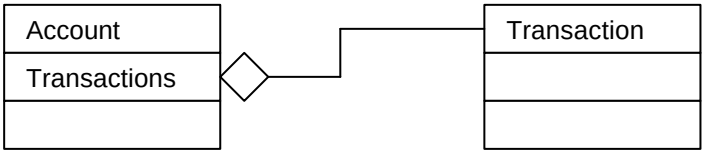

Figure 3: Class diagram showing the relationship (association) between Account and Transaction

Notable features of the class diagram are:

- Each class 'box' is divided into 3 sections, showing (from the top to the bottom) the class Name, a list of its Attributes (Properties) and a list of its Operations (Methods).

- The diamond shown next to Transactions indicates an Aggregation association between the Account class and the Transaction class. We can also show this more explicitly by placing a number at either end (in place of the diamond) indicating the multiplicity in the relationship. In this case 1 at the Account end, and * at the Transaction end, indicating one to many.
- Simple names are used throughout, with plurality as appropriate – an Account is associated with a number of Transaction**s**.

Attributes

Attributes are values that apply to a class of objects. For example, a Transaction has several values that apply to it: the Amount of cash involved, the Date and Time of the transaction, an indication of the Type of transaction it is (Deposit, Withdrawal, Interest Payment, Bank Charges) and some Description of the nature of the transaction (ATM, Bank Counter, Phone Bank, Cheque etc.). The Account class also has attributes, one of which is the Customer to whom the account belongs, also the list of Transactions and the PIN for authorizing access to the account. Re-drawing the class diagram:

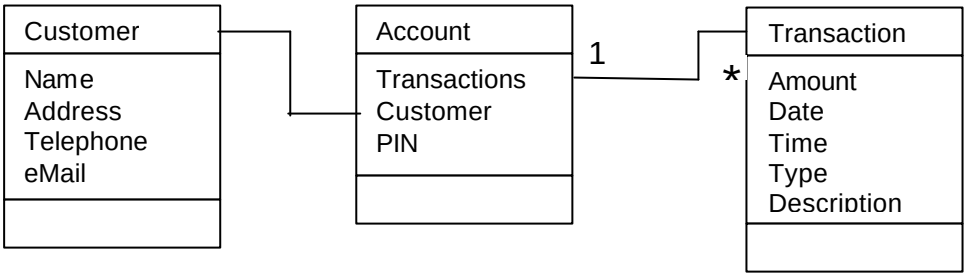

Figure 3: Bank system class diagram showing Attributes and the Customer association

Note that the relationship between Account and Transaction is that an Account has a list of Transactions (Aggregation), but that this association is most directly implemented by making the list of Transactions an Attribute of Account. At a conceptual level, Attributes and Associations are very similar, and may only differ when we reach implementation. The Customer attribute of Account is more complex than a single value here, since we already have defined a Customer class that contains more information about a customer (Name, Address, Telephone Number, email address, PIN etc.). This is another Association (although not an aggregation in this case). The real distinction is whether an Attribute is a value (in which case it is a simple attribute) or a reference to an object (in which case it is an Association of some sort).

Operations

An operation of a class is some service that it can perform at the request of another object. Some operations will alter the state of an object (change the values of one or more of its attributes), others will pass some data from within the object back to the calling object and others will do both. At the implementation level (in Visual Basic), we distinguish between operations that return values and those that do not – those that return values are Functions, those that do not are Subs. You should think of operations as the **responsibilities** of a class.

Redrawing the class diagram to add operation names:

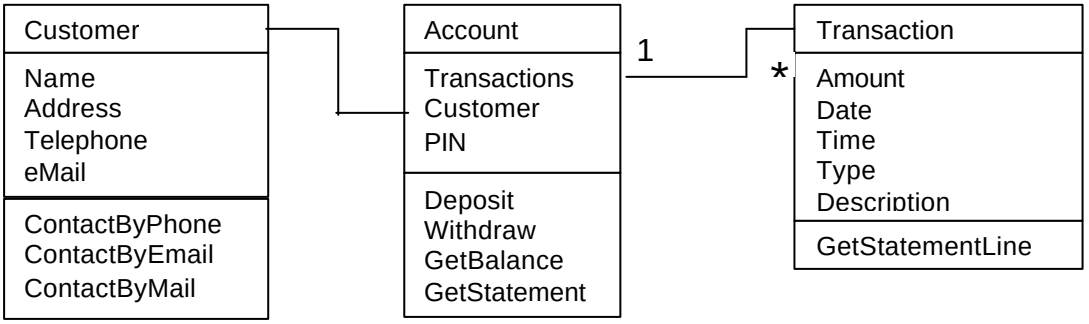

Figure 4: Classes with Operations

Operations will be implemented as Subs or Functions, and although we list the names of operations here, there is no indication of how these operations are invoked in the class diagram. Interaction diagrams and Activity diagrams show this.

