Requirements engineering

Paul Jackson

School of Informatics
University of Edinburgh

What are requirements?

The requirements of a system are the description of

- the services that a system should provide
- the constraints on its operation

Requirements reflect needs of customer

Requirements try to avoid design, expressing what is desired, not how what is desired should be realised

Requirements described at differing levels of detail, depending on the intended readers:

User requirements vs. System requirements

Whose requirements?

Requirements are usually relevant to multiple stakeholders:

- End users
- Customers paying for software
- Government regulators
- System architects
- Software developers
- Software testers
- ...

Input from stakeholders needed when developing requirements. But

- They may find it difficult to articulate their requirements.
- They may place different priorities on requirements

Requirements documents need to be clear about their target audience(s).

Requirements classification

Traditional to distinguish *functional* from *non-functional* requirements.

Functional requirements (*services*): What the system should do.

Non-functional requirements (*constraints*): How fast it should do it; how seldom it should fail; what standards it should conform to; how easy it is to use; etc.

Non-functional requirements may be more important than functional requirements!

- Can be workarounds for functional requirements
- User experience often shaped by non-functional.

Distinction not always clear-cut

- Security might initially be a non-functional requirement, but, when requirements refined, it might result in addition of authorisation functionality

Requirements capture processes

Process activities include

- Requirements elicitation
- Requirements analysis
- Requirements specification
- Requirements validation

Activities often overlapping, not in strict sequence, and iterated.

Several approaches possible for each activity. Choice is very dependent on nature of software developed and overall software development process.

Faulty requirement capture can have huge knock-on consequences later in development process. One motivation for incremental nature of Agile processes.

Requirements elicitation sources

- Goals: high-level objectives of software
- Domain Knowledge: Essential for understanding requirements
- Stakeholders
- Business rules: E.g. Uni regulations for course registration
- Operational Environment: E.g. concerning timing and performance
- Organisational Environment: How does software fit with existing practices?

(From SWEBOK V3, Ch1)

Requirements elicitation techniques

- Interviews
- Scenarios
- Prototypes
- Facilitated meetings
- Observation

Requirements elicitation: interviews

Traditional method: ask them what they want, or currently do

Can be challenging:

- Jargon confusing
- Interviewees omit information obvious to them

Good techniques include

- Being open minded: requirements may differ from those pre-conceived,
- Using leading questions, e.g. from first-cut proposal for requirements

Requirements elicitation: scenarios

Scenarios are typical possible interactions with the system

- Provide a context' or framework for questions.
- Allow "what if" or "how would you do this" questions.
- Examples Include use cases and user stories

Requirements elicitation: prototypes

Can include

- screen mock-ups
- storyboards
- early versions of systems

Like scenarios, but more “real”. High quality feedback. Often help to resolve ambiguities.

Requirements elicitation: facilitated meetings

Get discussion going with multiple stakeholders in a structured manner, to refine requirements

Helps with:

- Requirements that are not about individual activities
- Surfacing / resolving conflicts

Needs a trained facilitator.

Requirements elicitation: observation

Immersive method. Expensive.

Helps with:

- Surfacing complex / subtle tasks and processes
- Finding the nuances that people never tell you

Requirements analysis

Requirements elicitation often produces a set of requirements that

- *contradicts* itself (even the same stakeholder may request contradictory things)
- contains *conflicts* (e.g., one stakeholder wants one-click access to data, another requires password protection)
- is *too large* for all requirements to be implemented.

Requirements analysis is the process of getting to a single consistent set of requirements, classified and prioritised usefully, that will actually be implemented.

Requirements specification

Requirements almost always need to be recorded, maybe using:

- very informal means e.g. handwritten cards, in agile development
- a document written in careful structured English, e.g. `3.1.4.4 The system shall...`
- use case models with supporting text
- a formal specification in a mathematically-based language.

Probably reviewed, may be contractual.

Requirements validation

Checks include:

- Validity checks: do requirements reflect real needs? Are they up to date?
- Consistency checks
- Completeness checks
- Realism checks: can requirements be met using time and money budgets?
- Verifiability: is it possible to test that each requirement is met?
  - Applies to both functional and non-functional requirements. Non functional requirements must be measurable.

User Stories

Used in "agile" (low ceremony, lightweight) development processes, e.g. Extreme Programming (XP), to document requirements

User stories are brief, written by the customer on an index card. E.g.

`10. User A leaves the office for a short time (vacation etc.) and assigns his access privileges to user B, so B can take care of A's tasks while A is gone.` *Source: user; Priority: M*

Pros and cons of user stories

Pros:

- can really be owned by the customer: so more likely to be correct
- quick to write and change
- small, so relatively easy to estimate and prioritise

Cons:

- May be incomplete, inconsistent
- Only work in conjunction with good access to the customer
- Not suitable to form the basis of a contract

Now we go on to medium-ceremony approaches.

Reading

Required: *SWEBOK V3*, Chapter 1, Software Requirements.

Suggested: *Sommerville*, Part 1 chapter on Requirements Engineering.

Use cases

Paul Jackson

School of Informatics
University of Edinburgh

The Unified Modeling Language

UML is a graphical language for recording aspects of the requirements and design of software systems.

It provides many diagram types; all the diagrams of a system together form a UML model, which must be consistent (in a weak sense...).

Mostly tailored to an OO world-view

Often used just for documentation, but in model-driven development, a UML model may be used e.g. to generate and update code and database schemas automatically.

Many tools available to support UML

UML: Use cases

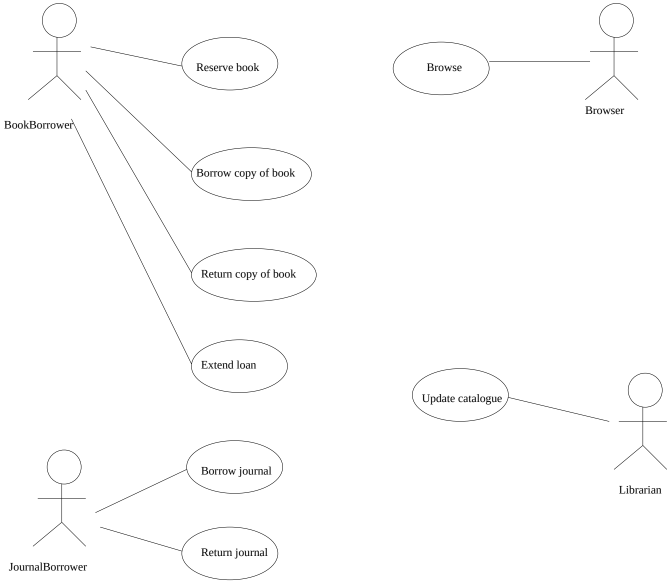
Document the functional requirements of the system *from the users' points of view.*

They help with three of the most difficult aspects of development:

- capturing requirements
- planning iterations of development which are good for users
- meaningful system testing

A set of use cases is *summarised* in a UML use case diagram.

A simple use case diagram



UML use cases: Actors

An **actor** – shown as a stick figure – can be:

- a human user of the system *in a particular rôle*
- an external system, which *in some rôle* interacts with the system.

Not a user: a particular *kind* of user. E.g., Bank Customer.

The same human user or external system may interact with the system in more than one rôle: he/she/it will be (partly) represented by more than one actor. (e.g., a bank teller may happen also to be a customer of the bank).

What is a use case?

A task involving the system which has value for an actor, e.g. `Borrow copy of book`.

Shown on diagram as named oval.

Each use case

- has a discrete goal the actor wishes to achieve
- includes a description of the a sequence of messages exchanged between the system and actors, and actions performed by the system, in order to achieve the goal.

Use cases primarily capture functional requirements, but sometimes non-functional requirements are attached to a use case.

Other times, non-functional requirements apply to subsets or all of use-cases.

Paths in a use case

Usually a use case describes a core of sequence of steps necessary to achieve its goal.

However might be alternatives, including some handling when all does not go to plan and the goal is not achieved.

Each path through use case called a *use-case instance* or *scenario*

- One talks about the main success scenario and alternate success or failure scenarios.

All scenarios in a use-case tied together by common user goal.

Warning: Sometimes *scenario* and *use-case* are synonyms

Example of use-case paths

Goal: Buy a Product

Main Success Scenario

1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills shipping info.
4. Systems presents full pricing information.
5. Customer fills in credit card info.
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirmation email to customer

Example of use-case paths (cont)

Extensions - variations on main success scenario

- 3a . If customer is regular customer
  - .1 . System displays current shipping and billing information
  - .2 . Customer may accept or override these defaults, returns to MSS at step 4, but skips step 5.
- 6a . System fails to authorize credit card purchase
  - .1. Customer may reenter credit card information or may cancel

Example fields in use case template

- Goal What the primary actor wishes to achieve
- Summary A one or two sentence description of the use case.
- Primary actor
- Secondary actors
- Trigger The event that leads to this use case being performed.
- Pre-conditions/Assumptions What can be assumed to be true when the use case starts
- Guarantees What the use case ensures when it succeeds
- Main Success Scenario
- Alternative scenarios

Use cases: connections and scope

A use case:

- can have different levels of detail, e.g. depending on where it is used in development process
- may refer to other use cases to provide further information on particular steps
- may be associated with other UML models (e.g. sequence and state diagrams) which show how it is realised;
- May describe different scopes: e.g. a system of systems, a single system or a single component of a system

Requirements capture organised by use cases

Use cases can help with requirements capture by providing a structured way to go about it:

1. identify the actors
2. for each actor, find out
    - what they need from the system
    - any other interactions they expect to have with the system
    - which use cases have what priority for them

Good for both requirements specification and iterated requirements elicitation.

There may be aspects of system behaviour that don't easily show up as use cases for actors.

Analysis vs design

Some actors are part of the requirements: usually the ones who derive benefit from a use case.

Others are part of the (business process) design: the ones who interact with the computer system to provide the benefit.

