Introducing UML

Unified Modelling Language

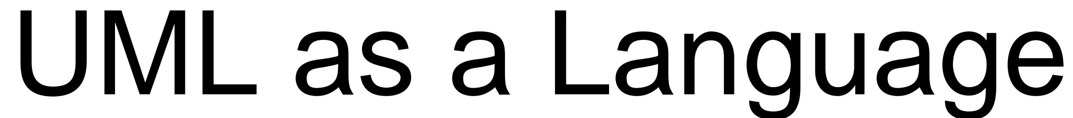
© Garth Gilmour 2008

garth@ggilmour.com

- UML is a language which is used graphically
  - Its purpose is to build models of software
- It is the only industry wide OO modeling notation
  - It was created by merging earlier standards
  - There is extensive tool support for using UML
    - For example the Eclipse Modelling Framework (EMF)

> The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing and documenting the artifacts of a software-intensive system

© Garth Gilmour 2008



UML as a Language

- The OMG controls the UML metamodel:
  - Standard elements and how they are created
  - Relationships that can exist between elements
  - How to extend the built-in elements
- A UML meta-metamodel has also been defined
  - This allows mappings between UML and other notations
- UML can be used directly as a programming language
  - This has been referred to as ‘Model Driven Architecture’ or ‘Executable UML’ and attempts to dispense with coding entirely
  - Until tool support improves this is unlikely to catch on

© Garth Gilmour 2008

Visualising Code

- Code is easy to visualise when it is being written
  - But hard to understand in retrospect
    - By another developer or yourself in a months time
- There are techniques that minimise this problem
  - By maximising quality and keeping the code alive
    - This is the idea of ‘self-documenting code’
  - The ‘Refactoring’ movement has helped enormously
    - Refactoring support is now built into the major IDE’s
- Even with the best code a picture tells a thousand words
  - It provides ‘jumping off’ points into the source code