Types for Attributes and Operations

Attributes and Operations can both have types associated with them. For example, the Amount attribute of a Transaction is a numeric type that will be implemented in the most appropriate type for a given implementation language (**Currency** in VB6, **Decimal** in VB7, **float** in C++ etc.). Similarly, Date and Time is implemented by the Date type if one exists, Description as a String etc. The Transactions attribute of Account is more complex, since it represents a multiple of Transaction objects at a conceptual level. In VB, we will probably implement this as a **Collection**, although a **vector** or **array of pointers** would be more likely in C++. The Customer attribute of Account would be a reference to a member of the Customer class (reinforcing the notion of an association, since this class is already part of the class diagram).

Operations that return information (Functions) also have a type, this being the type of data returned. For example, the **GetBalance** operation for Account will return a Currency value (or Decimal or float) and the **GetStatement** will return text (a String). Some operations may return objects - the Account class could, for example, be given a method, **GetCustomer**, which would return a reference to the Account's Customer attribute. We can depict the types of attributes and operations in the class diagram – this is optional, but helpful for implementation purposes.

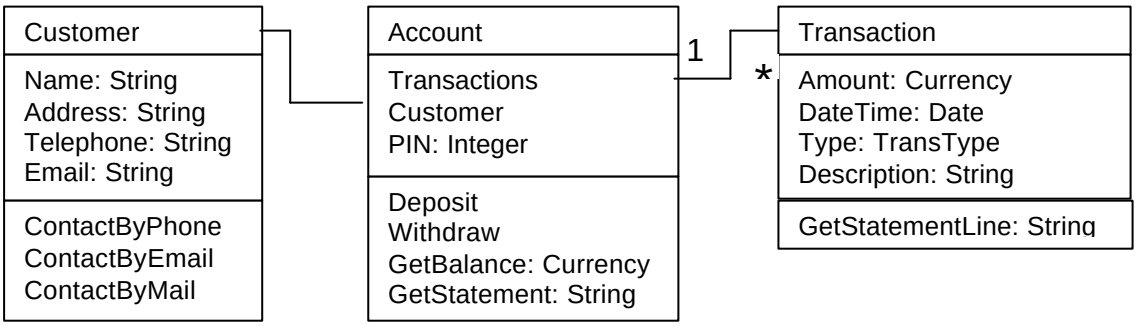

Figure 5: Class Diagram with Type information

Note that the Transaction class's Type attribute has been defined as a new data type, a **TransType**. Most object-oriented languages allow us to define new *enumeration* data types – that is data types that are defined as a fixed set of possible values. In VB, the type could be defined in a code module as:

```
Public Enum TransType
   ttDeposit
   ttWithdrawal
   ttInterest
   ttCharges
End Enum
```

Figure 6: A definition of the TransType enumeration in VB

The benefit of this enumeration is simply that it would not be possible to assign a value to a variable of type TransType that did not appear in the list. Note the naming convention in which each member of the enumeration is prefixed with tt – a mnemonic for the defined type (**T**rans**T**ype). In Visual Basic, defining an enumeration like this will also make programming easier, because the enumeration list will become a part of the pop-up members list whenever you write code to assign a value to a variable of this type.

Naming Conventions

The way you name classes, attributes and operations is not specified – you can use any names you like. However, a few conventions are considered worthy:

- Use Nouns for class names. Where necessary, concatenate words so that the actual purpose of the class is obvious – for example **AccountAuditLog** could be a class that implemented a log of all account operations for audit purposes
- Use plurals to indicate multiplicity in attributes – for example Transactions, which is the name to indicate multiple transaction objects
- Use verbs for the names of operations. Again, concatenate words where necessary. The name Statement could indicate an attribute of the Account class (which would be an acceptable way to do this, since we can implement a read-only property that returns the text of a statement), but as an operation, **GetStatement** is a more suitable name. Similarly, **Deposit** and **Withdraw** are better operation names than **Deposit** and **Withdrawal**. Note that since the word deposit is available both as a noun and a verb, this is a little more ambiguous. If in doubt, it might be better to use the more obvious **DoDeposit** and **DoWithdrawal**, although in this case I don't feel there is a real ambiguity.
- The names you come up with in the design stages (in use-case diagrams, class diagrams etc.) do not necessarily conform to the naming conventions you might use in code. Provided you stick rigidly to conventions this will not be a problem. For example, many programmers define class names with a **C** or **cls** prefix. Simply add that prefix to the implementation of the classes you design. You would not normally use the prefix in the design stages since it is better to use clearer, less technical terminology at this level (especially if you need to discuss the design with customers).

