

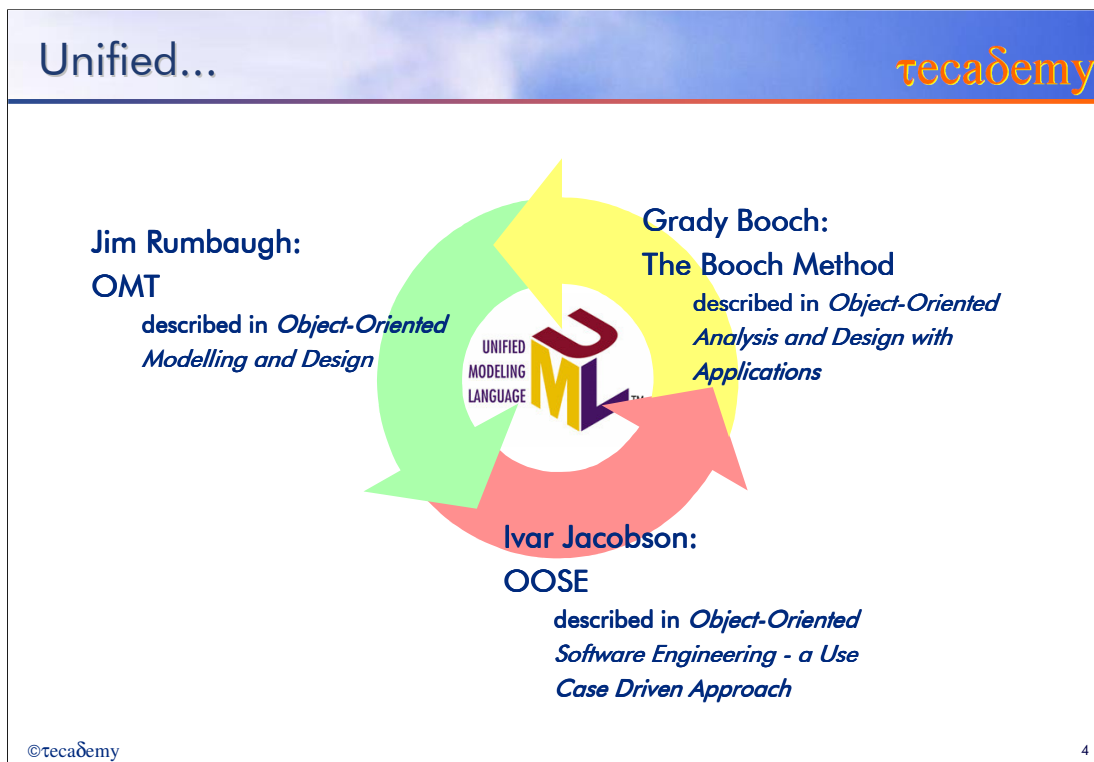
# Overview of the Unified Modelling Language

## Objectives

- This session provides an overview of the important features of the UML notation and its application.

## Contents

- ▶ UML - What is it?
- ▶ The Class Diagram
- ▶ Use Cases
- ▶ Activity Diagrams
- ▶ Scenarios and Sequence Diagrams
- ▶ State Diagrams
- ▶ Communication Diagrams
- ▶ Packages
- ▶ Deployment Diagrams



**unify** [yOOnifI] *v/t* make into one, unite; (*p/t* and *p/part unified*) *i* to have brought together; **unite** [yOOnIt] *v/t* and *i* join into one body; form one body; act together for common purpose; join together; join in marriage; be a connecting link between; come to agreement; form an alliance...

**Jim Rumbaugh's** Object Modelling Technique (OMT) was arguably the most-used of the mainstream OO methods of its time. Evolved in the late 80's from Yourdon's Structured Design method by Rumbaugh and his team at General Electric, it was first described fully in *Object-Oriented Modelling and Design* [Prentice-Hall, 1991]. The clear and expressive notation, tools and techniques, coupled with a credible development process, gained widespread interest amongst those considering an OO approach for the first time. The rapid growth in CASE and training support for the method helped establish it firmly as the most widely used OO development method. In 1995 Rumbaugh joined the Rational Software Corporation with the aim of integrating OMT with the work of Rational's Grady Booch.

**Grady Booch's** method evolved from his work in the Ada community, and was published in *Object-Oriented Design with Applications* [Benjamin Cummings, 1991]. The notation was characterised by the use of a cloud-like icon to represent classes and objects, and the process geared towards the incremental and iterative approach also adopted in OMT. It was weak in the area of analysis, however, and this was remedied in the second edition of Booch's book, re-titled *Object-Oriented Analysis and Design with Applications* [Addison-Wesley, 1994]. The analysis techniques owed much to the work of Ivar Jacobson, whose concept of the 'use case' was also being assimilated into OMT.

**Ivar Jacobson** developed the 'use case' as part of an OO method called Object-Oriented Software Engineering (OOSE). Described in *Object-Oriented Software Development - A Use Case Driven Approach* [Addison-Wesley, 1992], OOSE included guidelines on testing and re-use only briefly covered by OMT and Booch. Jacobson and his team evolved OOSE over a period of 20 years while working at Ericsson in Sweden, and in the autumn of 1995 Jacobson joined Booch and Rumbaugh at Rational.

Jim Rumbaugh, Grady Booch and Ivar Jacobson are arguably the most influential thinkers in the object-oriented community. They have since led the collaboration that has produced the Rational Unified Process (RUP), providing a lifecycle framework and toolset with which applications can be developed. The RUP will be described in more (but not exhaustive!) detail in a later session.

... Modelling... tecademy

Any software application is a model of some aspect of the real world

©tecademy 5

**model** [*model*] *n* something to be copied; small-scale reproduction; simulation, example, pattern; three-dimensional plan; one of a series of varying designs of the same type of object (*pres/part modelling*, *p/t* and *p/part modelled*) *v/t* and *i* form, shape from, plan out; mould, imitate...

In our minds, each of us carries around a model which represents our understanding of the world we live in. This world-model is highly structured and we have a number of representational systems which help us to reason about parts of our model and even extend it to include new knowledge. We rely on the fact that there is a degree of commonality in the world-models of different people to ensure that communication can take place between them.

The value of any model depends on how closely it resembles the original. Of course, a model is not the same as the original, but an abstraction concentrating on the features of most relevance, depending on the context. It can be regarded as a simulation to which the real world artefact or system can be mapped - with the consequence that a change in the original can also be represented by an analogous change in the model.

Any software application we build is, in effect, a model of some process or behaviour in the real world. It can be argued, therefore, that if a software system is a good model then it will have the same 'shape' as the real-world system and will respond to change in the same way. Proponents of object-oriented techniques argue that the 'traditional' practice of modelling the world as data held separate from and processed serially by programs is not a natural one since the world itself is not structured in that way. If we want to build more 'natural' (ie better) models, then it may help to understand how we model things in our head. And the best way to do that is to talk to someone about something they know and listen to the language they use.

... Language tecademy

The UML notation provides an aid to reasoning and a means of communication

©tecademy 6

**language** [*lang-gwij*] *n* system of vocal sounds, by which a group of persons can communicate; graphic representation of this; any system of signs, symbols, gestures *etc* used as means of communication; specialised terminology of a science, profession, craft *etc...*

Language is a technique we have developed to tokenise the concepts and structures that comprise the models in our heads, and communicate them to others. For example, your presenter's world-model includes a structure that represents the field of 'the UML'. The presenter's skill as a communicator depends on his ability to deconstruct this into a set of mutually-recognised linguistic tokens (words), and transmit these tokens serially in such a way that you, the listener, can re-assemble them into a (hopefully identical!) structure and graft it onto your own world-model. The language we use is tightly bound to the models we have in our head, and the more specialised our knowledge, the more specialised the language we use to express it.

The Unified Modelling Language supplies a set of notational elements that map closely to the constructs each of us uses in building and maintaining a mental model, and thus provides a 'set of mutually-recognised linguistic tokens' that allow us to communicate our model to others. Basically, if we can think and reason about something, if we can talk about it, we have a model in our head which we can represent using the UML.



In the remainder of this session, we will discuss these modelling constructs and introduce the UML notational devices that represent them.

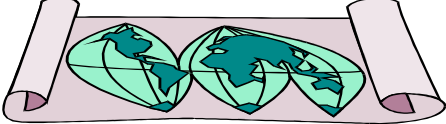
It should be noted, however, that the UML is *only* a set of notational conventions and techniques - it does *not* include a process to guide the development life-cycle. The provision of such a process was the original intent of the authors (in fact, the UML was originally called the Unified Method). It was eventually considered impracticable due to the wide variety of project types likely to be attempted. The authors have stated, however, that any method embracing a broadly iterative and incremental development approach (e.g. the Rational Unified Process, or RUP) would be suitable.

## Why Build Models?

tecademy

- Cheaper than building the real thing
- To provide a clear abstraction of concepts
- To gain understanding, before building
- To provide a common vision for diverse groups (users and developers)
- To manage, scope and document complexity



©tecademy
7

Why do we build models? Simply because they are cheaper than building the real thing.

Models have been built in many industries in order to ensure the success of the final product. Builders of aeroplanes model the aerodynamic behaviour of planes before committing to construction. Architects draw blueprints of floor layouts, and build scale models of houses, shops and office buildings before the construction crew start implementing the reality in bricks and mortar. UML's view of a model is based on that of OMT; it is an abstraction of something for the purpose of understanding it before building it.