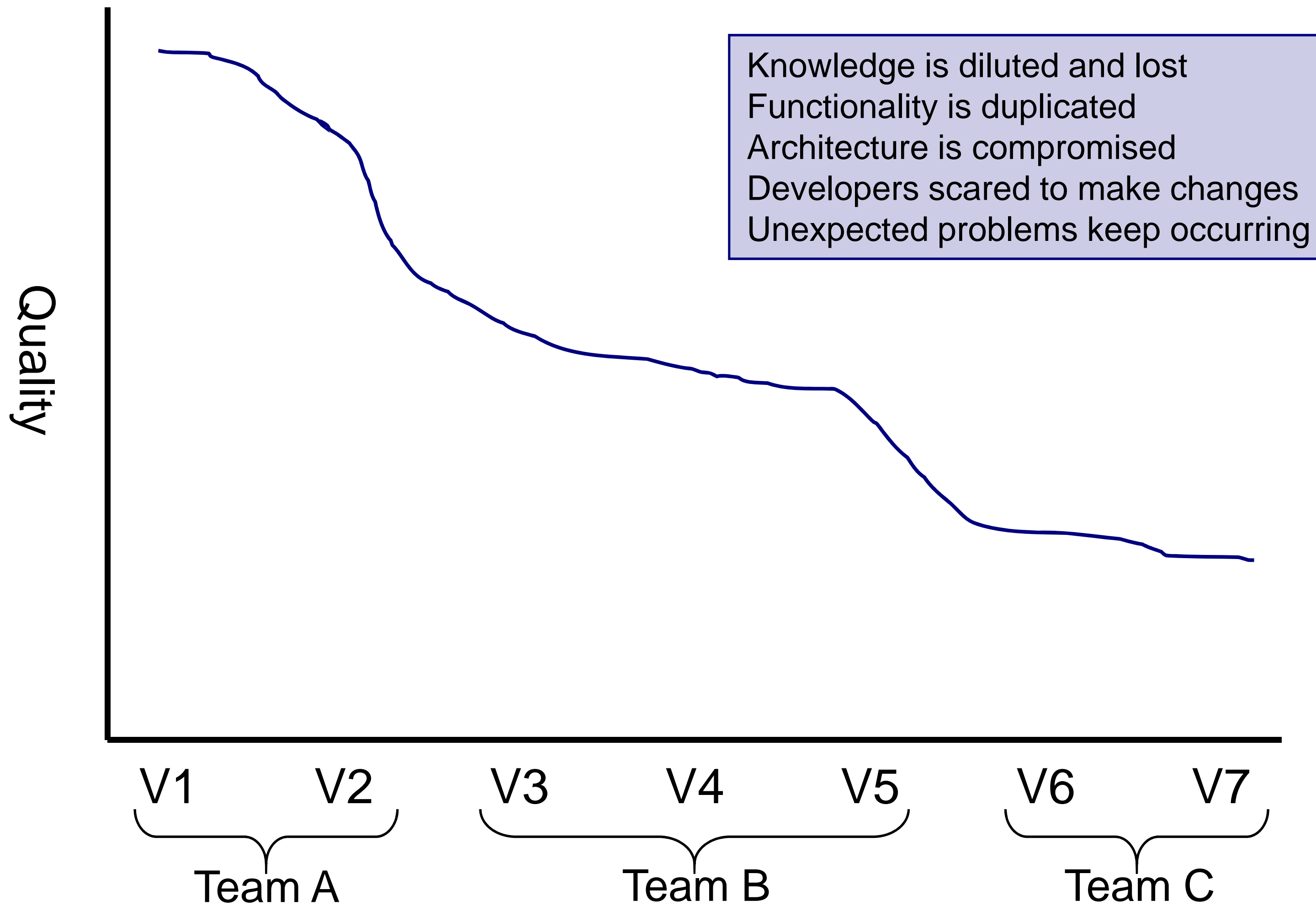
© Garth Gilmour 2008

Documenting Your Design

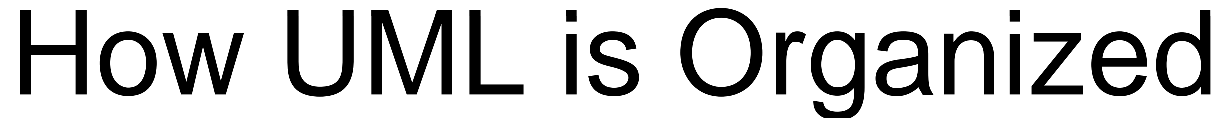
- Software spends less than 10% of its life in development
  - The rest is spent being used, maintained and extended
- Development teams typically don’t look far beyond V1.0
  - They are not encouraged or motivated to do so
  - Many projects pay the price at version 1.1 or 2.0
- A proper UML model is critical to documentation
  - The system can be ‘mothballed’ safely
  - New developers will have a starting point

© Garth Gilmour 2008

Software Over Time



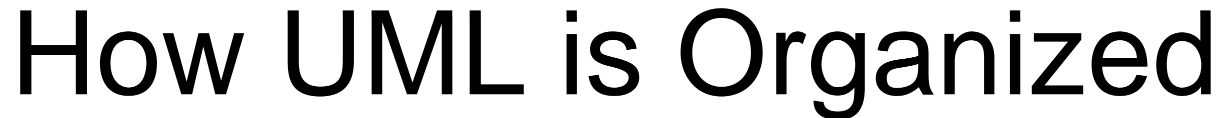
© Garth Gilmour 2008



How UML is Organized

- In theory UML provides different views of your system
  - This is known as the 4+1 architectural view
  - In practice UML is best at the logical view
- The Requirements View organises your functional requirements into Use Cases
  - Use Case Diagrams provide an overview
  - Use Case Reports provide the details
- The Logical View specifies your design as:
  - The static structure of your classes
  - The dynamic ways in which your objects collaborate

© Garth Gilmour 2008

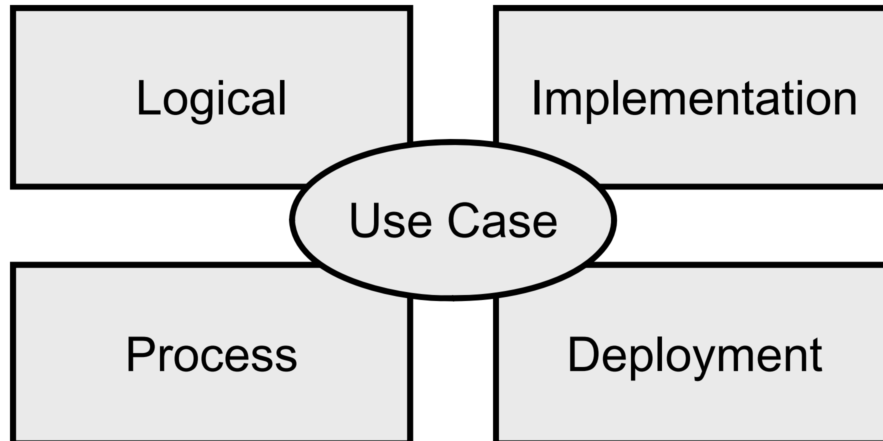


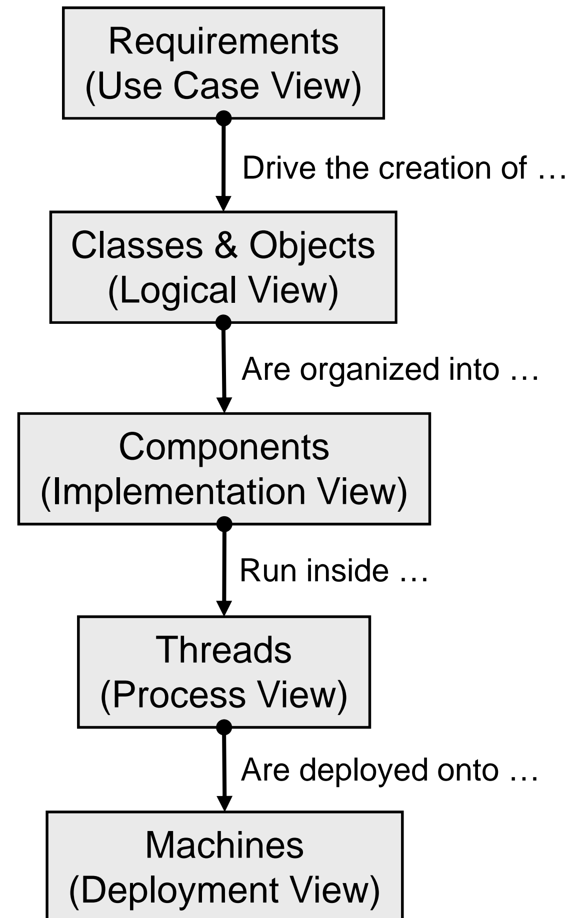
How UML is Organized

- The Implementation View represents your build
  - When compiled into libraries and executables
  - The architecture is compressed into a set of components
- The Process View represents the runtime
  - How your components are run on processes and threads
  - This is the weakest part of the UML
- The Deployment View represents installation
  - Onto ‘nodes’ which represent different types of hardware
  - Either in test configurations or on-site