Inheritance

One feature of class diagrams is that they show how various classes relate to each other – we've already seen that in terms of associations and aggregations. However, the principle of inheritance is also a form of relationship between classes. One class can be said to be an extension of another if it inherits from it and adds properties or methods of its own, or changes any of the properties or methods of the class it extends. Vocabulary to look out for in literature is Generalization-Specialization (the specialized class inherits from the generalized one), Sub-Class and Super-Class (a Sub-Class inherits from a Super-Class), Base-Class (a class that is inherited from) and Derived class (a class that inherits from another).

Inheritance in class diagrams is shown as in figure 6:

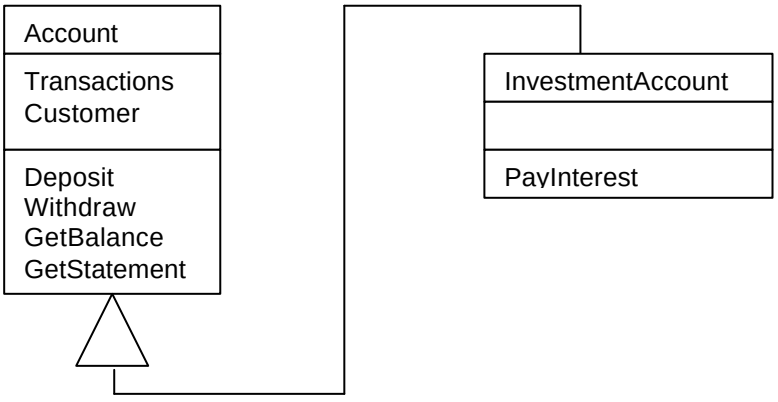

Figure 6: Inheritance in a class diagram

In figure 6, we can see that an **InvestmentAccount** is an extension of a plain Account. As such we can assume that it already has all of the attributes and operations of an Account (there is no provision for *removing* any of these), but extends it by adding a **PayInterest** operation. Programming in C++, Java or Visual Basic 7, you can make use of inheritance directly, but in VB6 it would be necessary to recreate all of the behaviour of the Account class in the InvestmentAccount class. Normally, this would be done by delegation, a form of association in which an object of the sub-class has an object of the super-class as one of its private internal members. It would still be necessary to recreate the

whole of the Account class interface, but each method would simply delegate the work to the internal member of the super-class. For example:

```
' CInvestmentAccount class

Private mvarAccount As CAccount

Private Sub Class_Initialize()   ' Ensure the account object is available.
   Set mvarAccount As New CAccount
End Sub

Public Sub Deposit(Amount As Currency, DateTime As Date, TrType As TransType, _
                   Description As String)
   ' Deposit to the internal account object…
   mvarAccount.Deposit Amount, DateTime, TrType, Description
End Sub

' … Other properties and methods…
```

Of course, the VB7 (VB.NET) version of this is much more straightforward…

```
Public Class CInvestmentAccount
Inherits CAccount
…..  ' Changes to the class
End Class
```

Interaction Diagrams

If you are an experienced programmer and have no need to communicate the way a system works to other people, it would be perfectly possible in small projects to go from a set of class diagrams to an implementation. However, it is normal practice to model how *objects of the various classes* collaborate with each other. As the size of the project increases, these interactions become more complex and some form of notation more necessary. Note, we do not consider classes as items that collaborate; an Object is a model of a real-world item, and we can expect it to interact with other objects, a class is simply a template for a specific type of object.

UML has several ways of depicting the dynamic nature of a system. State diagrams show parts of a system as a Finite State Machine (FSM) and are good for the design of automation systems. Activity diagrams show how complex systems interact in real time, and how temporal dependencies are managed. For our purposes, Sequence diagrams are most useful, providing a general picture of how objects interact.

