Small- and medium-scale design

Whenever you think about writing a class, you're doing design

- What functions does the class provide?
- What functions does it rely on?
- What functions can the programmer change?
- What features might change at run-time?
- What features are shared by all instances? What features should be in each instance?

The features of changeability, abstraction, coupling and cohesion are perhaps most obvious at this level – get a feel for them before moving to the larger (and more critical) scales

From specification to design

Specification is a precise statement of *what* was agreed to be built

Design is a statement of *how* the system will be built, in technical terms

- The division of functionality between modules/classes/methods
- The relationships between the parts

Don't forget: design decisions have a major impact on the maintenance and evolution of a system – in other words, good design is the key engineering skill

Small-scale design

The construction of individual classes

- What methods and variables to include
- Who should be able to see/alter what

Key concepts

- Feature sets – what goes in the class
- Generality – use the most general approach
- Abstraction – hide what can reasonably be hidden
- Generalisation and substitutability – avoid over-dependence
- Documenting behaviours – say what it does and how it does it

Basic techniques

Where do classes come from?

- Domain analysis – the nouns in requirements
- "Basic" computer science structures – lists, hash tables, …
- Re-used from elsewhere – sometimes have to adopt someone else's view of the world, *e.g.* Java's view of windowing

What does each class do?

- Its behaviour
- Its assumptions about the world

How is the application's functionality divided into classes?

What specialisations are possible?

- General *versus* specific functions
- E.g., information storage *versus* arrays, files

What can be provided generically, and what must be provided specifically?

What do clients need to know?

- Exposed methods and variables
- Any implications of particular algorithms

Anything exposed limits possible changes

UML for design

Basically the same kinds of diagrams, with more detail

- Class diagrams with types
- Activity diagrams more down at the implementation level
- …

Often just a refinement exercise – take the business objects to implementation

```
Order
--------------
dateReceived
isPrePaid
price
--------------
dispatch()
close()
```

```
Order
--------------
Date dateReceived
boolean isPrePaid
long price
--------------
void dispatch( Date )
void close()
```

Just added (Java) types to the members

Good and bad feature sets

The *software* doesn't care how you split features across classes; it's the *engineers* who care

- Know what bit does what, and what each bit does

Make classes strongly cohesive and weakly coupled

- Perform an identifiable, bounded function – can you describe a class' purpose in a single sentence?
- Does it need other classes? One definite class, or a member of a general abstraction?

Example – averaging numbers

```
class ArrayAverager {
   public int average( int[] as ) { … }
}
```

Must be an array of ints – no arguments

```
class VectorAverager {
   public int average( Vector as ) { … }
}
```

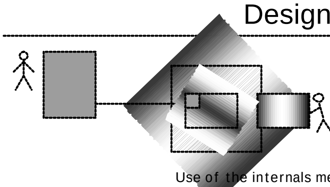
Can be any kind of vector, but that's still only one kind of collection

```
class Averager {
   public int average( Enumeration enum ) { … }
}
```

Can average any collection

In Java the last two are a bit more expensive in terms of run-time type checking

Designing software



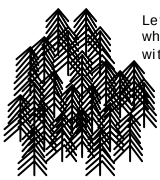
"I know how that data structure is implemented, so I'll just change it directly a bit…"

Use of the internals means that a change here propagates into all users

In the "good old days", all software was written in one piece – usually in one file

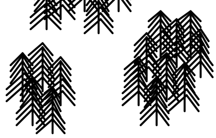
- Any piece of the code can change any other
- No idea what effect a change can have

Designing forests

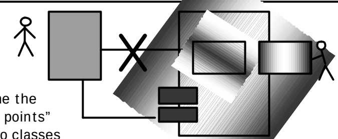


Left to their own devices, forests burn down whenever there's a fire: they're a single system with no boundaries to contain problems

Putting fire-breaks into the forest means that a single fire, whilst locally damaging, isn't globally disastrous