For example, consider a `FindBook` use case of a library, in which the user enters details of a book and wants to end up with a copy of it. Maybe the system will give the user directions to where the book is on the shelf. Maybe it will alert a librarian to go and fetch it. In the latter case, should the librarian be shown as actor? In some sense, the choice is a design decision.

Using use cases in development

Use cases are a good source of system tests: requirements documented as desired interactions, which translate easily into tests.

Earlier, they can help to validate a design. You can walk through how a design realises a use case, checking that the set of classes provides the needed functionality and that the interactions are as expected.

Politics

Through emphasising the value of tasks to actors, use cases help us understand *what is important to whom*.

To avoid a project being cancelled, should make sure system delivers added value:

- soon
- to all the people who might scupper it
- in every iteration

Of course, use cases might identify a project is not worth even starting because benefits to key actors are minimal

Possible problems with use cases

- Interactions spelled out may be too detailed, may needlessly constrain design
- May specify secondary actors that are not essential for fulfilling goal of primary actor
  - Does borrowing a book have to involve a librarian?
- Focus on operational nature of system may result in less attention to software architecture and static object structure
  - Refactoring during design may help
- May miss requirements not naturally associated with actors

Reading

Required: *Tokeneer ID Station System Requirements Specification*. See Section 5 for use-case-like scenarios. More generally, browse the document to get a feel of what a real Requirements Specification looks like.

Suggested: *Sommerville*. Use Cases discussed are both in Requirements and System Modeling chapters. Look up Use Cases in index to find the relevant sections.

Suggested: *Stevens*, Chapter 7.

Software design and modelling

Paul Jackson

School of Informatics
University of Edinburgh

What is design?

Design is the process of deciding how software will meet requirements.

Usually excludes detailed coding level.

What is good design?

(Some) criteria for a good design

- It can meet the known requirements
  (functional and non-functional)
- It is maintainable:
  i.e. it can be adapted to meet future requirements
- It is straightforward to explain to implementors
- It makes appropriate use of existing technology,
  e.g. reusable components

Notice the human angle in most of these points, and the situation-dependency, e.g.

- whether an OO design or a functional design is best depends (partly) on whether it is to be implemented by OO programmers or functional programmers;
- different design choices will make different future changes easy – a good design makes the most likely ones easiest.

Levels of design

Design occurs at different levels, e.g. someone must decide:

- how is your system split up into subsystems?
  (high-level, or architectural, design)
- what are the classes in each subsystem?
  (low-level, or detailed, design)

At each level, decisions needed on

- what are the responsibilities of each component?
- what are the interfaces?
- what messages are exchanged, in what order?

What is architecture?

Many things to many people.

$\boxed{\text{The way that components work together}}$

More precisely, an architectural decision is a decision which affects how components work together.

Includes decisions about the high level structure of the system – what you probably first think of as “architecture”.

Pervasive, hence hard to change. Indeed an alternative definition is “what stays the same” as the system develops, and between related systems.

Classic structural view

Architecture specifies:

- what are the components?
  Looked at another way, where shall we put the encapsulation barriers? Which decisions do we want to hide inside components, so that we can change them without affecting the rest of the system?
- what are the connectors?
  Looked at another way, how and what do the components really need to communicate? E.g., what should be in the interfaces, or what protocol should be used?

The component and connector view of architecture is due to Mary Shaw and David Garlan – spawned specialist architectural description languages, and influenced UML2.0.

More examples of architectural decisions

- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- is there an appropriate software framework that can be used?
- what conventions do components have about error handling?

Clean architecture helps get reuse of components.

Detailed design

Happens inside a subsystem or component.

E.g., maybe the system architecture has been settled by a small team, written down, and reviewed. Now you are in charge of the detailed design of one subsystem. You know you have to write in Java, you know what external interfaces you have to work to and what you have to provide. Your job is to choose classes and their behaviour that will do that.

Idea: even if you're part of a huge project, your task is now no more difficult than if you were designing a small system.

(But: your interfaces are artificial, and this may make them harder to understand/negotiate/adhere to.)

Design Principles: initial example

Which of these two designs is better?

A)
```
public class AddressBook {
  private LinkedList<Address> theAddresses;
  public void add (Address a) {theAddresses.add(a);}
  // ... etc. ...
}
```

B)
```
public class AddressBook extends LinkedList<Address> {
  // no need to write an add method, we inherit it
}
```

C) Both are fine

D) I don't know

Design Principles: initial example (cont.)

A is preferred.

- an `AddressBook` is not conceptually a `LinkedList`, so it shouldn't extend it.
- If B chosen, it is much harder to change implementation, e.g. to a more efficient `HashMap` keyed on name.

Design principles 1

Cohesion is a measure of the strength of the relationship between pieces of functionality within a component.

High cohesion is desirable.

Benefits of high cohesion include increased understandability, maintainability and reliability.

Design principles 2

Coupling is a measure of the strength of the inter-connections between components.

Low or loose coupling is desirable.

Benefits of loose coupling include increased understandability and maintainability.

Design principles 3

- abstraction - procedural/functional, data
  the creation of a view of some entity that focuses on the information relevant to a particular purpose and ignores the remainder of the information
  e.g. the creation of a sorting procedure or a class for points
- separation of interface and implementation *specifying a public interface, known to the clients, separate from the details of how the component is realized.*
- encapsulation / information hiding
  grouping and packaging the elements and internal details of an abstraction and making those details inaccessible'

Design principles 4

- decomposition, modularisation
  dividing a large system into smaller components with distinct responsibilities and well-defined interfaces
- sufficiency, completeness
  all the important characteristics of an abstraction, and nothing more.

Modeling

Let's say: a model is any precise representation of some of the information needed to solve a problem using a computer.

E.g. a model in UML, the Unified Modeling Language. Use case diagrams are part of UML. A UML model

- is represented by a set of diagrams;
- but has a structured representation too (stored as XML);
- must obey the rules of the language;
- has a (fairly) precise meaning;
- can be used informally, e.g. for talking round a whiteboard;
- and, increasingly, for generating, and synchronising with, code, textual documentation etc.

Why design? Why model?

Fundamentally:

Design, so that you'll be able to build a system that has the properties you want.

Model, so that you can design, and communicate your design.

Both can be done in different styles...

Pros and cons of BDUF

Big Design Up Front

- often unavoidable in practice
- if done right, simplifies development and saves rework;
- but error prone
- and wasteful.

Alternative (often) is simple design plus refactoring.

XP maxims:

You ain't gonna need it

Do the simplest thing that could possibly work

Reading

Suggested: SWEBOK v3 Ch2 for an overview of the field of software design

Suggested: Sommerville 10th Ed, Ch 6 on Architectural Design

Suggested: *An Introduction to Software Architecture* tech report. David Garlan and Mary Shaw. 1994.

Suggested: *Software Architecture and Design* chapters of *Microsoft Application Architecture Guide, 2nd Edition.*

UML class diagrams

Paul Jackson

School of Informatics
University of Edinburgh

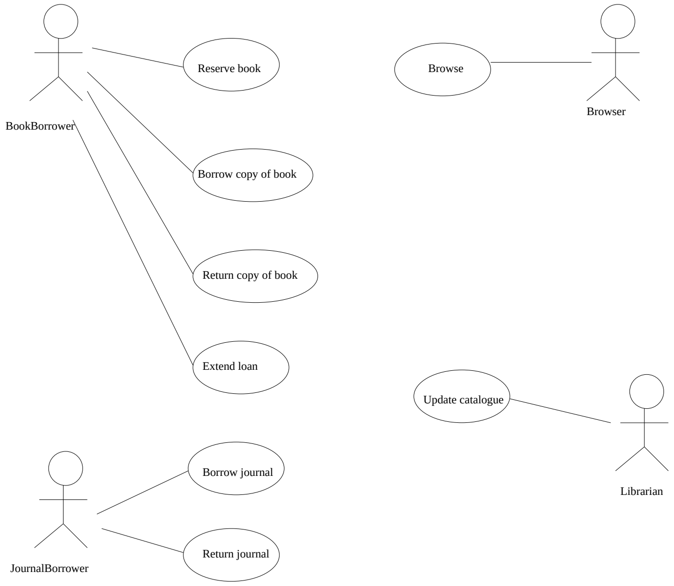
The Unified Modeling Language

UML is a graphical language for recording aspects of the requirements and design of software systems.

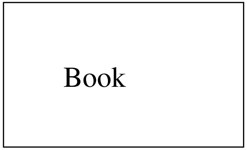
It provides many diagram types; all the diagrams of a system together form a UML model. Three important types of diagram:

1. *Use-case diagram*. Already seen in requirements lecture.
2. *Class diagram*. Today.
3. *Sequence diagram*. In the future.

Reminder: a simple use case diagram



A class

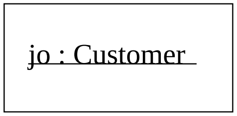


A class as design entity is an example of a **model element**: the rectangle and text form an example of a corresponding **presentation element**.

UML explicitly separates concerns of actual symbols used vs meaning.

Many other things can be model elements: use cases, actors, associations, generalisation, packages, methods,...

An object



This pattern generalises: always show an instance of a classifier using the same symbol as for the classifier, labelled instanceName : classifierName.

Classifiers and instances

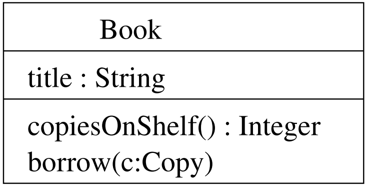
An aspect of the UML metamodel that it's helpful to understand up front.

An **instance** is to a **classifier** as an object is to a class: instance and classifier are more general terms.

In the metamodel, Class inherits from Classifier, Object inherits from Instance.

UML defines many different classifiers. E.g., UseCase and Actor are classifiers.