Sequence diagrams indicate the order that objects interact with each other. A single interaction diagram typically captures the behaviour of a single use-case. For example, if we wish to model the "Withdraw Cash" use-case, we need to depict the interactions that take place to implement the following sequence (taken from the use-case descriptive text):

1. Bank card inserted in ATM
2. User enters PIN – account verifies
3. User selects Withdraw Cash operation
4. User enters amount
5. Amount is checked
6. Card is returned
7. Cash is dispensed
8. Receipt is issued
9. Audit roll is updated
10. Transaction info is updated
11. Receipt is dispensed

The classes involved in this are Account, Transaction, and ATM. The interactions are shown as follows:

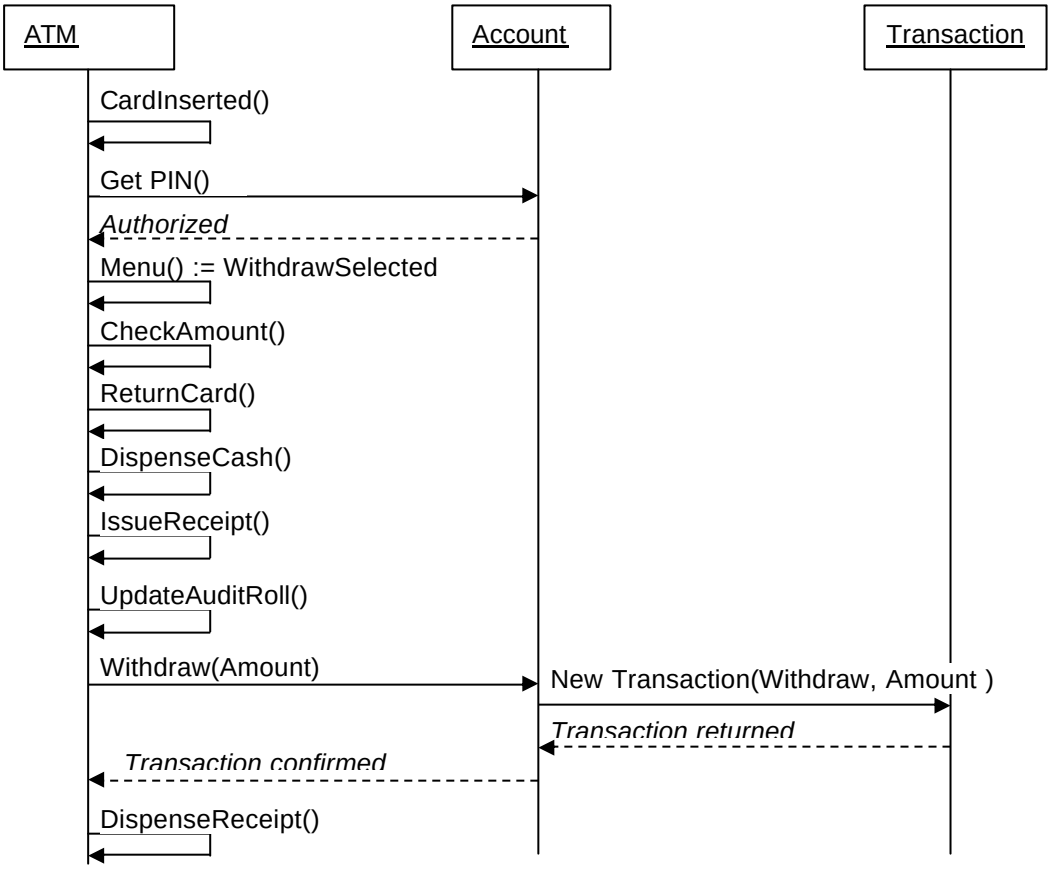

Figure 7: Interactions for the Withdraw use-case

Important features of this diagram are:

- Each object has a time-line, with increasing time running towards the bottom of the diagram. We can tell the relative time of an interaction from this
- Interactions are calls to a method or property of an object, or, as in the case of the Account to Transaction interaction, the creation of an object from the class
- Most of the interactions here take place within the ATM object itself (in a simulation, we might make this a Windows form). These are referred to as Self-Delegations, since one method of the object is executing another method of the same object.
- Returns from an interaction are shown as dashed lines. These are values returned due to a method call (a function) or the creation of an object. Note that one of these interactions (the *Transaction Confirmed* one) indicates that the Withdraw sub really should be implemented as a function, with a Boolean result.

Although these is a lot more to UML than what is described here, these notes should be sufficient to get you started in developing a project with some formality. The Software Development module is about programming, and so I'm not too concerned about the detailed accuracy of your analysis and design diagrams and methods, beyond the fact that there should be some analysis and design and therefore you need to document it.

*For more details on UML and its use in programming, I can recommend **UML Distilled**, by Martin Fowler, as a nice, short introduction. There are other, better guides to UML, notably the standard text by Rumbaugh, Booch and Jacobson, but these are more substantial reads (and more expensive).*