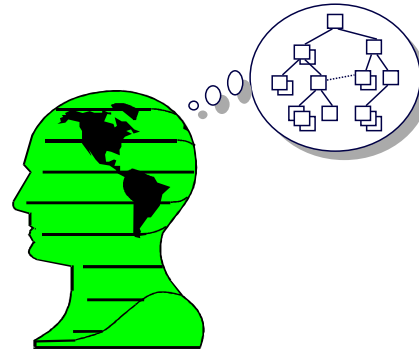
Models are essential firstly for customer communication and secondly for team understanding. Having a common reference of what is being described helps in most aspects of development, but it is particularly important in system development, where we are often creating things that have no physical comparison. Having a common language to communicate and criticise ideas is probably the most important aspect of a development method. The visualisation of what a system may do, before years of code have been developed, can substantially increase the quality of systems and code.

The human mind can only deal with a small number of concepts at once, then quickly forgets the decisions and ideas developed. Managing, scoping and documenting complexity is a primary goal. Most systems being developed today are too complex to understand directly. UML models reduce complexity by separating out a small number of important things to deal with at a time. This is done using a range of models and development phases, where the attention can be focused on the important issues at that point in the development.

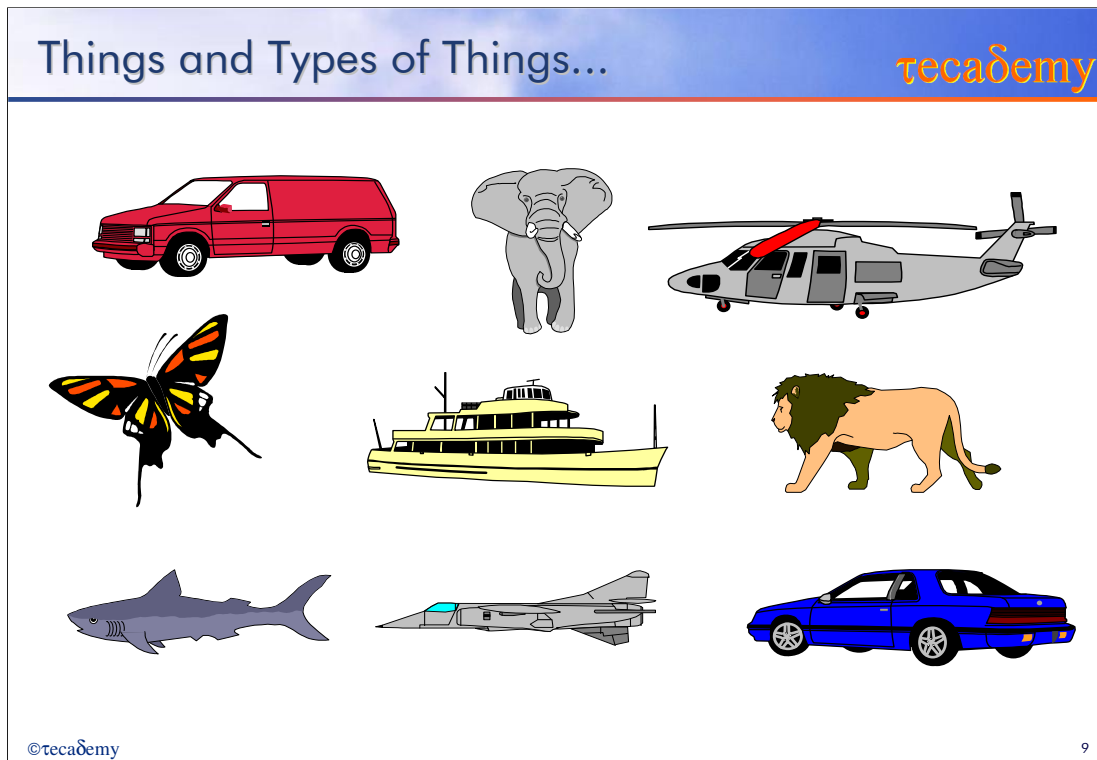
UML models represent knowledge using the same five basic construction elements our brains use to make sense of the world. If there is knowledge, there is a model – the job of the Modeller is to deconstruct this model and transfer it to paper so it can be shared and discussed. Let's look at how models are constructed.

## The Five Elements of a Model

- Things
  - (*Objects*)
- Types of things
  - (*Classes*)
- 'Using' relationships
  - (*Associations*)
- 'Is made up of' relationships
  - (*Aggregations*)
- 'Is a type of' relationships
  - (*Generalisation/Specialisation/Inheritance*)



All the different chess games ever played or to be played are made up of combinations of fewer than ten basic moves. Similarly, the mental world-models of billions of humans, though different, are composed of the same five basic elements. These are described in more detail on the following pages.

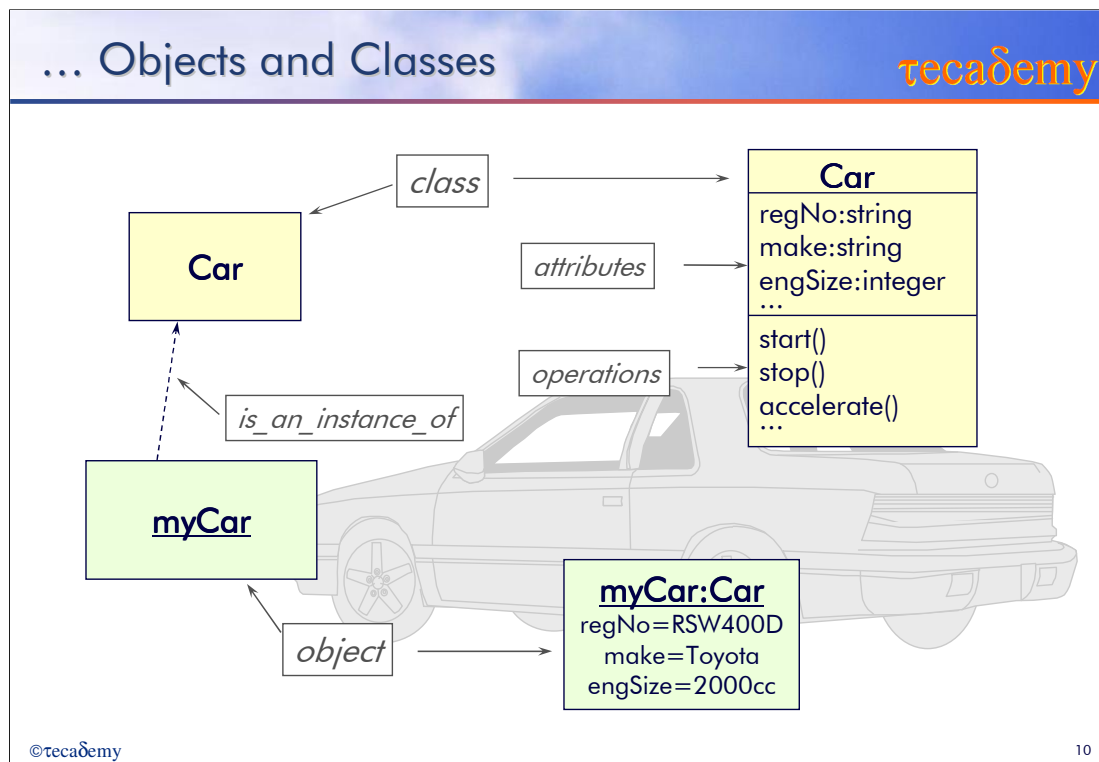


First and foremost, we recognise that the world is made up of *things*. These may be hard, tangible things such as helicopters, printers, people or credit cards; or more conceptual things such as flight plans, fonts, job descriptions or bank accounts. We call these things *objects*. Objects have identity, attributes, behaviours and rules which govern those behaviours.

One of the things that distinguishes we humans from other animals is our ability to reason about not just one particular object, but about a stereotypical object, or an object *type*. Our minds tend to hold generic descriptions of objects having similar properties - we talk and reason about *cars* generally, without needing an actual car to refer to. This process of conceptualisation is called *classification*; your car, his car and her car are all objects of the Car *class*. The class is the second of our world-model constructs.

Classification and the ability to reason about classes forms an integral part of our world-model and governs the way that we learn; from birth, our very first mental act is to classify the world into two types of things - edible things and non-edible things. Everything we have learned since then has built on this classification scheme!

Our classification of objects depends very much on the context in which they appear. In the slide above, possible classifications could include flying and non-flying things. You can probably think of others.



UML notation for class and object are shown above. The class is shown as a rectangle with the class name in bold face. It can be expanded to show all or some of the data *attributes*, with all or some of the *operations* (behaviours) provided by the class shown in a third compartment.