Showing attributes and operations



Syntax for signature of operations (argument and return types) adaptable for different programming languages. May be omitted

Compartments

We saw the standard:

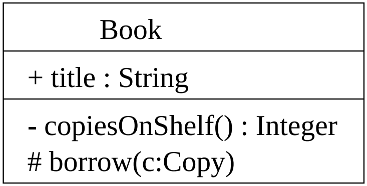
- a compartment for attributes
- a compartment for operations, below it

They can be suppressed in diagrams.

They are omitted if empty.

You can have extra compartments labelled for other purposes, e.g., responsibilities of the class...

Visibility

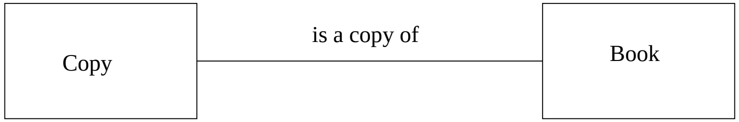


Can show whether an attribute or operation is

- public (visible from everywhere) with $+$
- private (visible only from inside objects of this class) with $-$

(Or protected (#), package ($\sim$) or other language dependent visibility.)

Association between classes

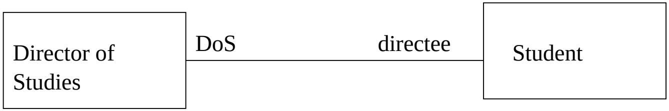


This generalises: association between classifiers is always shown using a plain line. (Recall the associations between actors and use cases!)

An instance of an association connects objects (e.g. Copy 3 of War and Peace with War and Peace).

An **object diagram** contains objects and links: occasionally useful.

Rolenames on associations



Can show the role that one object plays to the other.

Useful when documenting the class: e.g. a *class invariant* for DirectorOfStudies could refer to the associated Student objects as self.directee (a set, if there can be more than one).

Can use visibility notation $+$ $-$ etc on role names too.

Class invariants

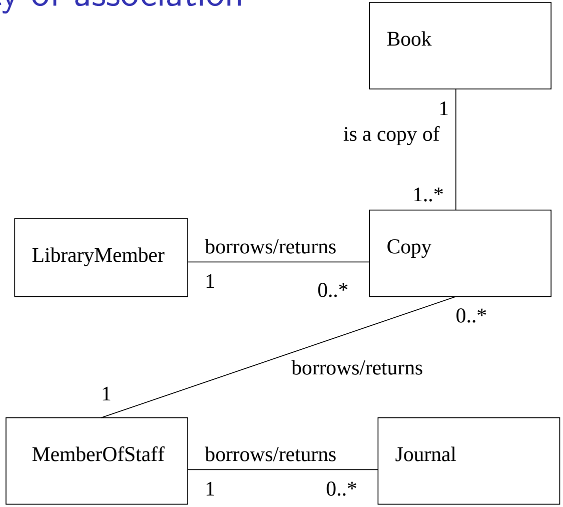
A class invariant is a statement which is supposed to be true of every object of the class, all the time - a “sanity check”.

Very useful to make these explicit. Can be included as comments on class diagrams, and in code.

May be formal, e.g. `x + y = z`, or informal, e.g. “the attribute `docstring` describes the action of the button in concise English”.

If formal, it can be useful to have class invariants automatically checked.

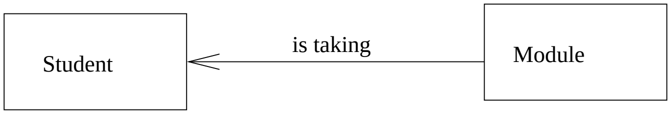
Multiplicity of association



Commas for alternatives, *two* dots for ranges, * for unknown number. E.g. each Copy is a copy of exactly one Book; there must be at least one Copy of every Book.

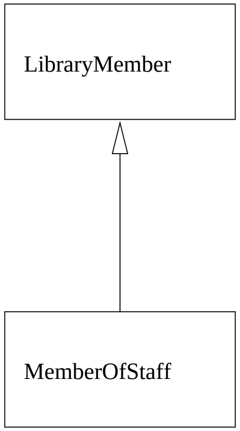
Navigability

Adding an arrow at the end of an association shows that some object of the class at one end can access some object of the class at the other end, e.g. to send a message.



Crucial to understanding the coupling of the system. NB direction of navigability has nothing to do with direction in which you read the association name.

Generalisation



This generalises: generalisation between classifiers is always shown using this arrow.

Usually, but not necessarily, corresponds to implementation with inheritance.

Abstract operations and classes

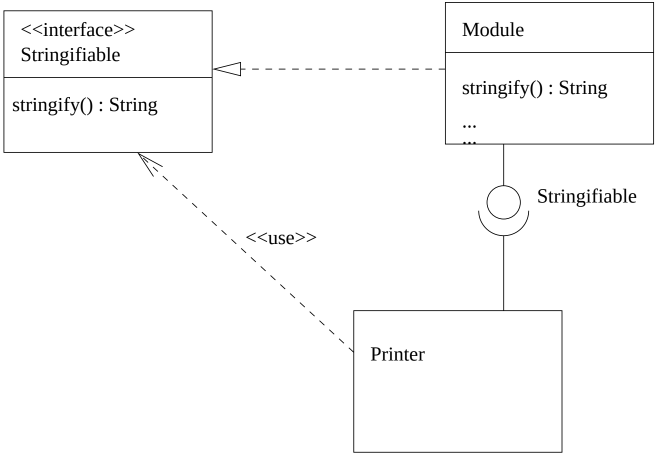
An operation of a class is abstract if the class provides no implementation for it: thus, it is only useful if a subclass provides the implementation.

A class which cannot be instantiated directly – for example, because it has at least one abstract operation – is also called abstract. Java...

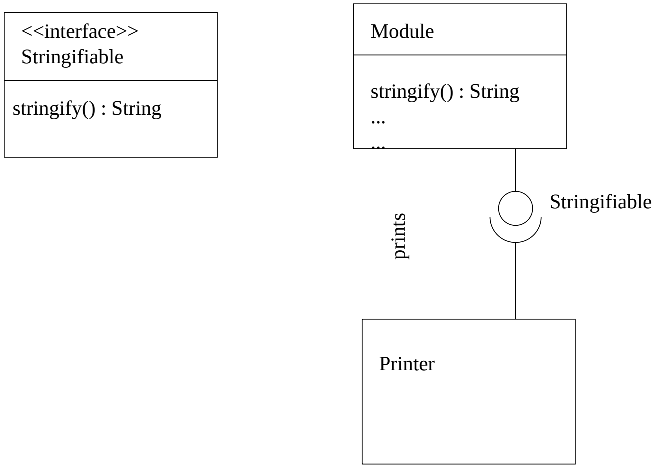
Can show *abstract* operation or class using italics for the name, and/or using the *property* {*abstract*}.

Interfaces

In UML an interface is just a collection of operations, that can be *realised* by a class.



Simpler diagram: WRITE ONCE



Many things other than classes can realise interfaces: can use the lollipop symbol on e.g. components, actors.

Identifying objects and classes

Simplest and best: look for noun phrases in the system description!

Then abandon things which are:

- redundant
- outside scope
- vague
- attributes
- operations and events
- implementation classes.

(May need to add some back later, especially implementation classes: point is to avoid incorporating premature design decisions into your conceptual level model.)

Similarly, can use verb phrases to identify operations and/or associations

Reading

Suggested: Stevens

- Ch 2: Object concepts
- Ch 3: The LIbrary case study
    - Includes basics of how to identify classes
- Ch 5: Essentials of class models
    - Includes use of CRC cards for class design
- Ch 6: For abstract classes and interfaces

Software component interactions and sequence diagrams

Paul Jackson

School of Informatics
University of Edinburgh

What do we need to know? Recap

Recall that this is an *overview* of software engineering, dipping into some aspects. We've discussed:

- how to analyse requirements and summarise them in a use case diagram;
- how to tell good design from bad;
- how to record basics of the static structure of our designed system in a class diagram;
- how to get started with choosing an appropriate static structure.

We have not discussed dynamic aspects of design: what operations should our classes have, and what should they do?

Dynamic aspects of design

Suppose that we have decided what classes should be in our system, provisionally. What next? Well, we have to meet the requirements...

In the end, we need to know what operations they have, and what each method should do.

Two ways of looking at this:

1. inter-object behaviour: who sends which messages to whom?
2. intra-object behaviour: what state changes does each object undergo as it receives messages, and how do they affect its behaviour?

Complementary: but in this course, we only consider 1. For 2, UML provides an enhanced variant on the FSMs you saw last year.

For more info, do SEOC next year, and/or read about State diagrams in the recommended texts.

Thinking about inter-object behaviour

There's no algorithm for constructing a good design. Create one that's good according to the design principles...

1. Your classes should, as far as possible, correspond to domain concepts.
2. The data encapsulated in the classes is usually pretty easy to define using the real world as a model.
3. Then look at the scenarios in the use cases, and work out where to put what operations to get them done.

Some of this is easy. Hard parts are usually when several objects have to collaborate and it isn't clear which should take overall responsibility.

Interaction diagrams

describe the *dynamic* interactions between objects in the system, i.e. the pattern of message-passing.

Two main uses:

- Showing how the system realises [part of] a use case
- Showing how an object reacts to some message

Particularly useful where the flow of control is complicated, since this can't be deduced from the class model, which is static.