© Garth Gilmour 2008

The 4+1 Architectural View





© Garth Gilmour 2008

The UML Diagrams

Diagram	Description
Use Case	Provides an overview of requirements
Activity	Details a flow of events (usually requirements)
Class	Defines a set of classes and relationships between them
Sequence	Illustrates how messages pass between objects
Collaboration	Same as above but from a different perspective
Object	Shows the values within a set of objects
State	Details the lifecycle of an object
Component	Defines a set of components
Deployment	Describes how components are placed in nodes

© Garth Gilmour 2008

How to Apply the UML

- Modelling is not an ‘all or nothing’ activity
  - There are a range of possible applications
    - Depending on the size, scope and budget of the project
- Tool vendors promote extensive use of UML:
  - Building both analysis and design models
  - Fully modelling each function of the system
  - Using UML to auto-generate code
  - Automatically synchronizing models and code
- This level of usage is not suitable for everyone
  - It works for large, distributed projects
  - Especially those with separate architectural teams

© Garth Gilmour 2008

Pragmatic UML

- UML should be used pragmatically
  - According to the scope of the project
- Pick those diagrams that are most useful to you
  - Use Case, Class and Sequence diagrams are most helpful
  - Other diagrams can be used where appropriate
- Do as much modelling as your project requires
  - Build a model that captures the essence of your system
  - Think from the perspective of new hires or the team that will maintain and extend your code in the future

© Garth Gilmour 2008

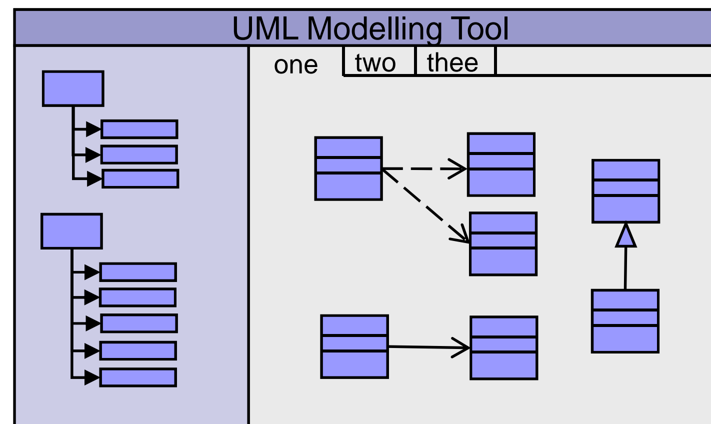
Advice About Using UML Editors

- UML editors can be subdivided into:
  - Those which are mostly a tool for drawing diagrams
  - Those which are a tool for creating a model
- Each has its own advantages and drawbacks
  - A diagramming tool lets you draw UML diagrams quickly and easily, possibly with additions from other notations
    - However a diagram can only be printed, saved or exported as an image
    - If you make a change to one class you must alter many diagrams
  - A modelling tool builds up an internal model of your design which is represented graphically in a set of diagrams
    - When you change part of the model all the diagrams are updated
    - The model can be used for code generation and may support scripting
    - However the tool will have a steep learning cure and be prone to errors
      - A action that would be natural in a diagramming tool may corrupt the model

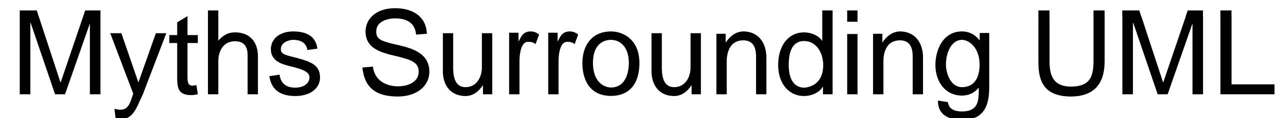
© Garth Gilmour 2008

Advice About Using UML Editors

- The standard layout of a modelling tool is given below
  - The diagrams on the right are representations of the information stored in the model shown on the left
- Remember that the model is most important
  - Always create something in the model first and then drag it into the appropriate diagram (the reverse may case problems)



© Garth Gilmour 2008



Myths Surrounding UML

- You can use UML before choosing a coding language
  - This never happens in practice and would be a waste of valuable development time because of the amount of rework required
- UML requires expensive tools
  - A whiteboard is all you need…
- Only UML diagrams can be used in a UML design
  - You can supplement the UML with anything else you like
- All the design must be complete before you write code
  - In an iterative process we alternate between the two
- The design is more important than the code
  - Your customers are not paying for diagrams…

© Garth Gilmour 2008

Details Shown On UML Diagrams

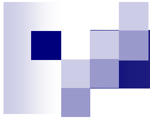
- A common concern is the level of detail shown:
  - Is there enough detail on this diagram?
  - How much is too little or too much?
  - Can I start writing the code now?
- When learning the UML we want a definitive answer
  - We take our cue from the examples in books and slides
- You need to develop an awareness of:
  - When you have done as much abstract design work as possible
    - Time to write code to validate the structures you have visualized…
  - When you are becoming tied up in the minutiae of code
    - Time to step back from the code and re-evaluate your UML