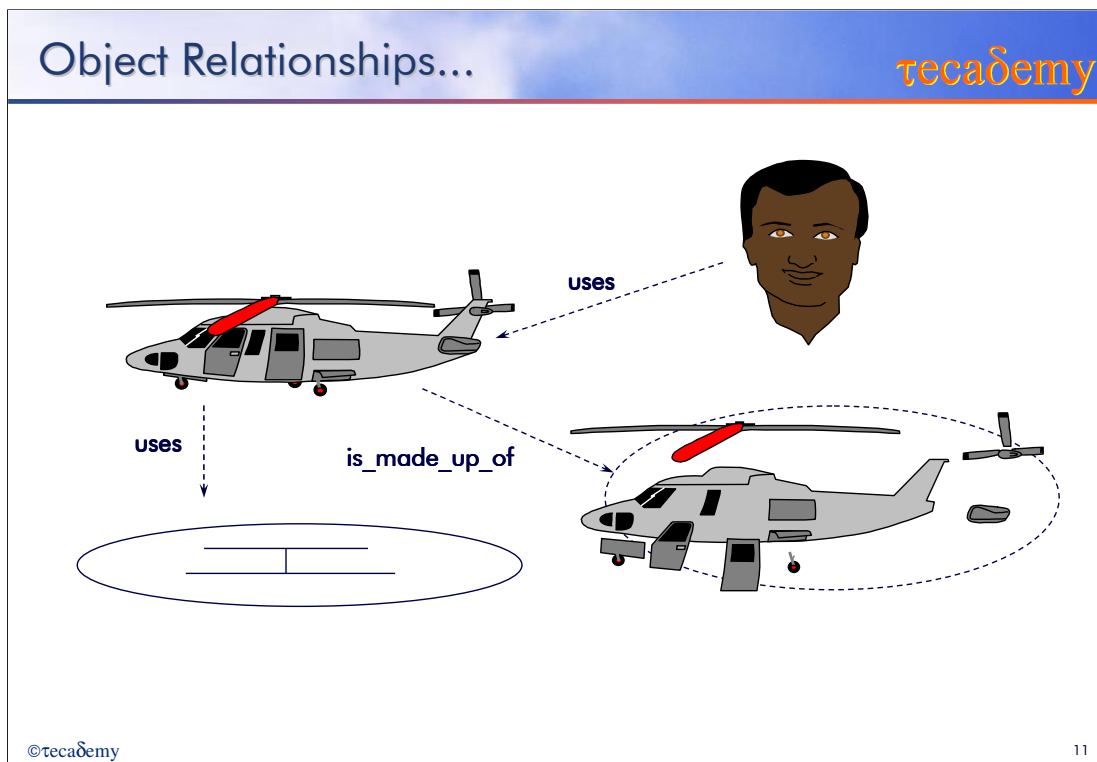
Classes exist to create, or *instantiate*, objects in their own image. An object is an instance of its class for the lifetime of the object. Objects collaborate with each other to perform the dynamic behaviour that makes a system do what it does.

Attributes describe the data an object is responsible for remembering and maintaining. Operations represent the services (or functions) offered by the object. They are implemented by *methods*. An operation is *what* the object does; the method is *how* it does it.

Notationally, objects are also rendered as rectangles that contain the object name, which is underlined. The class to which the object belongs can be shown by drawing a dashed arrow from the object to the class. Alternatively, and more commonly, the class name can be shown following a colon after the object name, in which case the dashed arrow is redundant. The attribute values supplied by the object can be listed, separated from the object name by a line. The operations are not shown, since unlike the attribute values, they will be the same for each object and can therefore be derived from the class.

Objects are *abstractions* – they represent only the features relevant within the context of the problem domain.

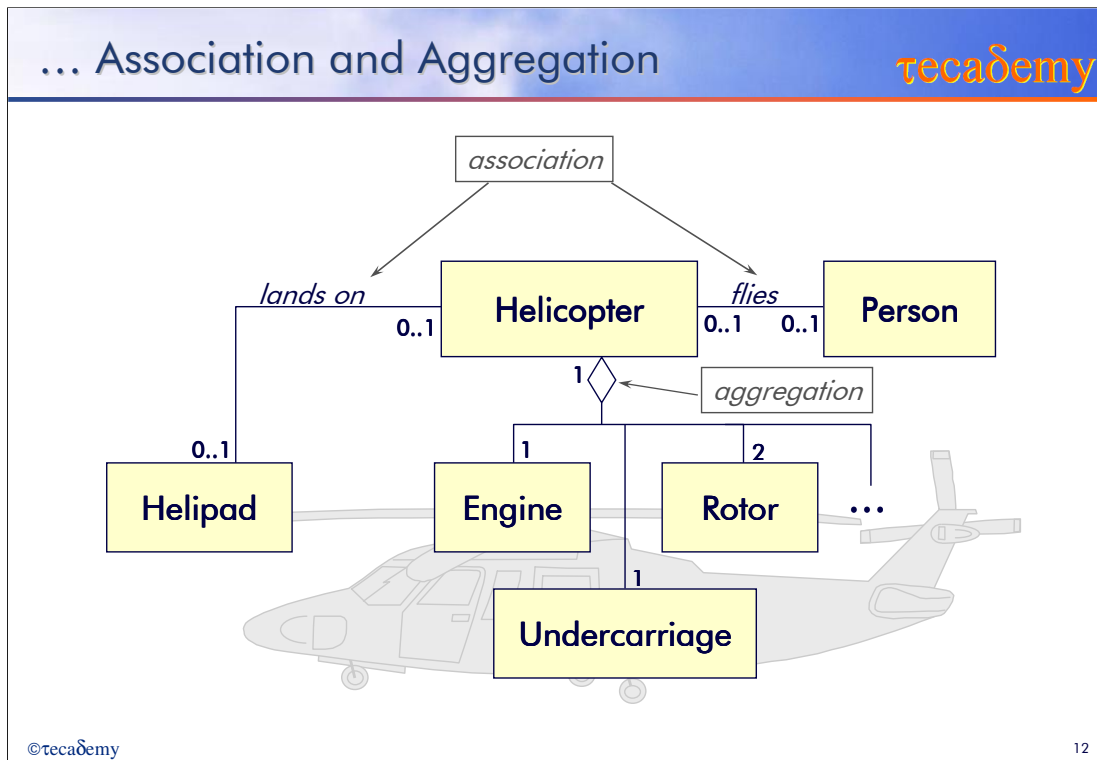
The object's implementation details (such as attributes and methods) are hidden from the outside world – all interactions with an object are conducted via the interface provided by its operations. This principle of *encapsulation* protects the object from being 'interfered' with by its clients; it also protects clients from being affected if the object's implementation should change – provided the interface remains the same!



Things on their own, however, do not do very much; systems tend to be composed of objects using other objects. A 'uses' relationship, or *association*, exists when one object exploits the functionality of another object, in a kind of client-server arrangement - a client object uses the services of a server object. This is a dynamic relationship that can exist for as long as the service is required, and then be broken. This is the third important feature of our world-model, since a system is described as much by its structure as by the objects that participate in that structure.

The fourth component in our conceptual toolset is another relationship that can exist between objects, and mirrors the fact that objects can be made up of, or conversely be part of, other objects. This is a much stronger relationship than the association relationship described above, since it implies the behaviour of a composite object is the result of a close collaboration between its components. In this *aggregation* relationship, the parts exist only for the life of the whole.

As an example of these relationships, a *person* object can pilot (use) a *helicopter* object ; the helicopter object is made up of other objects (rotor blade objects, wheel objects, door objects etc.), each of which in turn can be made up of other objects. The relationship between the person and the helicopter is associative because it can be broken without affecting the viability of the participating objects; the relationship between the helicopter and the rotor blades is an aggregation since the behaviour of the helicopter depends on it having possession of a healthy set of rotor blades.



Notationally, the example on the previous page can be represented by showing associations as lines between classes. The semantics of these associations can be shown by writing an appropriate verb form next to the association line in italics. To aid readability, a triangular arrowhead can be added to show the direction in which the association should be read.

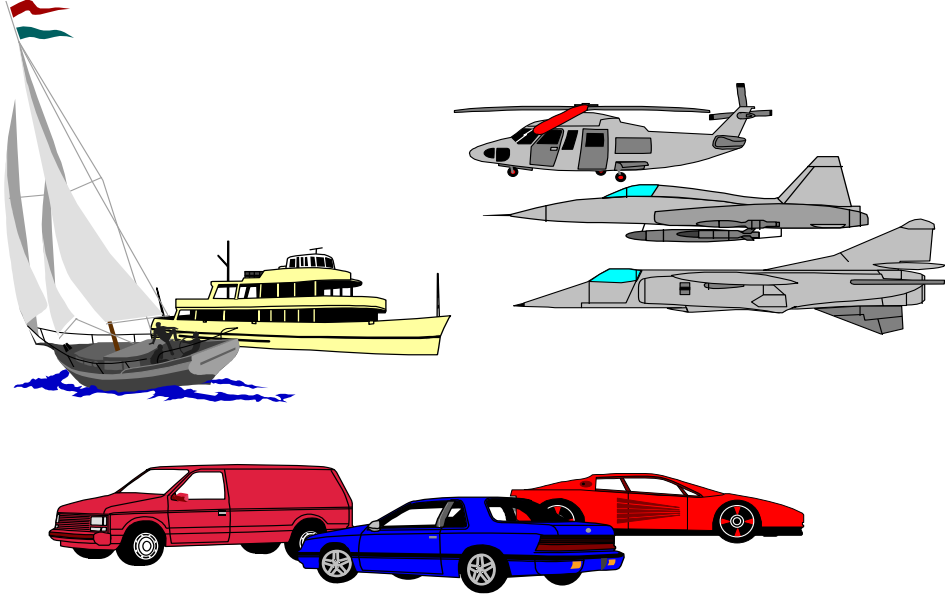
The stronger aggregation relationship is denoted by a diamond at the 'whole' or aggregate end of the relationship. This names the relationship, so no association name is required.

Also shown is the multiplicity of the relationship in both cases. A helicopter will consist of exactly two rotors; one rotor will belong to exactly one helicopter. A helicopter may or may not be flown by a pilot, but if it is, there will be only one pilot.

Other multiplicities can be shown - an asterisk \* means zero or more; one or more, 1..\*; a range, 3..7; alternatives, 5,8,11. When no multiplicity is shown, the default is 'undecided'.

Associations exist to show that an object of one class can call upon the services of another object. The client object does this by sending a *message* to the object supplying the service. A message can be thought of as a function call – the message name should match the name of one of the supplier object's operations. Data can be carried by a message as arguments.