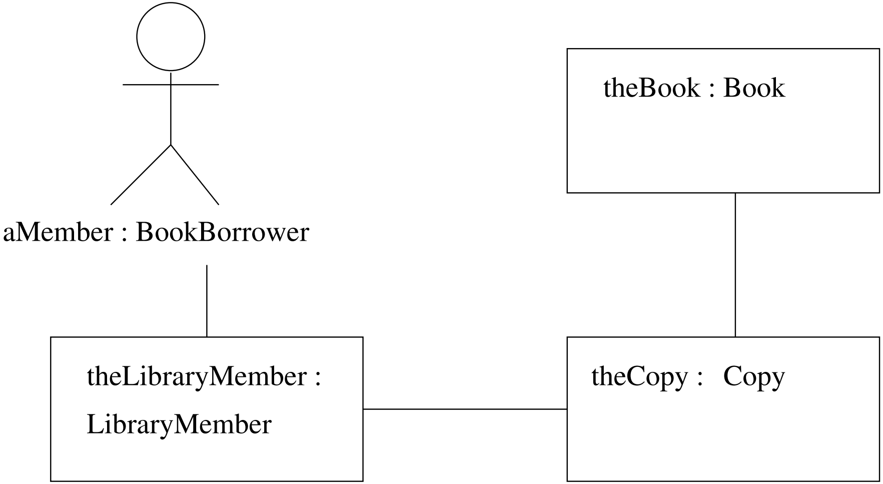
UML has two sorts, *sequence* and *communication* diagrams – the differences are subtle, and we'll only talk about sequence diagrams.

Developing an interaction diagram

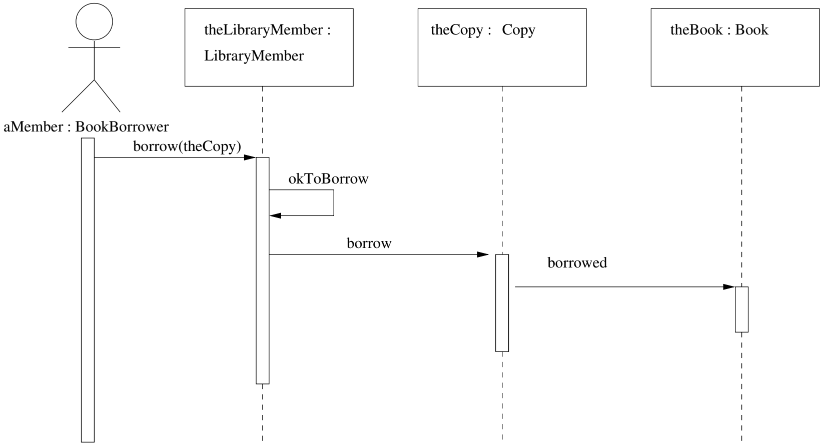
1. Decide exactly what behaviour to model.
2. Check that you know how the system provides the behaviour: are all the necessary classes and relationships in the class model?
3. Name the objects which are involved.
4. Identify the sequence of messages which the objects send to one another.
5. Record this in the syntax of a sequence diagram.

Simple :-)

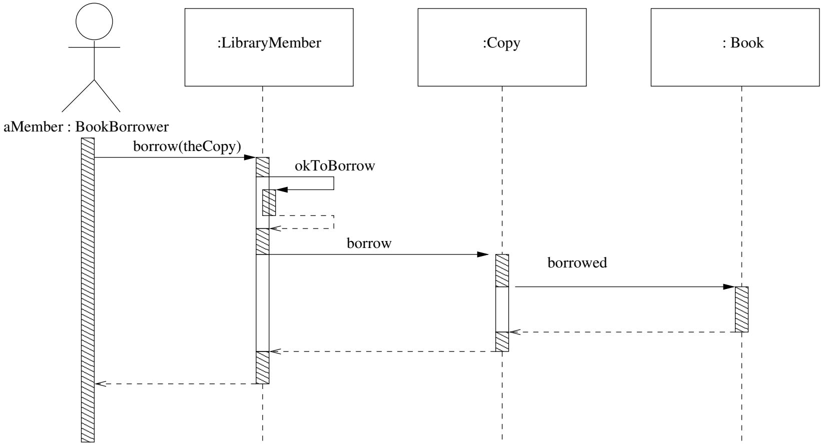
A collaboration



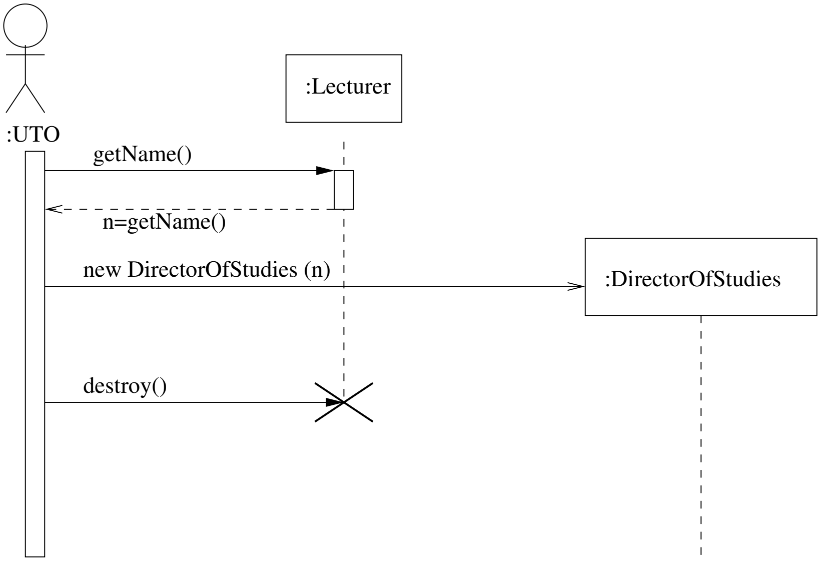
Sequence diagram



Showing more detail



Creation/deletion in sequence diagram



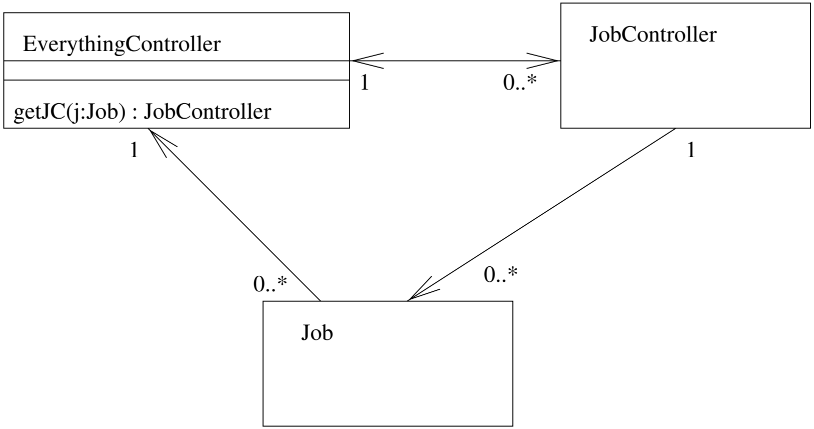
What is a good interaction pattern?

In designing an interaction, your first aim is obviously to design *some* collection of operations that can work together to achieve the aim.

Next, consider:

- conceptual coherence: does it make sense for this class to have that operation?
- maintainability: which aspects might change, and how hard will it be to change the interaction accordingly?
- performance: is all the work being done necessary?

Designing interactions



Problems?

Law of Demeter

A design principle to reduce longer-range coupling

In response to a message m, an object O should send messages *only* to the following objects:

1. O itself
2. objects which are sent as arguments to the message m
3. objects which O creates as part of its reaction to m
4. objects which are *directly* accessible from O, that is, using values of attributes of O.

More complex sequence diagrams

We've only discussed very simple sequence diagrams.

UML provides further notation for e.g.

- conditional behaviour
- iterative behaviour
- concurrent behaviour
- Including one diagram in another

For this course, you should be familiar with first two.

Reading

Required: At least one of

- The original paper on CRC cards, a technique for designing interactions: *A Laboratory for Object-Oriented Thinking*, by Kent Beck and Ward Cunningham.
- Stevens, Section 5.6 on CRC cards

You should know the idea of the CRC cards technique, including the basics of what each letter in "CRC" refers to.

CRC cards are covered in detail in SEOC.

Suggested: Stevens

- Ch 9: for basics of UML Sequence diagrams
- Ch 10: for conditional and iterative behaviour

Design Patterns

Paul Jackson

School of Informatics
University of Edinburgh

Design Patterns

“Reuse of good ideas”

A pattern is a named, well understood good solution to a common problem in context.

Experienced designers recognise variants on recurring problems and understand how to solve them. Without patterns, novices have to find solutions from first principles.

Patterns help novices to learn by example to behave more like experts.

Patterns: background and use

Idea comes from architecture (Christopher Alexander): e.g. **Window Place:** observe that people need comfortable places to sit, and like being near windows, so make a comfortable seating place at a window.

Similarly, there are many commonly arising technical problems in software design.

Pattern catalogues: for easy reference, and to let designers talk shorthand. Pattern *languages* are a bit more...

Patterns also used in: reengineering; project management; configuration management; etc.

A very simple recurring problem

We often want to be able to model tree-like structures of objects: an object may be

- a thing without interesting structure, a leaf of the tree, or
- itself composed of other objects
    - which in turn might be leaves or might be composed of other objects...

We want other parts of the program to be able to interact with a single class, rather than having to understand about the structure of the tree.

Composite is a design pattern which describes a well-understood way of doing this.

Example situation

A graphics application has primitive graphic elements like lines, text strings, circles etc.

A client of the application interacts with these objects in much the same way: for example, it might expect to be able to instruct such objects to draw themselves, move, change colour, etc.

Makes sense to have a

- Graphics interface or an abstract base class which describes the common features of graphics elements, and
- subclasses Text, Line, etc.

Want to be able to group elements together to form pictures, which can then be treated as a whole: for example, users expect to be able to move a composite picture just as they move primitive elements.