© Garth Gilmour 2008

Details Shown On UML Diagrams

- In real life diagrams start vague and become precise
  - The first draft captures the essentials and is low on detail
  - Subsequent drafts add detail and bring us closer to code
  - The final version is an accurate representation of the code
- Don’t be afraid to use the UML for ‘sketching’
  - Your diagram has value as long as it clarifies your thinking
- However don’t use ‘sketching’ as an excuse
  - Too many projects sketch some vague UML diagrams and hope the details will sort themselves out in code
  - This will only happen in small projects with open lines of communication and talented and experienced developers

© Garth Gilmour 2008



Analysis Level Diagrams

Help understand the problem
Sketching a solution with few details
Large ‘chunky’ classes with many jobs

‘Spike’ Applications

Proof on concept programs that confirm you can use a particular technology and that it can accomplish what you require

Design Level Diagrams

Represent a possible solution
Names, signatures, relationships
Enough detail to move into coding

Implementations

The application is incrementally built over a series of iterations, preferably with unit, acceptance and non-functional tests

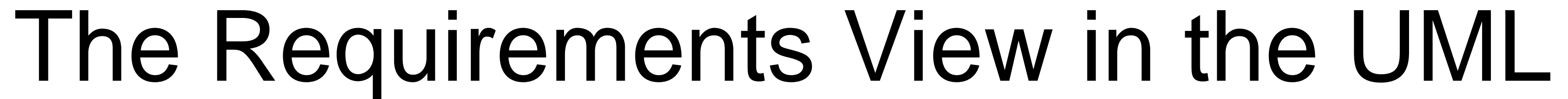
Blueprints of Code

Full detail of implementation
Notes attached to describe intentions
Stereotypes used to show usage of patterns etc…
Usually reverse engineered from completed code

© Garth Gilmour 2008

Creating Use Cases

© Garth Gilmour 2008
garth@ggilmour.com



The Requirements View in the UML

- The UML captures requirements as stories
  - Use Cases are stories written in a structured way
- A story describes how the user and system interact
  - In order to accomplish a particular goal
- The story is told from users point of view
  - Users discover what they want from the system through discussing existing procedures and imagining possible scenarios
  - Stories can start as very personal and then be generalized to apply to a particular set of stakeholders

© Garth Gilmour 2008

The Requirements View in the UML

- Use Cases are not about dissecting out details
  - They raise the level of abstraction by grouping features into ‘chunks’ that are easier to understand
  - Developers must understand the stories behind the system in order to be able to put features in context
    - Without this context it is not possible to make the informed choices required to produce a proper architecture
- The detail you need depends on your process
  - Stories in the RUP end up being fully specified
    - The documentation may run to 15 pages per story
  - Stories in Agile methods only exist as index cards
    - Because the details can be discovered as required

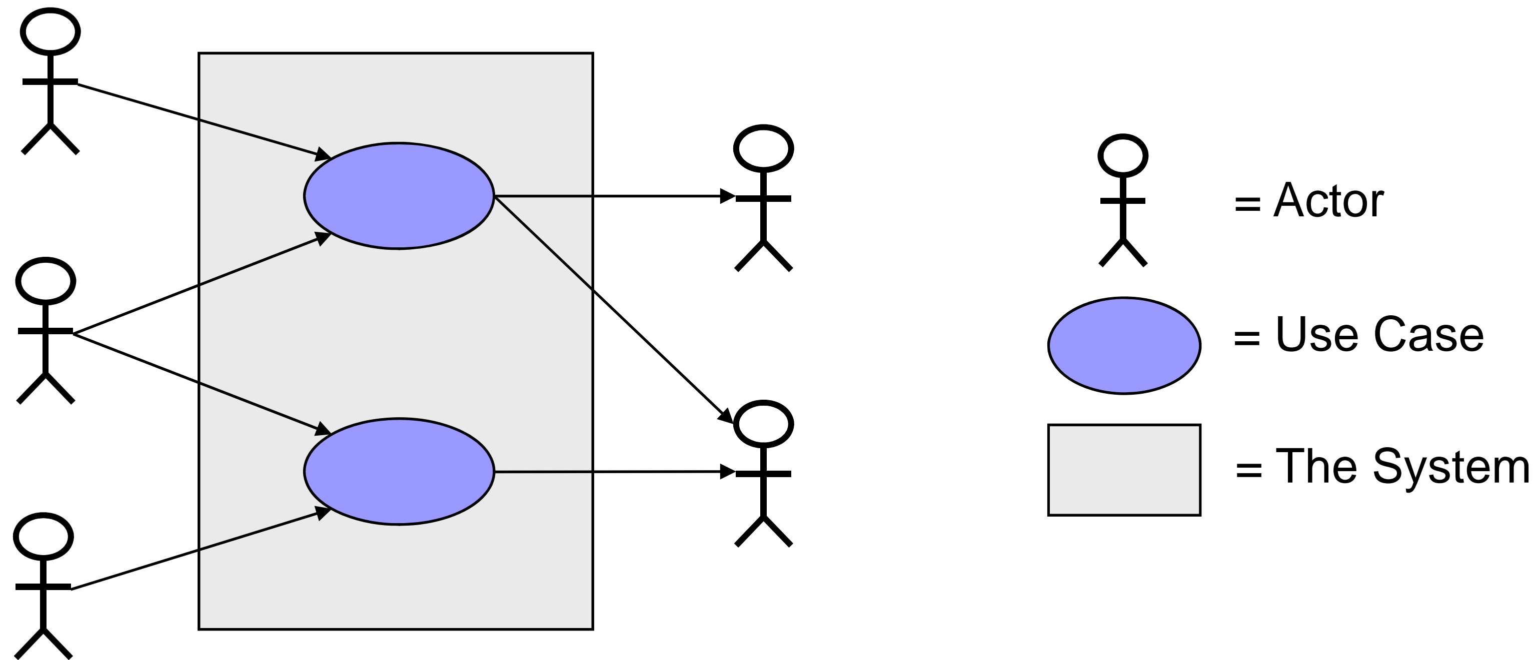
© Garth Gilmour 2008

Introducing the Use Case Diagram

- The Use Case Diagram organizes use cases
  - It shows Actors and Use Cases
- Actors represent things external to the system
  - An actor either initiates a Use Case or is contacted during its execution to provide or store information
  - Actors can be human users or other systems
- A Use Case represents a logical group of tasks
  - Which form a dialog between an actor and the system
    - In order to achieve some well defined result
  - Use cases can include many functions
    - For example ‘Maintain Account’ and ‘Configure System’