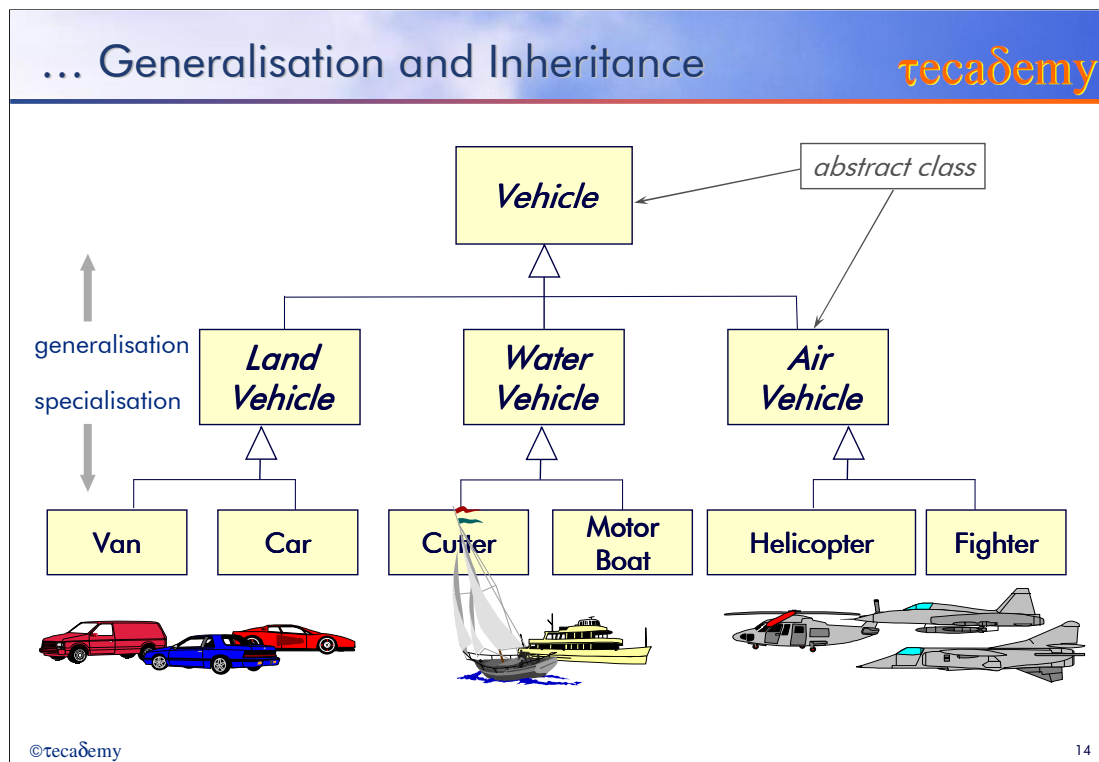
Class Relationships...
tecademy



©tecademy
13

Our fifth and final construct is a relationship that exists between classes. In the above slide, the classes *car* and *van* have properties in common - they can be thought of as special types of *land vehicle*. A further generalisation can be made - land vehicles can be thought of as *vehicles*, and this generalisation includes other classes, such as *cutter* - a cutter is a type of *water vehicle*, which is a type of vehicle. These generalisation-specialisation hierarchies are characterised by this *is-a-type-of* relationship.

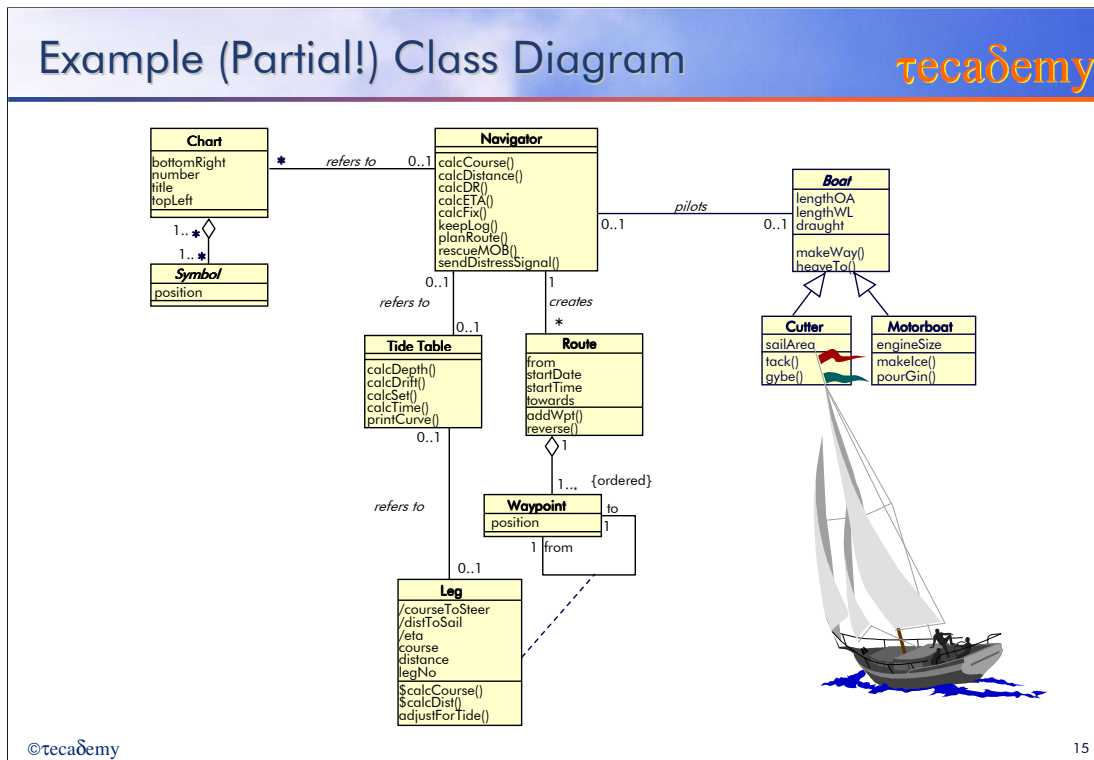
*Inheritance* is a feature of this relationship - water vehicles inherit 'vehicle-istic' properties from the vehicle class, and motor boats inherit water-vehicle characteristics from the water vehicle class. We extend our world-models by this approach - if a new object is described as a kind of car, you instantly know a lot about it, and it only remains to add the details that make it differ from a saloon or coupé or estate.



The UML notation for the generalisation-specialisation relationship is a line with a large, open arrow-head from the more specialised *subclass* to the more general *superclass*. The semantic is *is-a-type-of*; a cutter *is-a-type-of* water vehicle, so the arrow points from cutter to water vehicle. Subclasses inherit all the features of their superclasses – for this reason, the generalisation-specialisation relationship is commonly called the *inheritance* relationship.

Each of the classes at the bottom of the hierarchy - van, car, cutter etc. – is *concrete* (can be instantiated); that is to say, objects like *myCar* and *yourFighter* can be created. Classes like Vehicle are unlikely to have direct instances; although it is true to say *myCar* is a vehicle, it is only a vehicle because it is a type of land vehicle which is in turn a type of vehicle. These uninstanciable superclasses exist as levels of abstraction; they make it possible to think of *all* vehicles in terms of the properties common to all vehicles. They are called *abstract classes* and are indicated by rendering the class name in italics. If italics aren't practical (e.g. when handwriting on a whiteboard), an alternative form is to append {abstract} after the class name, e.g. Air Vehicle {abstract}.

Note that not all superclasses need be abstract, but that all classes at the lowest level of hierarchy *must* be concrete.



Here is an example of a UML model - your presenter will walk you through it. It is part of a description of a system that could be used to navigate any type of boat, but the important thing to realise is that it is based on the real world-model of the navigator - it is nothing more or less than a semantic map, a symbolic representation of part of the navigator's knowledge.

It is important to understand that the UML is purely a notational tool. It can be used to represent a model of anything we can reason about, whether our goal is a re-engineered business, a software application or a set of job descriptions for a new department.

It just so happens that the key modelling constructs we have been discussing - objects, classes, associations, aggregation and inheritance - are all supported directly by any object-oriented programming language (OOPL). This means that, if a decision was made to implement the above model in software, the translation from model to code would be relatively straightforward provided an OOPL was used. More generally, it means that if we can model it, we can probably implement it. It's the modelling that's the hard part!\*

The remainder of this session concentrates on the UML support for the development of such a system.

\*If you doubt this, try constructing a detailed model of something you're familiar with - say, a game of noughts and crosses. You'll wonder how anyone ever manages to play at all!