Familiar (?) way to do this kind of task in Haskell

```
data graphicsElement =
        Line
      | Text
      | Circle
      | Picture [graphicsElement]

draw Line = -- whatever
draw Text = -- whatever
draw Circle = -- whatever
draw (Picture l) = (let x = map draw l in ())
```

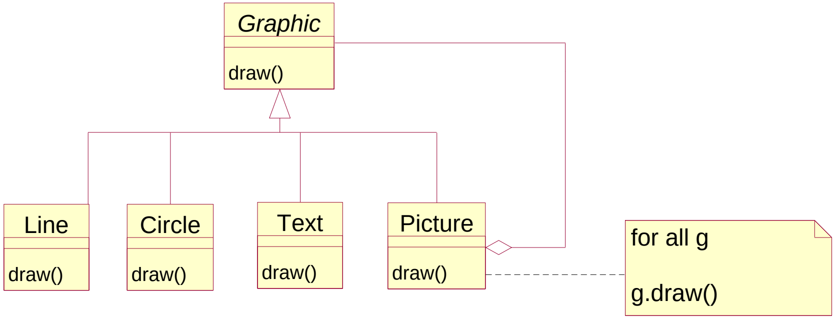
Drawbacks of the Haskell way

Clients must write recursive functions which pattern-match on the structure of the graphicsElement they have, *so all clients do in fact have to be aware of how elements of the datatype are built up.*

But this is just an example of how this mechanism does not support abstraction as well as we'd like: you can't (straightforwardly) wrap up the functions that should operate on a graphicsElement along with the datatype itself.

(Or can you? This example adapted from ML, and I don't speak Haskell...)

Composite pattern: best of both worlds



Benefits of Composite

- can automatically have trees of any depth: don't need to do anything special to let containers (Pictures) contain other containers
- clients can be kept simple: they only have to know about one class, and they don't have to recurse down the tree structure themselves
- it's easy to add new kinds of Graphics subclasses, including different kinds of pictures, because clients don't have to be altered

Drawbacks of Composite

- It's not easy to write clients which don't want to deal with composite pictures: the type system doesn't know the difference.
  (A Picture is no more different from a Line than a Circle is, from the point of view of the type checker.)

What could be done about this?

- Could use run-time checks on subtyping

Variations on Composite

- Might want to write some new method that walks over a whole Graphics tree. E.g. a *tree-map* method.
- To support it, need methods in Graphics like numChildren() and getChild(int i)
- Graphics then provides default implementations of these methods for the leaf subclasses.

Elements of a pattern

A pattern catalogue entry normally includes roughly:

- Name (e.g. Publisher-Subscriber)
- Aliases (e.g. Observer, Dependants)
- Context (in what circumstances can the problem arise?)
- Problem (why won't a naive approach work?)
- Solution (normally a mixture of text and models)
- Consequences (good and bad things about what happens if you use the pattern.)

Cautions on pattern use

Patterns are very useful *if you have the problem they're trying to solve*.

But they add complexity, and often e.g. performance penalties too. Exercise discretion.

You'll find the criticism that the GoF patterns in particular are "just" getting round the deficiencies of OOPLs. This is true, but misses the point.

Exercise: write a pattern language for Haskell!

Patterns: Reading

Required: Wikipedia entries on Observer and Template Method (or equivalent: what I want you to do is to know and understand those patterns to the extent that you could use them, describe them in UML class and sequence diagrams, and explain what they achieve and how).

Suggested: Read more on design patterns, e.g.

- Stevens: Ch18.2
- Sommerville: Look up *design patterns* in index
- `http://en.wikipedia.org/wiki/Design_Patterns`

Construction: version control and system building

Paul Jackson

School of Informatics
University of Edinburgh

The problem of systems changing

- Systems are constantly changing through development and use
  - Requirements change and systems evolve to match
  - bugs found and fixed
  - new hardware and software environments are targeted
- Multiple versions might have to be maintained at each point in time
- Easy to lose track of which changes realised in which version
- Help is needed in managing versions and the processes that produce them.

Software Configuration Management to the rescue

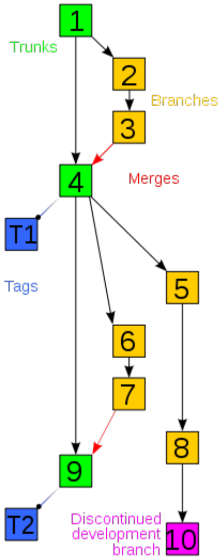
CM is all about providing such help.

Common CM activities:

- Version control
    - tracking multiple versions,
    - ensuring changes by multiple developers don't interfere
- System building
    - assembling program components, data and libraries,
    - compiling and linking to create executables
- Change management
    - tracking change requests,
    - estimating change difficulty and priority
    - scheduling changes
- Release management
    - preparing software for external release
    - tracking released versions

Focus on first two today

Version control



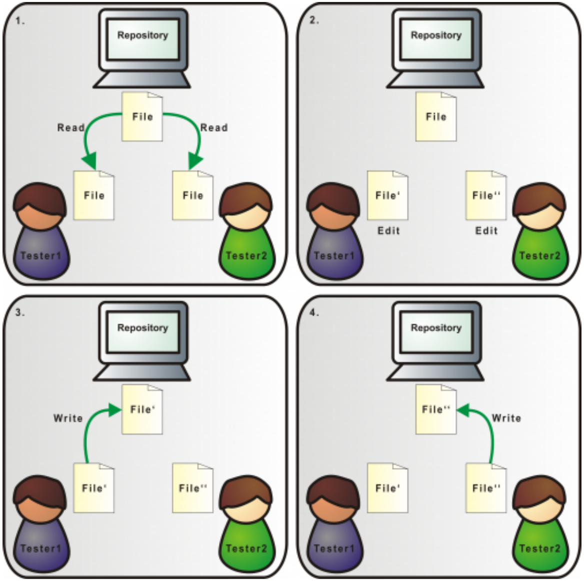
Version control

The core of configuration management.

The idea:

- keep copies of every version (every edit?) of files
- provide change logs
- somehow manage situation where several people want to edit the same file
- provide *diffs*/*deltas* between versions

Avoiding Race Conditions



RCS

RCS is an old, primitive VC system, much used on Unix.

Suitable for small projects, where only one person edits at a time.

Lock-Modify-Unlock model:

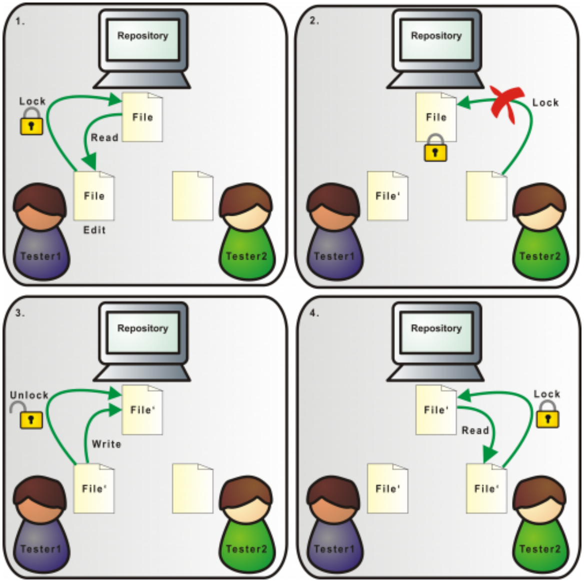
- Editor *checks-out* a file
    - Editor locks file
    - Others can check-out, but only for reading
- Editor makes changes
- Editor *checks-in* modified file
    - Lock is released
    - Changes now viewable by others
    - Others now can make their own changes

Keeps deltas between versions; can restore, compare, etc. Can manage multiple *branches* of development.

Remains a very useful tool for personal projects – and articles, lectures, essays, etc.

Further reading: *man rcsintro* on DICE.

Lock-Modify-Unlock



CVS and SVN

CVS is a much richer system, (originally) based on RCS.
Subversion (SVN) very similar.

Handles entire directory hierarchies or projects – keeps a single master *repository* for project.

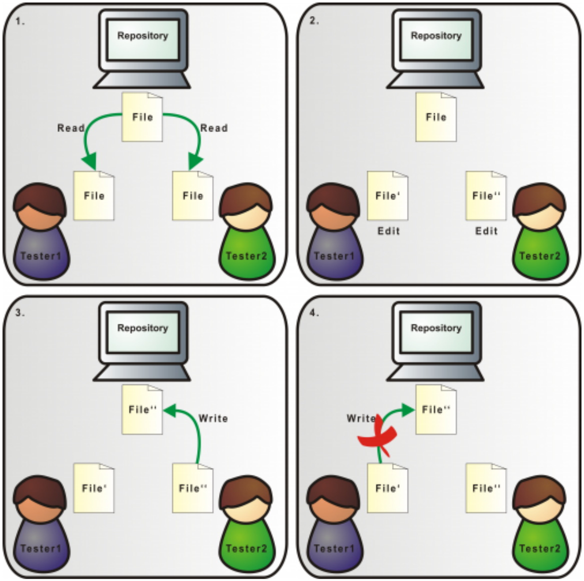
Is designed for use by multiple developers working simultaneously – Copy-Modify-Merge model replaces Lock-Modify-Unlock.

Pattern of use for Copy-Modify-Merge:

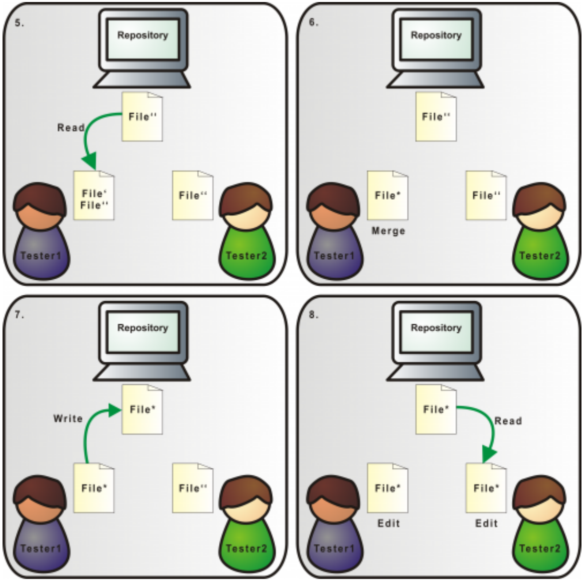
- *check out* entire project (or subdirectory) (not individual files).
- Edit files.
- Do *update* to get latest versions of everything from repository
  - system merges non-overlapping changes
  - user has to resolve overlapping changes - conflicts
- *check-in* version with merges and resolved conflicts

Central repository may be on local filesystem, or remote.