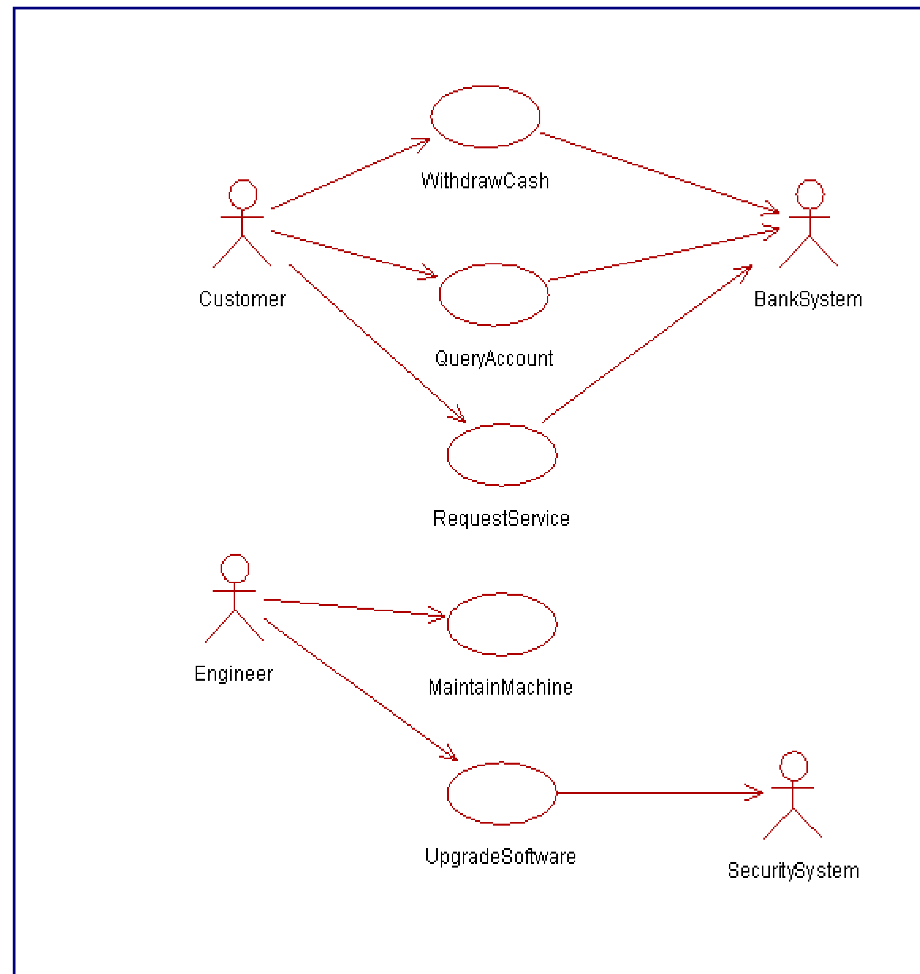
© Garth Gilmour 2008

Introducing the Use Case Diagram



© Garth Gilmour 2008

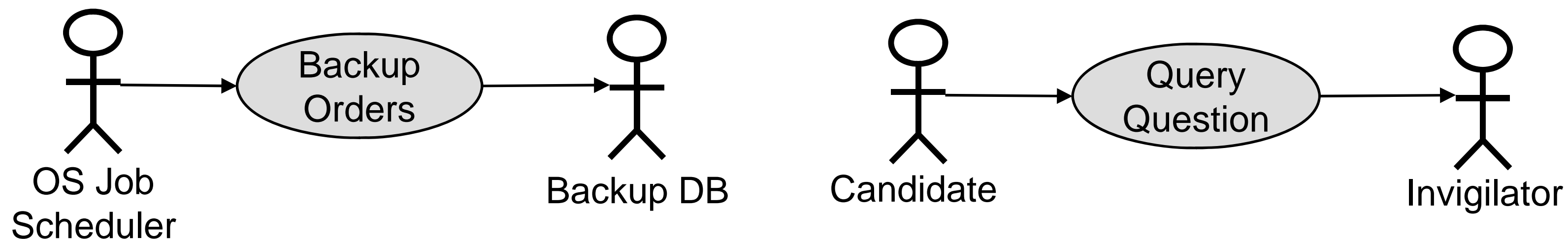
A Use Case Diagram For An ATM



© Garth Gilmour 2008

Use Case Diagrams

- Normally a Use Case:
  - Is initiated by a human actor (Customer, Employee etc…)
  - Makes use of external system actors (Web Services etc…)
- Other combinations are possible
  - The scenario could be started by an external system
  - Human actors could be consulted as the scenario progresses