Use Cases
tecademy

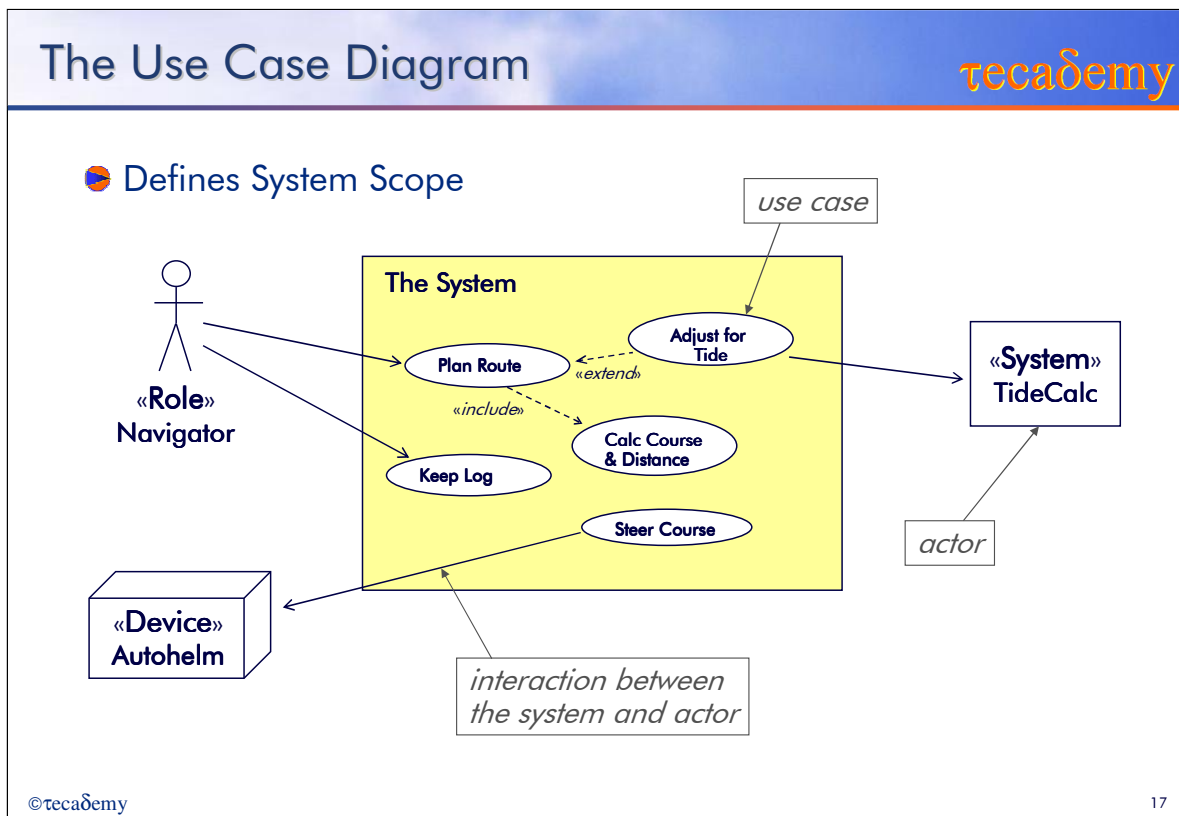
- ▶ **A Use Case:**
  - ▶ Is an interaction between an external entity, or *Actor*, and the system.
  - ▶ Identifies a discrete functional requirement of the system - a 'case' of 'use' of the system by an actor.
  - ▶ Is described in natural language.
  - ▶ Builds into a complete statement of functional requirements.
  - ▶ Can be written by or verified with the customer.

©tecademy
16

A Use Case is a related sequence of transactions that form a discrete dialogue between one or more external entities, or *Actors*, and the system. An example of a use case would be the registration of a new customer (in an Order Processing system, say), or the calibration of a sensor (in an environment control system). An actor could be a human user, or a device, or another system - anything that needs to interact with the functionality of the system. Strictly speaking, an actor represents a particular role that an external entity may play - i.e. the same person might add new customers to the system *and* process orders from that customer and could therefore be represented as two different actors.

A use case, then, describes a 'case' of 'use' of the system by an actor. The description of the use case is written in natural language, with the focus very much on *what* is required from the dialogue rather than *how* it is to be implemented. The full use case description will typically comprise a use case identifier or name, the actor(s) conducting the dialogue, preconditions (what must be true before the dialogue can take place), a description of the dialogue (usually half to three-quarters of a side of A4), a description of the errors or exceptions that may arise, and postconditions (what must be true when the use case is ended).

Identifying use cases also establishes the *scope* of the required system - what it is expected to do, and, just as importantly, what it is not. This information, together with the relationships between the system and the outside world, can be represented graphically on a *Context Diagram*.

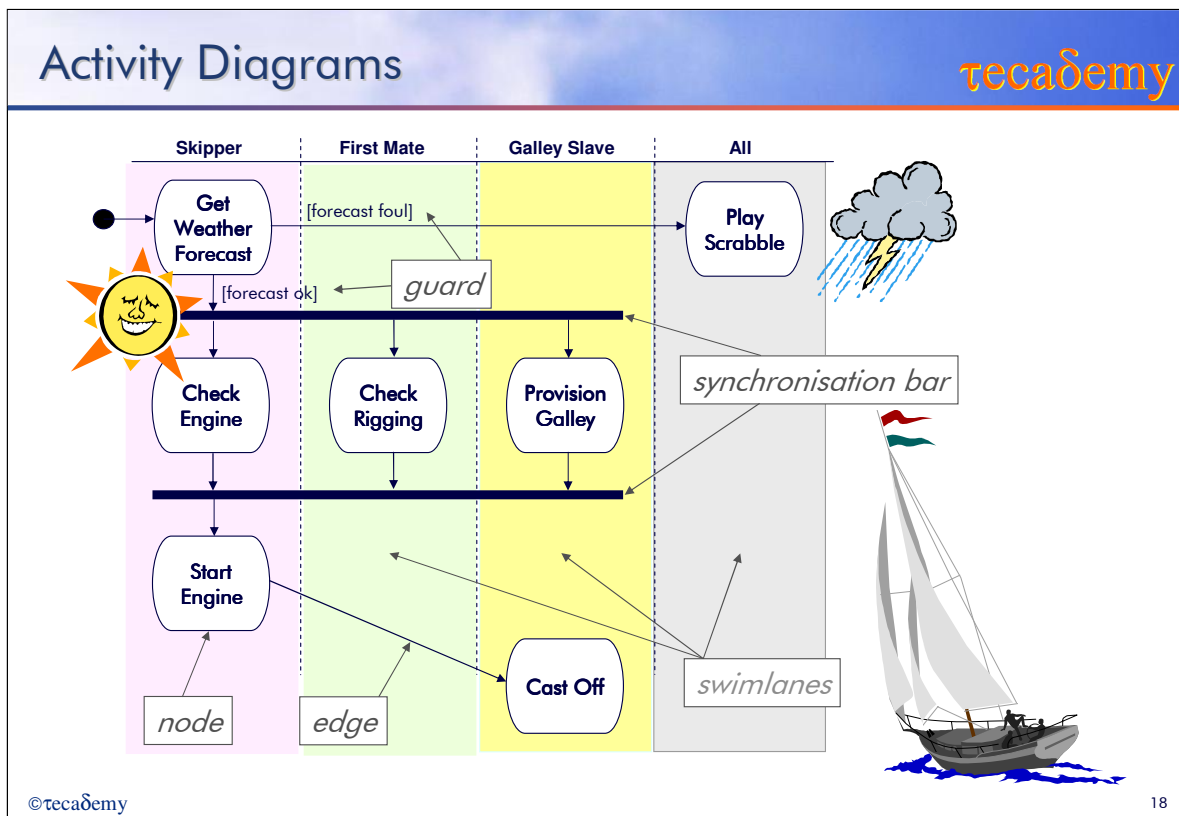


The Use Case Diagram is an ideal tool for visualising system scope and communicating it in an unambiguous way to customers and developers.

The diagram provides much useful input to the modelling process. The identification of the actors (people, other systems etc.) will help to identify those who have an interest in the system and might participate in the development process. The information flowing across the system boundary shows information required or produced by the use cases; these could be prototyped as screens or reports for customer approval.

These use cases, when collected together, form a complete statement of functional requirements of a system. The process of defining use cases requires no specialist IT knowledge, but is heavily dependent on expert domain knowledge for their value. They could, therefore, provide a useful format for the preparation of a requirements document by whoever is responsible for providing the requirements in the first place. They could serve as a deliverable from, say, a Joint Application Design (JAD) planning session. They could also (thinking ahead!) provide the basis of a strategy for customer acceptance trailing prior to delivery.

Identifying use cases is also a useful technique for learning the language of the problem area, which is essential for identifying the key concepts likely to become classes. Another deliverable from this process, then, is the beginnings of the *Data Dictionary*, which is the repository of all knowledge of the system. This is the key document of the development process, and it will be added to and refined at every stage up to and including implementation.




The use case diagram represents clearly what behaviour a system will provide, but it is a static representation - i.e. there is no indication of the sequence in which use cases may be combined to carry out some business process or activity. Activity diagrams supply the time axis, and can be a useful addition to the use case model.

Notationally, steps in an activity are represented in round-ended rectangles called *nodes*; a transition from one node to another is shown with an arrow and is called an *edge*. Completion of one node triggers a transition to the next. A node represents the performance of a task or duty in a workflow. It may also represent the execution of a statement in a procedure. It could be a use case, a step in a use case, or even an operation performed by an object. Activity diagrams can be used at different stages of development - to describe processes in a business model, or the flow of control through a piece of program code.

Synchronization bars are used when two or more nodes can occur in parallel or in any sequence. If a node has more than one possible exit transition, a guard condition can be checked to determine the edge taken. Guards are shown as a boolean expression in square brackets. If a decision is made as a discrete step, it is represented as a diamond-shaped icon with two or more guarded exit transitions (not shown above).

Swimlanes can be used to assign responsibility for a node to an entity (e.g. a job-role, a department, a class etc).

Scenarios


- ▶ **An *instance* of a use case**
- ▶ **Identifies how classes collaborate to enact use case**
- ▶ **Example: Navigator plans a route**
  - ▶ Navigator selects 'plan route' on GPS
  - ▶ GPS creates route
  - ▶ GPS prompts Navigator for first waypoint
  - ▶ Navigator gets the position of the start point from the chart
  - ▶ Navigator passes position to GPS as waypoint
  - ▶ GPS adds waypoint to Route
  - ▶ GPS prompts Navigator for next waypoint
  - ▶ Navigator gets the position of the next waypoint from the chart
  - ▶ GPS prompts Navigator for next waypoint or 'end'
  - ▶ ...