Copy-Modify-Merge



Copy-Modify-Merge



Distributed version control

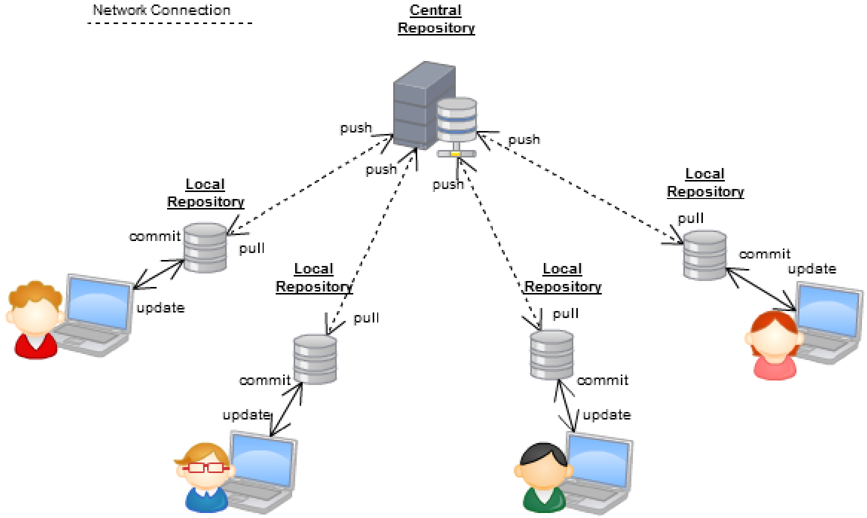
E.g. Git, Mercurial, Bazaar.

All the version control tools we've talked about so far use a single central repository: so, e.g., you cannot check changes in unless you can connect to its host, and have permission to check in.

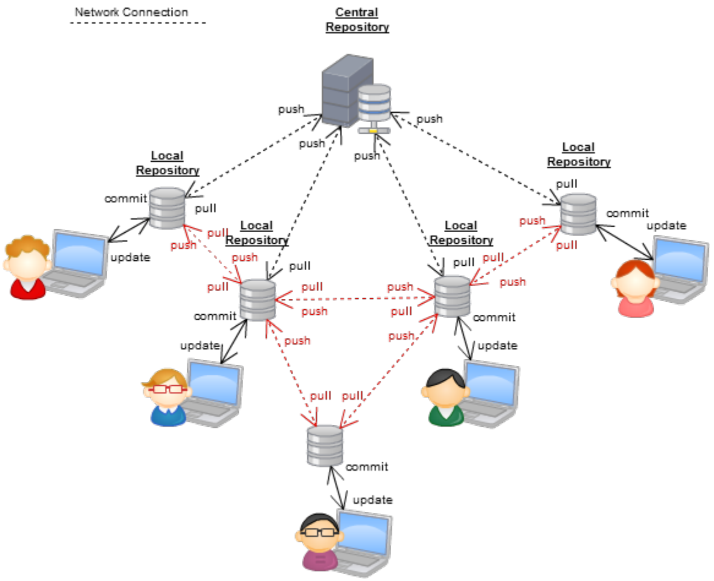
Distributed version control systems (dVCS) allow many repositories of the same software to be managed, merged, etc.

- reduces dependence on single physical node
- allows people to work (including check in, with log comments etc.) while disconnected
- much faster VC operations
- much better support for branching
- makes it easier to republish versions of software
- But... much more complicated and harder to understand

Distributed VCS



Distributed VCS



Branches

Simplest use of a VCS gives you a single linear sequence of versions of the software.

Sometimes it's essential to have, and modify, two versions of the same item and hence of the software: e.g., both

- the version customers are using, which needs bugfixes, and
- a new version which is under development

As the name suggests, branching supports this: you end up with a tree of versions.

What about merging branches, e.g., to roll bugfixes in with new development?

With CVS/SVN, this is very hard. Distributed version control systems support it much better, so developers use branches a lot more.

Releases

Releases are configurations packaged and released to users.

alpha release for friendly testers only: may still be buggy, but maybe you want feedback on some particular thing

beta release for any brave user: may still have more bugs than a real release

release candidate sometimes used (e.g. by Microsoft) for something which will be a real release unless fatal bugs are found

bugfix release e.g. 2.11.3 replaces 2.11.2 - same functionality, but one or more issues fixed

minor release e.g. 2.11 replacing 2.10: basically same functionality, somehow improved

major release e.g. 3.0 replacing 2.11: significantly new features.

Variants, e.g. even (stable) versus odd (development) releases...

Build tools

Given a large program in many different files, classes, etc., how do you ensure that you recompile one piece of code when another than it depends on changes?

On Unix (and many other systems) the `make` command handles this, driven by a *Makefile*. Used for C, C++ and other 'traditional' languages (but not language dependent).

part of a Makefile for a C program

```
OBJS = ppmtomd.o mddata.o photocolcor.o vphotocolcor.o dyesubcolcor.o
ppmtomd: $(OBJS)
        $(CC) -o ppmtomd $(OBJS) $(LDLIBS) -lpnm -lppm -lpgm -lpbm -lm

ppmtomd.o: ppmtomd.c mddata.h
        $(CC) $(CDEBUGFLAGS) -W  -c ppmtomd.c

mddata.o: mddata.c mddata.h
```

Makefiles list the *dependencies* between files, and the commands to execute when a depended-upon file is newer.

`make` has many baroque features – and exists in many versions. (Just use GNU Make.)

Like version control, a Makefile is something *every* C program should have if you want to stay sane.

Ant

`make` can be used for Java.

However, there is a pure Java build tool called Ant.

Ant *Buildfiles* (typically `build.xml`) are XML files, specifying the same kind of information as `make`.

There is an Eclipse plugin for Ant.

part of an Ant buildfile for a Java program

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name ="Dizzy" default = "run" basedir=".">
<description>
    This is an Ant build script for the Dizzy chemical simulator. [...]
</description>
<!-- Main directories -->
<property name = "source"      location = "${basedir}/src"/> [...]
<!--General classpath for compilation and execution-->
<path id="base.classpath">
  <pathelement location = "${lib}/SBWCore.jar"/> [...]
</path> [...]
<target name = "run" description = "runs Dizzy"
                     depends =" compile, jar">
  <java classname="org.systemsbiology.chem.app.MainApp" fork="true">
    <classpath refid="run.classpath" />
    <arg value="." />
  </java>
</target> [...]
</project>
```

Maven

Maven extends Ant's capabilities to include management of dependencies on external libraries

Maven *buildfiles* (typically `pom.xml`) are XML files, specifying the same kind of information as `Ant` buildfiles but also which classes and packages depend on which versions of which libraries.

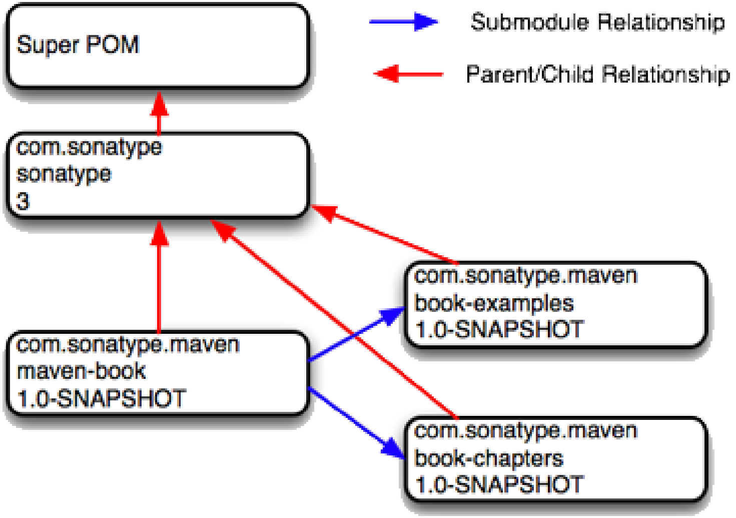
Maven is considerably more complex than Ant, and considerably more useful for projects using many external libraries (i.e., most Java projects).

There is an Eclipse plugin for Maven (actually, two).

A Maven buildfile for a Java program

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://ww
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.ap
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Maven Parent-Child



Per-platform code configuration

Different operating systems and different computers require code to be written differently. (Incompatible APIs. . .). Writing portable code in C (etc.) is hard.

Tools such as GNU Autoconf provide ways to automatically extract information about a system. The developer writes a (possibly complex) configuration file; the user just runs a shell script produced by autoconf.

(Canonical way to install Unix software from source:
`./configure; make; make install`.)

A newer tool is CMake.

Problem is less severe with Java. (Why?) But still tricky to write code working with all Java dialects.

Reading

Required: Chapter 1, Fundamental Concepts, of the SVN book
`http://svnbook.red-bean.com/`

Suggested: man rcsintro

Suggested: Eclipse Team Programming with CVS (see above)

Suggested: Tutorial about dVCS, `http://hginit.com/00.html`

Construction: High quality code for ‘programming in the large’

Paul Jackson

School of Informatics
University of Edinburgh

What is high quality code?

High quality code does what it is supposed to do...

... and will not have to be thrown away when that changes.

Obviously intimately connected with requirement engineering and design: but today let's concentrate on the code itself.

What has this to do with programming in the large?

Why is the quality of code more important on a large project than on a small one?

Fundamentally because other people will have to read and modify your code – even you in a year's time count as "other people"! E.g.,

- because of staff movement
- for code reviews
- for debugging following testing
- for maintenance

(Exercise. Dig out your early Java exercises from Inf1. Criticise your code. Rewrite it better *without changing functionality*.)

How to write good code

- Follow your organisation's coding standards - placement of curly brackets, indenting, variable and method naming...

Coding standard example

- Suppose you are used to...