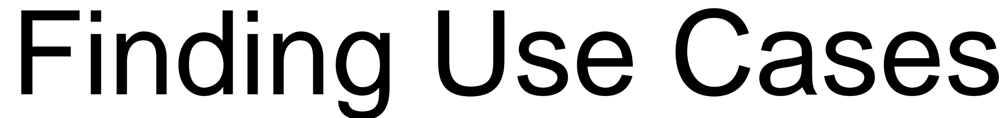


© Garth Gilmour 2008

Use Case Diagrams Are Important

- The Use Case Diagram is at the heart of UML
  - Everything that follows is based around it
- All stakeholders can see:
  - The Use Cases that make up the system
  - Which Use Cases are initiated by which actors
  - Which actors are referenced during a Use Case
- The Use Case model is the basis for:
  - The customers understanding of the system
  - Test plans produced by the QA dept
  - Analysis and design by developers
  - Project planning by managers

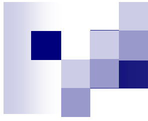
© Garth Gilmour 2008



Finding Use Cases

- Discovering Use Cases is a crucial ability
  - There is no step by step procedure that can be taught
- The key is that stories are related features
  - E.g. ‘Update Shopping Cart’ rather than ‘Add to Cart’, ‘Remove from Cart’ and ‘Change Item in Cart’
  - This is the simplest way of linking features
    - By create/read/update/delete operations (CRUD)
    - Note that creating the cart might be a separate story
- Unworkable Use Cases tend to stand out
  - Confusing features with stories produce many simple Use Cases
  - Unworkable Use Cases are too large and confusing to read

© Garth Gilmour 2008



Use Case Diagrams and Reports

The Use Case Diagram is NOT all the UML uses to capture requirements

It only illustrates what the stories are and how they are related to actors

© Garth Gilmour 2008

The Use Case Report

- Every Use Case is detailed in a Use Case Report
  - Five to fifteen pages of structured documentation
- Use Case Reports contain scenarios
  - Flows of events that can occur as the system is used
  - Scenarios are organised into:
    - The main flow of events
      - The ‘happy path’ that users normally take
    - Alternate flows of events
      - Standard paths that users will occasionally take
    - Exceptional flows
      - What happens when the system breaks

© Garth Gilmour 2008

Preconditions:

The ATM is running and has cash available

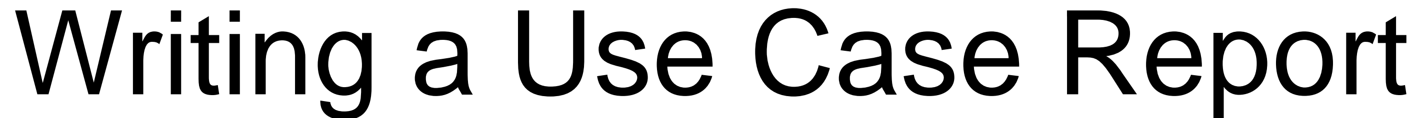
Postconditions:

The ATM has dispensed cash, the users balance has been reduced

Flows Of Events:

- Basic Flow (The Happy Path)
  1) User enters their card and pin number
  2) System verifies the users identity and displays withdrawal amounts
  3) User selects an amount to withdraw
  4) System verifies amount and returns card
  5) User removes card
  6) System dispenses cash
  7) User removes cash
- Alternative Flows
  - User specifies the amount to withdraw
  - User has to re-enter password
- Exceptional flows
  - Card jams in reader
  - Bank System unavailable

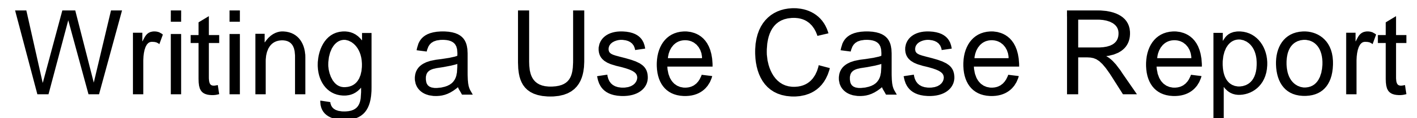
© Garth Gilmour 2008



Writing a Use Case Report

- The system is always seen as a 'black box'
  - The users presents some information to the system
  - The system then returns some result to the user
  - This dialog continues until a goal is achieved
- Maintaining this dialog is critical
  - The user never makes choices without the system acting
  - The system never returns data without the user acting on it
  - This is good discipline and makes the Use Case readable
- Alternative flows are treated differently
  - Simple ones are often written as a single paragraph
  - Complex ones should be presented as a dialog

© Garth Gilmour 2008



Writing a Use Case Report

- As reports become more detailed the ‘black box’ rule may be violated in some cases
  - This is were the system will have to perform a certain sequence of actions regardless of the eventual design
  - E.g. ‘the amount is debited from the customers account in DB1 and credited to the vendors in DB2 minus the transaction fee’
- The general rules relating to UML diagrams apply
  - Many versions of the same report can be viewed as correct, depending on which stage of the process you are at
  - Some projects have two versions of the requirements documentation, one for customers and one for developers

© Garth Gilmour 2008

Advanced Techniques in Reporting

- Ideally all Use Cases would be independent
  - Each could be understood in isolation from the others
- In practise Use Cases are often interdependent
  - Modelled via the Include, Extend and Generalize relationships
- Use Cases can Include one another
  - We pause one Use Case in order to execute a flow from another:
    - ‘If this is a new customer then include {Create Account}’
    - ‘If the amended order no longer qualifies for a discount then the {Issue Additional Fee} Use Case is included’
  - We use an Include when:
    - Common behaviour is found inside multiple Use Cases
    - Behaviour occurs separately and within other Use Cases