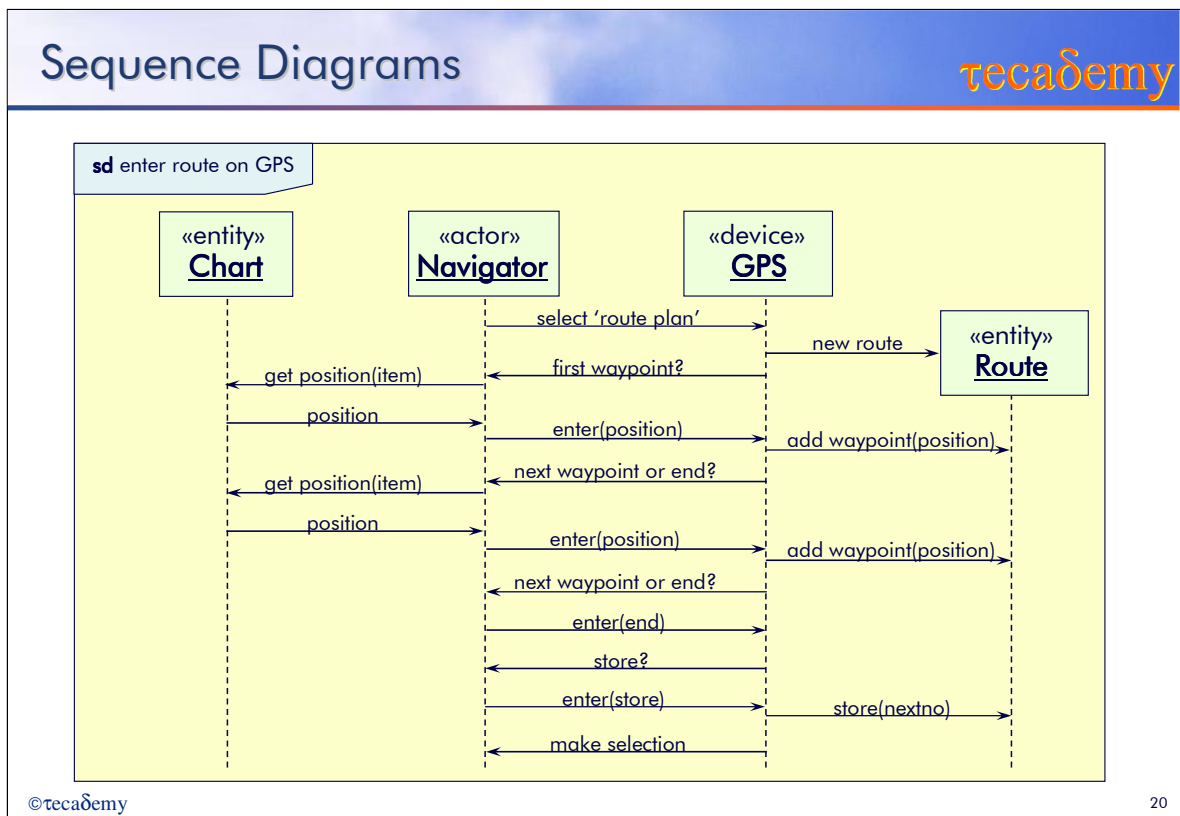
©tecademy
19

If a system can be likened to a play, then the use cases can be compared to the plot-lines that must be begun, acted out and brought to a logically consistent conclusion. Classes can be thought of as the *dramatis personae* that have to collaborate to enact the plot. The task of the playwright (developer) is to develop the characters (classes) to work through the plot (use case) as efficiently as possible without stepping out of character.

There may be many ways of conducting a dialogue described by a use case. There will be at least one successful path (possibly more), and at least one for each of the possible errors or exceptions. Each of these paths can be described as a scenario. A scenario describes a sequence of events that comprise a walk through a use case in terms of the classes and the interactions between them. Just like a script, which is in fact what it is.

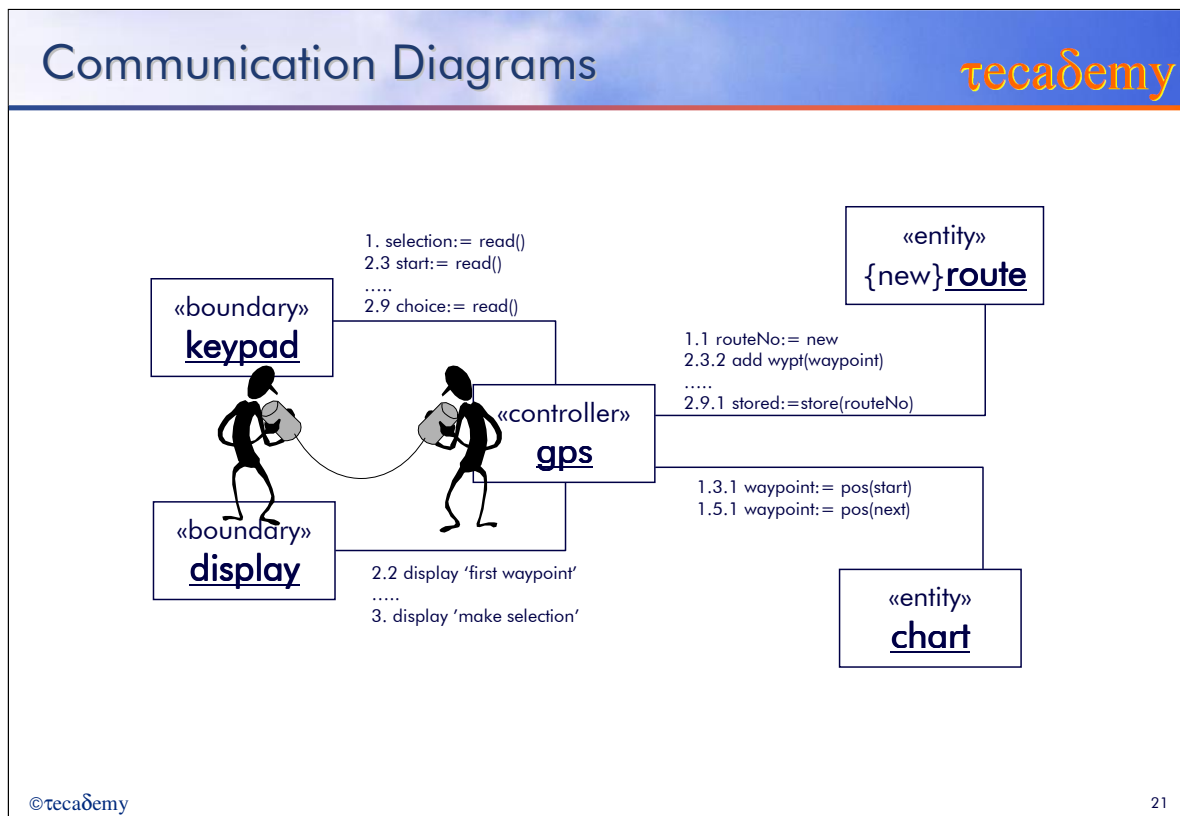
Scenarios are written in natural language, although it helps if that language is simple; line-by-line sentences in a subject-verb-object format of the kind written by small children. This makes it possible (or necessary) to identify who (subject, client class) is doing what (verb, relationship, message) with or to whom (object, server class). Scenarios are written using, as far as possible, the classes discovered from the use cases. If the plot demands behaviour that would take a class 'out of character', a new class is created to provide the appropriate behaviour.

Where use cases provide a strategy for final testing, scenarios represent the individual test cases. Expect a lot of them.



The *sequence diagram* is a graphical representation of a scenario. It is constructed by representing the objects (nouns in the scenario) with vertical lines, and the messages (verbs) with arrows pointing from the client (subject noun) to the supplier (object noun). Time, and therefore sequence, flows down the page. A sequence diagram is presented in a box. The diagram title is prefixed with 'sd' and shown in a pentagonal compartment embedded in the top left-hand corner.

Representing scenarios in this way often reveals, in a way that text can't, opportunities to optimise the messaging that goes on, making it possible to identify necessary operations, their parameters and their result types. This information can be fed back to the class definition in the data dictionary. Looking at the individual object lines makes it possible to identify the events to which an object can respond (the arrowheads hitting the line), and their behaviour in response to those events (the arrows leaving the line). This information can be summarised, for each object in each scenario, on a state transition diagram.



Another way of graphically depicting a scenario is the *communication diagram* (formerly known as the *collaboration diagram*) shown above. Objects are represented by the icon we met earlier, a line between two objects shows that messaging between them occurs, and the actual messages sent are written explicitly showing the direction of invocation. Sequence is indicated in a way that reveals the dependencies between the operations (hence the 'nesting' of sequence numbers - 2.1 happens as a result of 2, 2.1.1 as a result of 2.1 and so on).