```
public Double getVolumeAsMicrolitres() {
      if (m_volumeType.equals(VolumeType.Millilitres))
            return m_volume * 1000;
      return m_volume;
}
```

- ... and you see

```
public Double getVolumeAsMicrolitres()
  {
    if(m_volumeType.equals( VolumeType.Millilitres))
      {
        return m_volume*1000;
      }
    return m_volume;
  }
```

- Even worse: mixed styles in one file - inevitable without standards!

How to write good code

- Follow your organisation's coding standards - placement of curly brackets, indenting, variable and method naming...
- Use meaningful names (for variables, methods, classes...) If they become out of date, change them.
- Avoid cryptic comments. Try to make your code so clear that it doesn't need comments.
- Balance structural complexity against code duplication: don't write the same two lines 5 times (why not?) when an easy refactoring would let you write them once, but equally, don't tie the code in knots to avoid writing something twice.
- Be clever, but not too clever. Remember the next person to modify the code may be less clever than you! Don't use deprecated, obscure or unstable language features unless absolutely necessary.
- Remove dead code, unneeded package includes, etc.

Which of these fragments is better?

1.
```
for(double counterY = -8; y < 8; counterY+=0.5){
      x = counterX;
    y = counterY;
    r = 0.33 - Math.sqrt(x*x + y*y)/33;
    r += sinAnim/8;
    g.fillCircle( x, y, r );
 }
```
2.
```
for(double counterY = -8; y < 8; counterY+=0.5){
  x = counterX;
  y = counterY;
  r = 0.33 - Math.sqrt(x*x + y*y)/33;
  r += sinAnim/8;
  g.fillCircle( x, y, r );
}
```
3. They are both fine.

Be consistent about indentation. Don't use TABs

Which of these fragments is better?

1. `c.add(o);`
2. `customer.add(order);`
3. They are both fine.

Use meaningful names.

- A good length for most names is 8-20 chars
- Follow conventions for short names
  - e.g. `i`, `j`, `k` for loop indexes

What else is wrong with this?

```
r = 0.33 - Math.sqrt(x*x + y*y)/33;
r += sinAnim/8;
g.fillCircle( x, y, r );
```

Use white space consistently

Use comments when they're useful

```
if (moveShapeMap!=null) {
    // Need to find the current position. All shapes have
    // the same source position.
    Position pos = ((Move)moveShapeSet.toArray()[0]).getSource();
    Hashtable legalMovesToShape = (Hashtable)moveShapeMap.get(pos);
    return (Move)legalMovesToShape.get(moveShapeSet);
}
```

and not when they're not

```
// if the move shape map is null
if (moveShapeMap!=null) {
```

Too many comments is actually a more common serious problem than too few.

Good code in a modern high-level language shouldn't need many *explanatory* comments, and they can cause problems.

"If the code and the comments disagree, both are probably wrong" (Anon)

But there's another use for comments...

Javadoc

Any software project requires documenting the code – *by which we mean specifying it, not explaining it*.

Documentation held separately from code tends not to get updated.

So use comments as the mechanism for documentation – even if the reader won't need to look at the code itself.

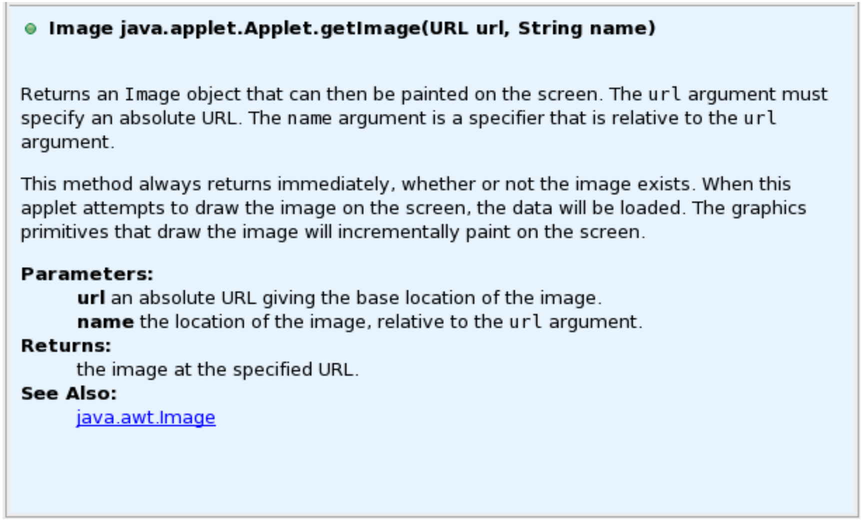
Javadoc is a tool from Sun. Programmer writes doc comments in particular form, and Javadoc produces pretty-printed hyperlinked documentation. E.g. Java API documentation at `http://docs.oracle.com/javase/7/docs/api/`

See Required reading for tutorial.

Javadoc example from tutorial

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
 }
```

Rendered Javadoc (Eclipse)



● **Image java.applet.Applet.getImage(URL url, String name)**

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
- **url** an absolute URL giving the base location of the image.
- **name** the location of the image, relative to the `url` argument.

Returns:
- the image at the specified URL.

See Also:
- java.awt.Image

Coding style

Good naming, commenting and formatting are vital components of a good coding style.

But good coding is about much more too. E.g.

- Declaration and use of local variables
  - Good to try keeping local variable scope restricted
- How conditional and loop statements are written
  - With *if-else* statements, put normal frequent case first
  - Use *while*, *do-while*, *for* and *for/in* loops appropriately
  - Avoid deep nesting
- How code is split among methods
  - Good to try avoiding long methods, over 200 lines
- Defensive programming
  - using assertions and handling errors appropriately
- Use of OO features and OO design practices (e.g. patterns)
- Use of packages

The relevance of object orientation

Construction is intimately connected to design: it is design considerations that motivate using an OO language.

Key need: control complexity and abstract away from detail. This is essential for systems which are large (in any sense).

Objects; classes; inheritance; packages all allow us to think about a lot of data at once, without caring about details.

Interfaces help us to focus on behaviour.

Revise classes, interfaces and inheritance in Java.

A common pattern in Java

```
interface Foo {
...
}

class FooImpl implements Foo {
...
}
```

Why is this so much used?

Use of single implementations of interfaces

All about information hiding

- Expect that users of `FooImpl` objects always interact with them at type `Foo`.
  - Maybe they are not even aware of `FooImpl` name.
  - Alternative means of constructing `FooImpl` objects can be set up
- Can use access control modifiers (e.g. *private*, *public*) on `FooImpl` members to define its interface
  - However interface and implementation still mixed together in one class definition
  - This was an early criticism of the OO take on the ADT paradigm
- With distinct interface `foo`, users need see nothing of `FooImpl`

Packages

Recall that Java has packages. Why, and how do you decide what to put in which package?

Packages:

- are units of encapsulation. By default, attributes and methods are visible to code in the same package.
- give a way to organize the namespace.
- allow related pieces of code to be grouped together.

So they are most useful when several people must work on the same product.

But

- the package "hierarchy" is not really a hierarchy as far as access restrictions are concerned – the relationship between a package and its sub/superpackages is just like any other package-package relationship.

Reading

Required: something that makes you confident you completely understand Java packages (e.g., see course web page).

Required: the JavaDoc tutorial

Suggested: *Code Complete* 2nd Ed. Steve McConnell

Verification, validation and testing

Paul Jackson

School of Informatics
University of Edinburgh

Verification, validation and testing

"VV&T" generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).

Verification: are we building the software right?

Validation: are we building the right software?

Testing is a useful (but not the only) technique for both.

Other techniques useful for verification:

- static analysis
- reviews/inspections/walkthroughs

Other techniques useful for validation:

- prototyping/early release

“Bug”: or more precisely:

From IEEE610.12-90 (IEEE Standard Glossary of Software Engineering Terminology):

- Fault: An incorrect step, process, or data definition in a computer program
- Mistake: A human action that produces a fault
- Failure: The [incorrect] result of a fault
- Error: The difference between a computed result and the correct result

The common term “defect” usually means fault.

Testing

Ways of testing: black box (specification-based) and white box (structural).

Different testing purposes:

- Module (or unit) testing
- Integration testing
- System testing
- Acceptance testing
- Stress testing
- Performance testing

and many more. i.e., large area: whole third-year course on testing. Basics only here. For more see SWEBOK.

Why test?

Testing has three main purposes:

- to help you find the bugs
- to convince the customer that there are no/few bugs
- to help with system evolution.

Crucial attitude: *A successful test is one that finds a bug.*

How to test

Tests often have a contractual role. For this and other reasons, they must be:

- repeatable
- documented (both the tests and the results)
- precise
- done on configuration controlled software

Ideally, test spec should be written at the same time as the requirements spec: this helps to ensure that the requirements are highly testable. It may seem premature to consider testability when writing requirements but it's now standard.

E.g. may write requirements in numbered sentences and keep a tally of which test(s) tests which requirement(s). Use cases may help structure tests.

Evolving tests when they don't catch new bugs

Assume an implementation passes all current tests.

What if a new bug is identified by customer or by code review?

A good discipline is:

1. Fix or create a test to catch the bug.
2. Check that the test fails.
3. Fix the bug
4. Run the test that should catch this bug: check it passes
5. Rerun *all* the tests, in case your fix broke something else.

Test-first development

The motivating observation: tests implicity define

- interface, and
- specification of behaviour

for the functionality being developed.