Abstraction boundaries



Methods define the allowed "entry points" for clients into classes

Define exactly what a client can see, and *enforce it*

- Can only access what they're explicitly allowed to access
- Changing something, but keeping the view the same, makes the change invisible

A class like this implements an *abstraction* – an ideal, pure version of a concept

Abstraction in Java

A Java class' interface is formed by the `public` members – methods and variables

The `private` members are inaccessible to clients – they lie within the abstraction boundary

The `protected` members are accessible to only a certain kind of classes – sub-classes and those in the same package

Interfaces restrict exposure to methods – no variables to modify

Sometimes called a "functional interface" because there's always programmer-written code round every change

The Liskov Substitutability Principle

"If A is a sub-class of B, then an instance of A may be substituted anywhere an instance of B is expected"

E.g., Customer – BusinessCustomer – PrivateCustomer

All the rules of Java (and other O-O languages) are built around maintaining the LSP

- The basis for inclusion polymorphism

Methods in sub-classes must behave "the same" as those in parent classes

Effects

- Language must keep method signatures compatible
- Programmer must keep method behaviours compatible

Named after Barbara Liskov, one of the pioneers of language design

Look for general abstractions

`java.util.Enumeration` is the general abstraction of “running over a collection of objects”

- Doesn’t matter where the objects come from – they might be generated on the fly, at each call – who cares?

Often find lots of special cases of “the same thing”

- Collections of objects; sorts of user interface widgets

LSP guarantees that programs written for the general case can handle all the special cases

What it doesn’t guarantee is that the program is still meaningful – that relies on the programmer making all the methods “do the right thing” in each sub-class

Design by contract

Methods aren’t just coded for effect: most transform data in some way

- *Rely* on certain things being true when called
- …and, given that they *were* true, *guarantee* certain things to be true after the method terminates

Rely and guarantee conditions can form the basis of *designing by contract*

Basically a structured form of documentation

- Mathematically stated, may be automatically checkable
- Encourages precision – which may help avoid some problems before they occur

The idea is described in Bertrand Meyer, *Object-oriented software construction*, Prentice Hall (1994)

The contract

Each method includes two assertions

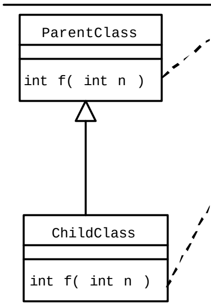
- **Pre-condition**: must be true when the method is called
- **Post-condition**: will be true afterwards *if and only if* the pre-conditions were true

Each class has an additional assertion

- **Invariant**: must always be true (*except* it can become transiently false during the actual execution of a method)

Meyer’s programming language, Eiffel, provides this model as part of the language; in Java - as we’ll see later - it can be included as part of structured approaches to bug-location

Assertions and sub-classing



{pre: n > 10,
post: f(n) > 2 * n}

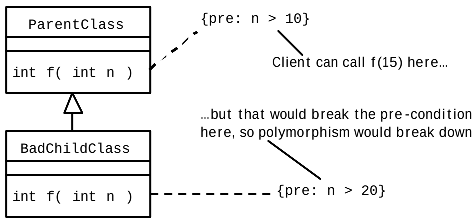
Pre-condition of sub-class must be *no stronger* than that of the parent

{pre: n > 5,
post: f(n) > 3 * n}

Post-condition of sub-class must be *at least as strong* as that of the parent (same for invariant)

Why is that?

Otherwise a client's rely and guarantee conditions aren't maintained under substitution – the LSP, again



Summary – small scales

Changes

- Restrict the visibility of "sensitive" structures and methods, so they can be changed without affecting clients
- Concentrate on behaviour rather than implementation
- Make sure sub-classes maintain the behaviours, in spirit if not in terms of mechanisms

Cohesion

- make a class do one thing well

Build complex systems from simple blocks

- Compose multiple simple blocks rather than building "super-size" classes

Moving on up the scale

Medium Scales

Moving up the scale introduces new relationships between classes

- One class *contains* another
- One class is *part of* another
- One class *uses* another

Object-oriented systems are built as *compositions* of classes each providing a well-defined behaviour