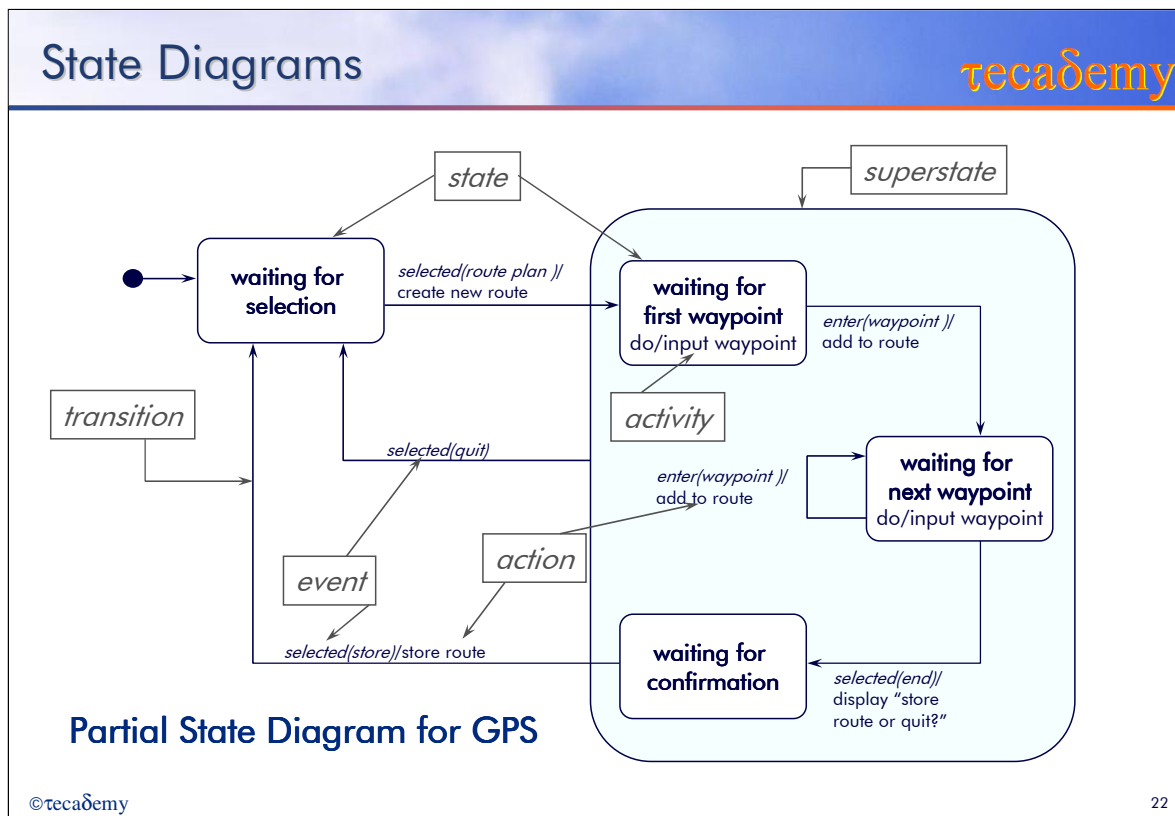
In the design phase, communication diagrams can also be useful in modelling different types of object communication, such as synchronous versus asynchronous, or timeout and balking behaviour.

The information content of a communication diagram is exactly the same as in a sequence diagram, to the extent where a CASE tool will generate one from the other and toggle between them at a touch of a function key.

So what is the difference between the two - why have both?

The obvious difference is the immediate appearance. In the sequence diagram, the order of events is easy to read - in the communication diagram, it must be followed through the numbering scheme. On a large diagram this might not be easy. The state behaviour of an object as a result of events is also not so apparent. The communication diagram, however, shows the pattern of objects more clearly - replace the objects with classes, the message lines with associations, and the messages as operations on the target class and the result is a class diagram.. The class diagram can be thought of as the sum of all the communication diagrams.

If the class diagram can be derived from the communication diagrams, which are derived from the scenarios, which are in turn derived from the use cases that form the requirements, then the class diagram can be shown to meet the requirements - an important contribution to the validation process.



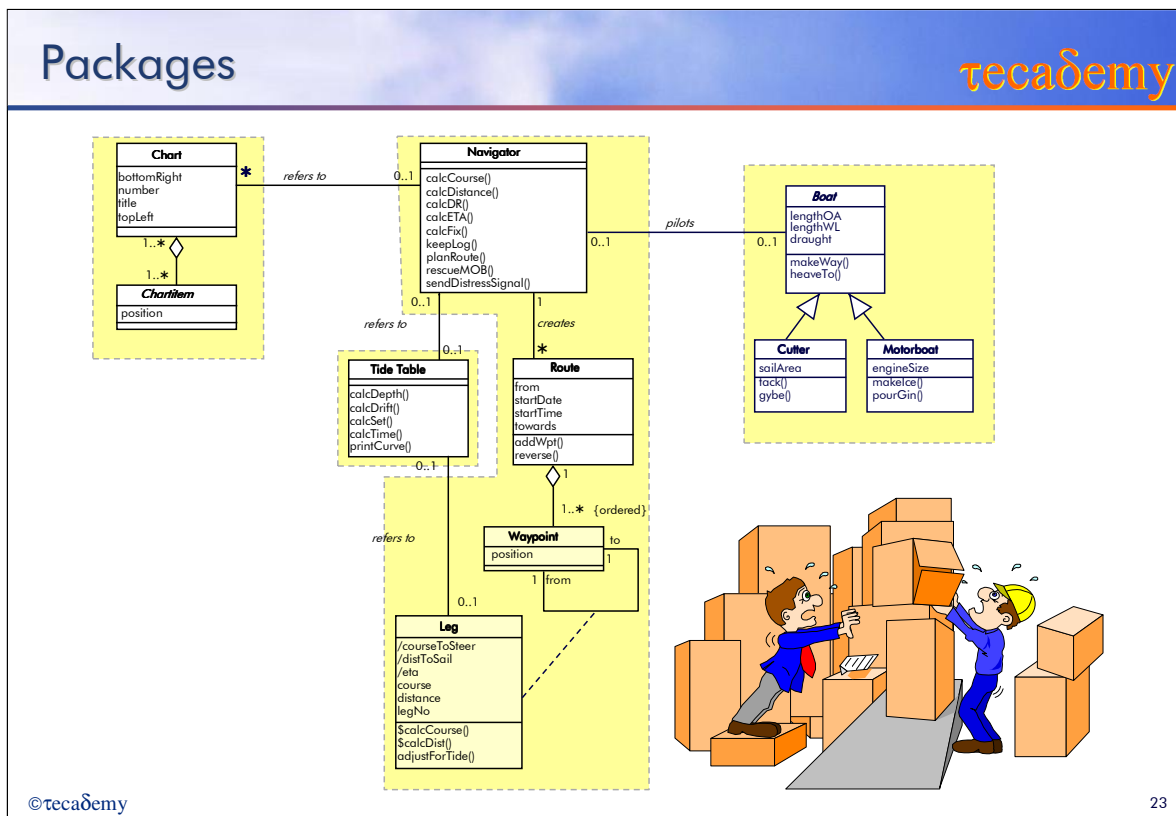
*State diagrams*, sometimes called *State Transition Diagrams (STDs)* or *Statecharts*, show the behaviour of a single object over its lifetime. They show the states an object can be in, what they do while in a particular state (activity), the events (or messages) to which they can respond while in that state, what they do in response to events (operations), and what state they move to as a result (transition).

States are often implemented by attribute values - in order to define a state, it may be necessary to add new attributes to the appropriate class in the class diagram. Activities and operations also point to the methods that a class will need to provide in order to implement its behaviour.

All the information in a STD can be derived from the sequence diagrams on which an object appears - it is built up on a scenario-by-scenario basis until all the scenarios have been considered. The events come from the messages sent to the object, the states from the gaps between those incoming messages, the operations and activities from the outgoing arrows.

Not all objects will have interesting state behaviour - those that do will often play a strong controlling role in the system and will therefore need to be investigated carefully to ensure all possible state/event combinations have been considered. This may lead to the development of new scenarios or even new use cases.

State diagrams can also be drawn to model system behaviour, where use cases are the events that cause the system to transition from a precondition state to a postcondition state.



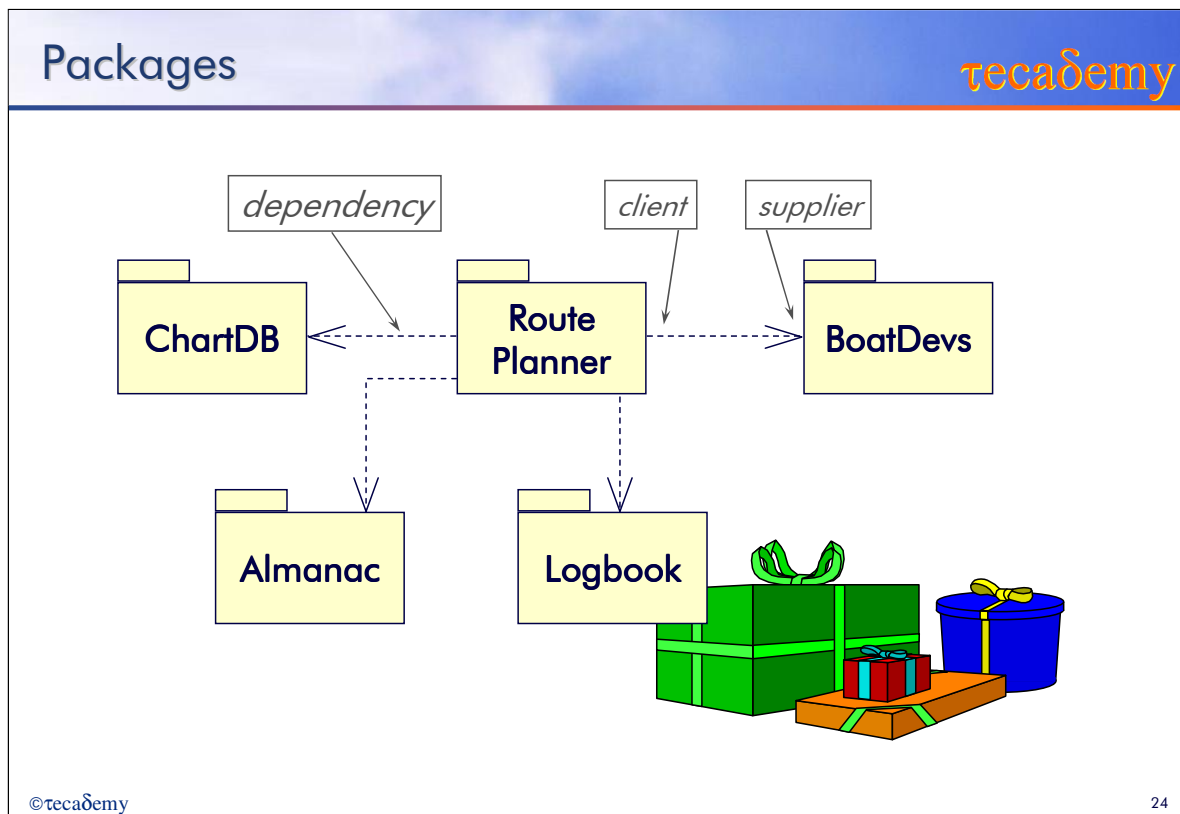
Class Diagrams can easily consist of several hundred classes. Large systems require the definition of many classes, leading to complex class diagrams running to many pages. Viewed as a whole, such a system would present a very large surface area, which would make understanding difficult. Closely-related portions of the class diagram may be encapsulated and abstracted into separate groups of classes called *packages*.