Basic idea is

- write tests **before** the code that they testing.
- run tests as code is written,

As a consequence:

- bugs found at earliest possible point
- bug location is relatively easy

Further advantages of test-first development

TFD

1. **ensures adequate time for test writing**: If coding first, testing time might be squeezed or eliminated. That way lies madness.
2. **clarifies requirements**: trying to write a test often reveals that you don't completely understand exactly what the code *should* do.
3. **avoids poor ambiguity resolution**: if coding first, ambiguities might be resolved based on what's easiest to code. This can lead to user-hostile software.

Test-driven development

A subtly different term, covers the way that in Extreme Programming detailed tests *replace* a written specification.

Test automation and JUnit

Automation of tests is essential, particularly when tests must be re-run frequently.

JUnit is a framework for automated testing of Java programs. It's use is required in Coursework 3.

Lots of sources of help. E.g.:

- `http://www.junit.org`
- *Using JUnit with Eclipse IDE* `http://www.onjava.com/pub/a/onjava/2004/02/04/juie.html` (good introduction, details may not be quite right for the version we have)
- *Writing and running JUnit tests* from the Eclipse help documentation, Java Development User Guide, Getting Started, Basic Tutorial.
- *JUnit Tutorial* `//http://www.vogella.de/articles/JUnit/article.html`

Limitations of testing

- **Writing tests is time-consuming**
- **Coverage almost always limited**: may happen not to exercise a bug.
- **Difficult/impossible to emulate live environment perfectly**
    - e.g. *race conditions* that appear under real load conditions can be hard to find by testing.
- **Can only test executable things**, mainly code, or certain kinds of model – not high level design.

Reviews/walkthroughs/inspections

One complementary approach is to get a group of people to look for problems.

This can:

- find bugs that are hard to find by testing;
- also work on non-executable things, e.g. requirements specification
- tease out issues where the problem is that what's "correct" is misunderstood
- spot unmaintainable code.

Of course the author(s) of each artefact should be looking for such problems – but it can help to have outside views too.

For our purposes reviews/walkthroughs/inspections are all the same; there are different versions with different rules. "Review" for short.

Reviews, key points

A review is a meeting of a few people,

which reviews one specific artefact (e.g., design document, or defined body of code)

for which specific entry criteria have been passed (e.g., the code compiles).

Participants study the artefact before the meeting.

Someone, usually the main author of the artefact, presents it and answers questions.

The meeting does not try to fix problems, just identify them.

The meeting has a fixed time limit.

Static and dynamic analysis

Neither testing nor reviews are reliable ways to find subtle technical problems in code. Formal tool-supported analysis not limited to specific test cases can help. Roughly

- Dynamic analysis involves running the code (or at least, simulating it)
- Static analysis does not: the tool inspects the code without running it.

Both require clear specification of what is being checked. This can be given generically (“this code does not deadlock”, “no null-pointer exception will ever be raised”) or in terms of annotations of the code (“for every class, the class invariant written in the code will always be true when a public method is invoked”). Let’s look at such annotations.

Assertions

Assertions allow the Java programmer to do ‘sanity checks’ during execution of the program.

Suppose `i` is a integer variable, and we are writing a bit of code where we ‘know’ that `i` must be even (because of what we did earlier). We can write

```
assert i % 2 == 0;
```

to check this – if `i` is odd, an `AssertionError` exception is raised.

Assertion checking can (and should, for release) be switched off. Therefore, **never** do anything with *side-effects* in an assertion!

Preconditions, postconditions, invariants

Particularly common types of assertion about methods and classes are:

Precondition: a condition that must be true when a method (or segment of code) is invoked.

Postcondition: a condition that the method guarantees to be true when it finishes.

Invariant: a condition that should always be true of objects of the given class.
What does always mean? In all *client-visible* states: that is, whenever the object is not executing one of its methods.

Writing these conditions would be tedious – and we might want to write richer conditions than can be expressed in Java.

Java Modeling Language

The Java Modeling Language is a way of annotating Java programs with assertions. It uses Javadoc-style special comments.

Preconditions: `//@ requires x > 0;`

Postconditions: `//@ ensures \result % 2 == 0;`

Invariants: `//@ invariant name.length <= 8;`

General assertions: `//@ assert i + j = 12;`

JML allows many extensions to Java expressions. For example, quantifiers `\forall` and `\exists`. And much, much more.

Dynamic analysis

Running the program with assertion checking turned on is a basic kind of dynamic analysis. But more is possible.

The tools jmlc, jmlrac etc. compile and run JML-annotated Java code into bytecode with specific runtime assertion checking.

Required/Recommended Reading: Leavens and Cheon ‘Design by Contract with JML’, via `www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf` Section 1 is Required, the rest is Recommended. You should be able to read and write simple examples, like those in Section 1.

Static analysis

NB even type-checking during compilation is a kind of static analysis!

Static analysis has advanced a lot in recent years, and is much more practical for relatively "clean" languages like Java than for e.g. C++ (pointer arithmetic makes everything harder!).

Tools vary in what problems they address, e.g.

- common coding oversights such as failing to check return values for error codes
- correctness of pre/post-condition specification of methods
- concurrency bugs e.g. race conditions

Think of what these tools do as findings bugs, just like testing. Their methods are not usually sound (every problem flagged guaranteed to be a real error) or complete (guaranteed to find every error) – undecidability looms.

Static analysis tools for Java

OpenJML project provides support for static analysis of JML-annotated programs.

Treats programs as mathematical objects and automatically proves assertions.

Project also supports dynamic analysis tools.

FindBugs is relatively widely used: looks for *bug patterns*, code idioms that are often errors

ThreadSafe from the Informatics spinout *Contemplate* focusses on finding concurrency bugs

Reading

Required: GSWEBOK Ch11, on Software Quality (could delay till after discussion of process)

Required: some JUnit information, see slide above. You must know how to create and run a JUnit 4 test for a method of a class - the best way to learn this is to do it.

Required/Suggested: Design by Contract with JML (Section 1 required), see above

Suggested: GSWEBOK Ch5, on Software Testing

Suggested: Stevens Ch19.

Suggested: Sommerville Ch 8 (9th and 10th Ed), Chs 22-24 (8th Ed)

Refactoring

Paul Jackson

School of Informatics
University of Edinburgh

Refactoring definition

Refactoring is the process of re-organizing and re-writing code so that it becomes cleaner, or fits better into the current conception of the architecture.

Refactoring *does not change functionality*.

Why refactor?

Refactoring was once seen as a kind of maintenance. . .

- You've inherited legacy code that's a mess.
- A new feature is required that necessitates a change in the architecture.

But can also be an integral part of the development process

Agile methodologies (e.g. XP) advocate continual refactoring (XP maxim: "Refactor mercilessly").

What does refactoring do?

A refactoring is a *small* transformation which preserves correctness.

There are many examples.
For a catalogue of over 90 assembled by Martin Fowler, see `http://refactoring.com/catalog/`.

A sample:

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Extract Variable (Introduce Explaining Variable)
- Replace Conditional with Polymorphism

Extract Variable

Change

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
     (browser.toUpperCase().indexOf("IE") > -1) &&
      wasInitialized() && resize > 0 )
{
  // do something
}
```

to

```
final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  > -1;
final boolean wasResized  = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized)
{
  // do something
}
```

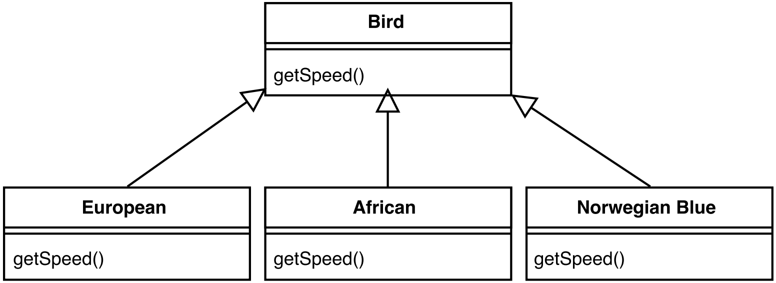
Replace Conditional with Polymorphism I

Change

```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case NORWEGIAN_BLUE:
      return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }
  throw new RuntimeException ("Should be unreachable");
}
```

Replace Conditional with Polymorphism II

to



Eclipse Refactoring

Eclipse has a built-in refactoring tool (on the Refactor menu).

Many of its refactoring operation can be grouped in three broad classes . . .

Eclipse Refactoring I: Renaming and physical reorganization

A variety of simple changes.

For example:

- Rename Java elements (classes, fields, methods, local variables)
  - On class rename, `import` directives updated
  - On field rename, getter and setter methods also renamed
- Move classes between packages
- `package` and `import` directives updated

Eclipse applies these changes *semantically*

- Much better than syntactic search-and-replace

Eclipse Refactoring II: Modifying class relationships

Heavier weight changes. Less used, but seriously useful when they are used. E.g.

- Move methods or fields up and down a class inheritance hierarchy.
- Extract an interface from a class
- Turn an anonymous class into a nested class

Eclipse Refactoring III: Intra-class refactorings

The bread-and-butter of refactoring: rearranging code within a class to improve readability etc. E.g.

- Extract Method: pull method code block into new method.
  - Good for shortening method or making block reusable
  - Also can extract local variables and constants
- Encapsulating fields in accessor methods.
- Change the type of a method parameter or return value

Safe refactoring

How do you know refactoring hasn't changed/broken something?

Perhaps somebody has *proved* that a refactoring operation is safe.

More realistically:

test, refactor, test

This works better the more tests you have: ideally, unit tests for every class.

Reading

Required: The article ‘Refactoring for everyone’ at `http://www.ibm.com/developerworks/opensource/library/os-ecref/`. Aim to remember: what refactoring is, and a few examples, not the details of the refactorings discussed here.

Suggested: Look at the *Reference - Refactor Actions* section of the Eclipse *Java development user guide* for full information on Eclipse’s current capabilities.

Suggested: Browse around Fowler’s page at `http://refactoring.com/`. Some of his book *Refactoring* is available on Google Books e.g., details of some of the refactorings in the catalogue.