© Garth Gilmour 2008

Advanced Techniques in Reporting

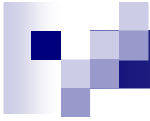
- Use Cases can Extend one another
  - When a condition may cause optional behaviour to be executed
  - The effect is similar to an alternative flow except that the behaviour is complex enough to merit its own use case:
    - ‘Create Joint Signatory Account’ might extend ‘Create Account’
    - ‘Generate Collated Report’ might extend ‘Generate Report’
  - Once the optional behaviour is complete we resume the original Use Case at the extension point
- We use an Extend when:
  - We need to empathise that behaviour is optional or complex
  - By placing behaviour in a separate Use Case it can be reused

© Garth Gilmour 2008

Advanced Techniques in Reporting

- Use Cases can use Generalization
  - The technique is very rarely seen in practise
- Generalization differs from Extends or Include
  - In the previous relationships we branch out and back once
  - In a Generalization we go back and forward any number of times
- Generalization is used because of common behaviour
  - Unlike an Include the behaviour is not a continuous sequence of steps but rather steps from multiple places in the flow
- The base Use Case may well be abstract
  - This means it will not be executed itself
  - Rather it is simply a repository for behaviours

© Garth Gilmour 2008



Login Use Case

Basic Flow:

1. User provides username
   …
2. System verifies username
   …
3. User provides password
   …
4. System verifies password
   …
5. User acknowledges login
   …

Admin Login Use Case
(child of Login)

Basic Flow:

1. As step 1 in parent
2. As step 2 in parent
3. User chooses to work as admin
4. System requests admin password
5. As step 3 in parent
6. As step 4 in parent
7. Use acknowledges now working as admin
8. System logs admin account active
9. As step 5 in parent

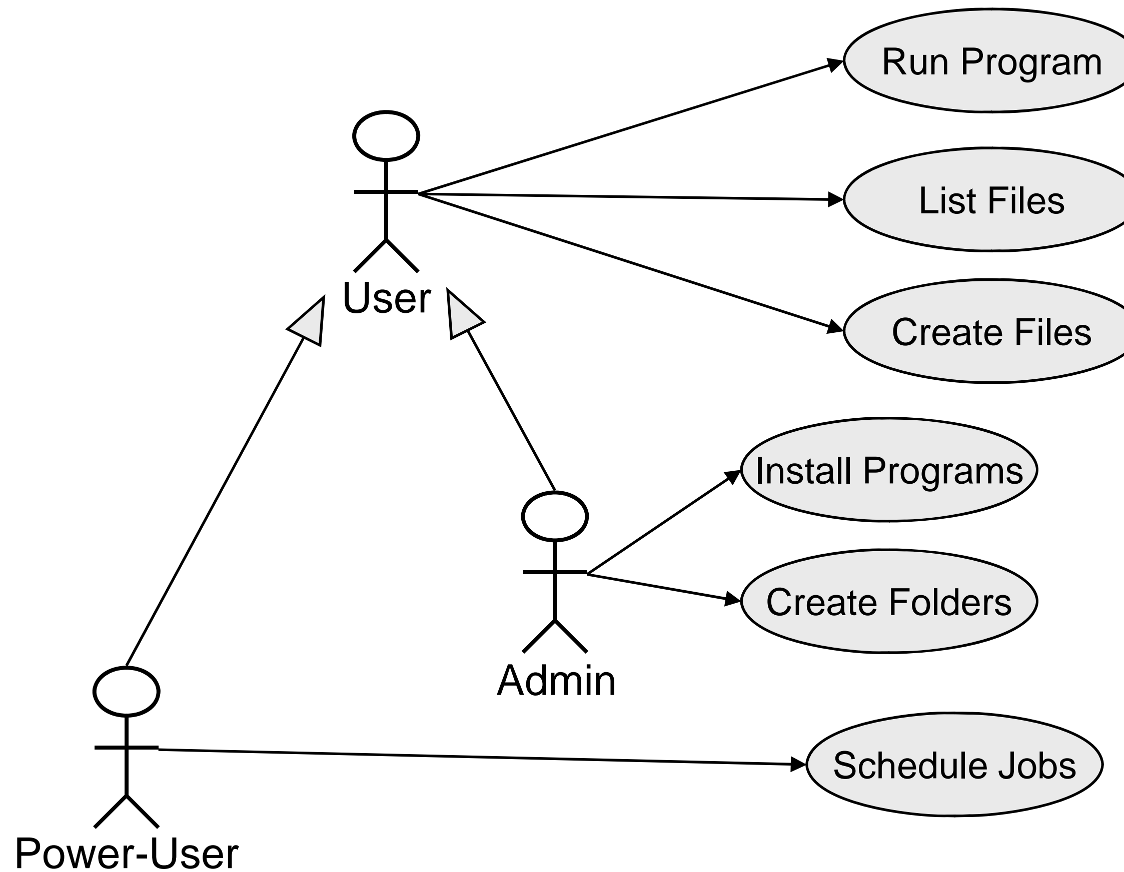
© Garth Gilmour 2008

Advanced Techniques in Reporting

- Generalization can also occur between actors
  - This is much less controversial and easier to understand
- Actors are generalized when one initiates the same Use Cases as another plus extra ones specific to their role
  - E.g. an ‘admin’ and a ‘power-user’ both initiate the Use Cases of a normal ‘user’ plus extra ones (such as ‘Install Programs’)
- Using actor generalization keeps the diagram simpler
  - Otherwise it would be cluttered with arrows
- Again it is possible for the base to be abstract
  - However users typically find this concept hard to visualize