Each package provides a major architectural function, and can be identified from the class diagram by looking for strongly-coupled, or functionally related classes. The coupling between two classes is a measure of how much they depend on each other. Classes tend to 'clump' together, particularly around aggregation or inheritance hierarchies; these 'clumps' may be more loosely associated with other 'clumps'.

Package boundaries tend to align with 'clumps', cutting through the relatively sparser associations between them. They are used to partition large models into manageable parts. Each package can contain related classes, and each class is owned by a single package. They can be nested - a package can contain other, lower-level packages as well as classes. Packages have no semantic content; they are used only as a way of organising the model.

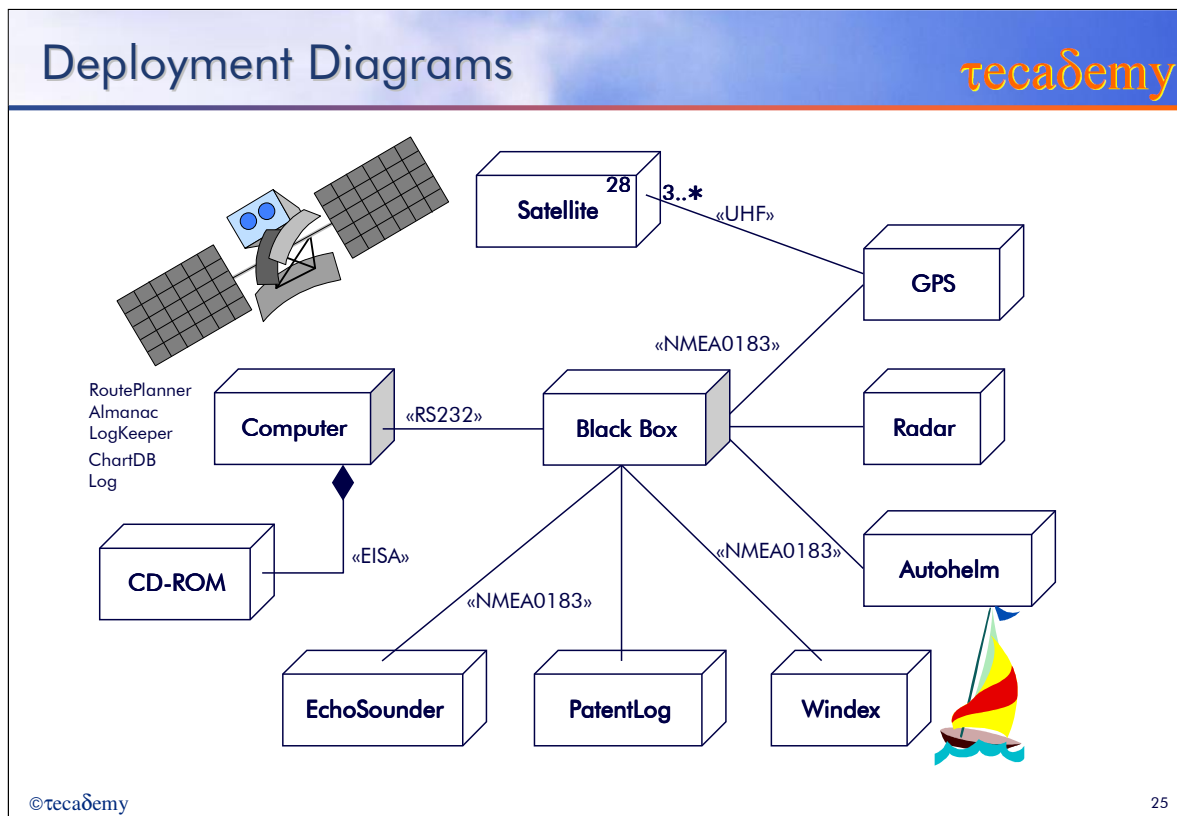
Large models may be partitioned into packages in a top-down manner at the start of, or even preceding, analysis, or they may be partitioned bottom-up by grouping related classes during analysis or design. Additional design-oriented packages are likely to appear during design, containing, for example, GUI or database classes. Thus packages provide a reuse mechanism at a level higher than individual classes.

Obviously any model may be split up in different ways. The goal is to produce packages of a manageable size, which are reasonably cohesive in that they package related components, and which minimise the complexity of the interfaces, or coupling, between packages.



The UML notation for a package is a 'tabbed folder'; these can be shown with just the package name or can be expanded to show nested packages or classes owned by the package. Relationships between packages are identified by looking at the associations that cut across the package boundaries and representing these with a dashed line. An arrowhead can be used to show the direction of dependency - a single arrowhead implies that there is a client/supplier relationship between the packages (i.e. navigation is one way, from the client to the supplier); a double-headed arrow implies a peer-to-peer relationship (i.e. message traffic is two-way).

Packages add value, in addition to breaking up a large model; the package diagram often describes a high-level structure inherent in the problem domain. Packages can also be used to group together use cases that are related by functionality or priority.



At some time it will be necessary to decide on the hardware that will host our system, and make decisions about how that hardware should be arranged and which hardware elements will run which part of the system. Even if a system architecture is already in place, as is often the case, the software will still need to be deployed optimally across the piece. *Deployment diagrams* are used to help model possible solutions and to describe the one selected.

In the UML, a hardware element containing some computational resource is called a *node*, and is represented on a deployment diagram by a three-dimensional rectangular solid. The 'solid' icon is intended to convey the idea of physical presence - these boxes are *real*. Another way to think of them is as classes that are to be implemented as hardware.

Each node should be named according to the type of hardware it represents, e.g. PC, printer, sensor etc. Shading can be used to distinguish between *processors*, i.e. those nodes that will host our system components (e.g. server, workstation), and *devices* that have no computational relevance to the system (e.g. printer, disk drive).

The components that will reside on a processor are shown as text placed next to the appropriate node. Mapping the logical components of the system to the physical in this way helps to identify possible performance bottlenecks (e.g. network traffic, or 'what happens if the server goes down?'-type questions) and to model alternative configurations.

Where two elements are connected, a line is drawn between them. Connection types can be designated with stereotypes if it aids clarity; for instance, «ISDN». Where more than one element of a type can be involved in a connection, the multiplicity can be shown in the top right-hand corner of the node's 'face'. Where a device is physically contained in another, for instance, the disk drive in a PC, the composition diamond can be used.

## Summary

- The UML is a language for representing and communicating knowledge
- It is not a method
- Notation includes support for the whole OO development life-cycle
  - Use Cases
  - Activity Diagrams
  - Scenarios and Sequence Diagrams
  - State Diagrams
  - Communication Diagrams
  - The Class Diagram
  - Packages
  - Deployment Diagrams
- The UML is the industry standard notation

## Check your Understanding – Pair these up

1. Association

A. An 'is-a-type-of' relationship

2. Attribute

B. A thing of interest in the system

3. Class

C. The term for 'create' an object

4. Aggregation

D. A 'uses' relationship

5. Inheritance

E. A 'whole-part' relationship

6. Instantiate

F. How objects communicate with each other

7. Message

G. Data an object maintains

8. Object

H. Describes objects of the same type

9. Operation

I. A service or behaviour offered by an object

Instructions: match the words on the left to the definitions on the right by drawing an arrow from one to the other.

## Summary : Check your Understanding

- |                |   |
|----------------|---|
| 1. Association | <u>D. A 'uses' relationship</u>                       |
| 2. Attribute   | <u>G. Data an object maintains</u>                    |
| 3. Class       | <u>H. Describes objects of the same type</u>          |
| 4. Aggregation | <u>E. A 'whole-part' relationship</u>                 |
| 5. Inheritance | <u>A. An 'is-a-type-of' relationship</u>              |
| 6. Instantiate | <u>C. The term for 'create' an object</u>             |
| 7. Message     | <u>F. How objects communicate with each other</u>     |
| 8. Object      | <u>B. A thing of interest in the system</u>           |
| 9. Operation   | <u>I. A service or behaviour offered by an object</u> |