© Garth Gilmour 2008

Generalization In Actors



© Garth Gilmour 2008

Supplementing Use Case Reports

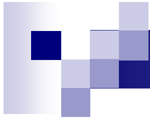
- Use Case Reports should not contain
  - Specifications of the controls or layout of the GUI
  - Precise descriptions of algorithms and calculations
  - Lists of items stored or withdrawn from the database
- These elements would make the story unreadable
  - Remember Use Cases are about ‘chunking’ requirements
- Other artefacts can supplement the Use Case
  - The recommended UML tool is the Activity diagram
- Activity diagrams specify an algorithm in detail
  - They are the UML version of a flowchart

© Garth Gilmour 2008

Supplementing Use Case Reports

- Activity Diagrams are used to show:
  - Different activities that are carried out
  - Conditions that affect which activity comes next
- Diamonds represent decision points
  - Conditions that trigger the decision are in brackets
- Some activities can be carried out in parallel
  - One bar is used to indicate when concurrency begins
  - Another shows when all activities must complete
    - Before execution continues
  - This does not constrain the implementation
    - The design does not have to be multi-threaded

© Garth Gilmour 2008



Read Card No & PIN

Validate Customer

[invalid]

[valid]

Read Amount

Count Cash

Validate Amount

Display Error

Return Card

Return Card

Vend Money

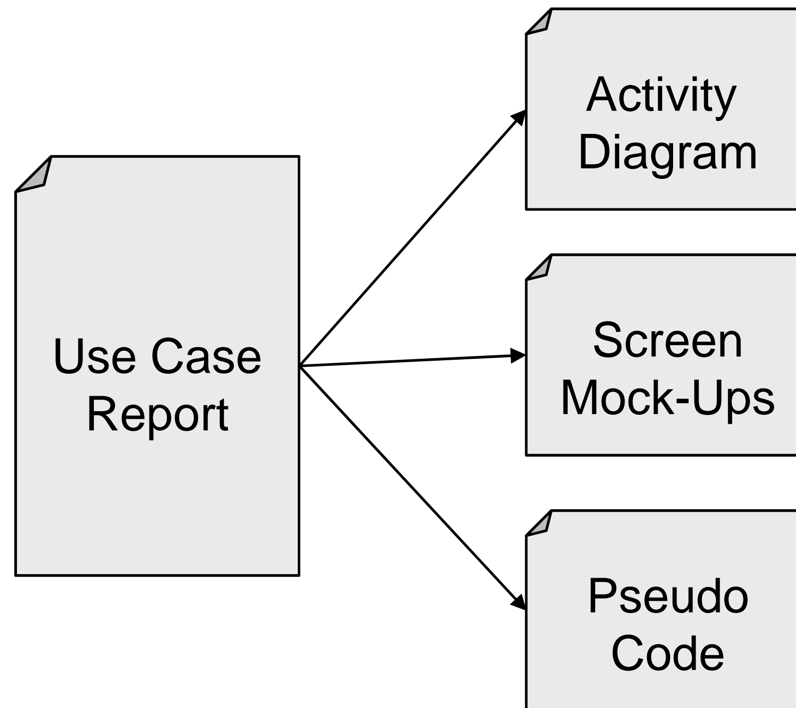
© Garth Gilmour 2008

Supplementing Use Case Reports

- Activity Diagrams are not the only artefact that can be used to supplement Use Case Reports
  - Just because something isn't a formal part of the UML doesn’t mean you cant or shouldn’t use it
- Other tools that can be used include:
  - Pseudocode
  - Screenshots
  - Other notations
    - DFD’s, Concept Models, GUI Flow Diagrams
- Each of these has benefits and drawbacks
  - Which combination you use depends on the client

© Garth Gilmour 2008

Supplementing Use Case Reports



© Garth Gilmour 2008

Pros and Cons of Using Prototypes

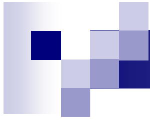
- Prototyping is a good idea
  - By working with mock-ups and screen layouts the user can actively think about the system
- But prototypes must be deliberately ‘rough’
  - Stakeholders tend to equate a polished GUI with a completed application – leading to false hope
  - If Stakeholders fixate on the GUI they will generate a stream of minor but time consuming change requests
- Tools now exist to create imperfect GUI’s
  - E.g. the Napkin plug-in for the Java language
  - This makes a generated GUI appear hand drawn

© Garth Gilmour 2008

Non Functional Requirements

- Use Cases only capture functional requirements
  - The behaviour that the system accomplishes
- Non functional requirements are kept elsewhere
  - The UP places these in a ‘Supplementary Specification’
  - These requirements are based around (F)URPS
    - Usability – how easy is the system to use?
    - Reliability – how often must the system be restarted?
    - Performance – how quickly does the system perform?
    - Supportability – how much upkeep does the system require?
- These requirements are critical in web based systems
  - Where levels of usage and access times are unpredictable

© Garth Gilmour 2008



- ...
- ...
- ...
- ...
- ...
- ...

Non-Functional Requirements

Iterations Plan

- Unit
- Subsystem
- System
- Acceptance

Testing Plan

Use Cases
(Diagram & Reports)

1. ...
2. ...
3. ...
4. ...
5. ...
6. ...

Functional Requirements

Analysis / Design

© Garth Gilmour 2008