Software practice

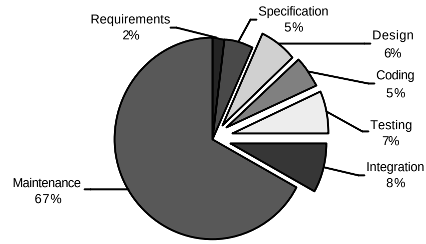
So far we've talked mainly about the *process* of developing software

- What to build
- Who wants it
- Where it'll be run

The next stage is to talk about the *practice* of development

- How to build it — Design
- How to actually do the building — Large-scale implementation
- Making sure we've built the right thing — Debugging and testing

26% "real work"



From Stephen Schach, *Classical and object-oriented software engineering*, Addison-Wesley (1999)

What's involved?

Software design

- Choosing how to arrange the software
- What makes a good design
- What happens as system size increases

Implementation techniques

- Coordinating a large team (technical, not managerial)
- Debugging – what bugs look like, where they hide, how to track them down

Testing

- Convincing yourself (and others) that you've built the right thing

Constructing software which can be evolved and maintained effectively

What is software design?

A (software) artefact consists of many more-or-less independent functions: how do you best put them together?

```
Type a character
Is is a special character?
    Is it a newline? -> …
    Is it a tab? -> …
Put the character on the screen
    Is it the end of a word? -> …
    Is is an insertion?
        Move characters along
    For each character on line
        Re-draw character
        Move cursor along
```

```
Type a character
Let control module filter special characters
Normal characters
    Add to buffer cursor position
    Call re-draw routine
```

These are two "correct" solutions: which is "better"?

Or, put another way, …

Software design is about developing a view of the artefact and then putting in place the structures needed to implement it effectively

- What functions should the system perform?
- What algorithms will the system use? What data do they need? How is this data best structured?
- How do the different functions relate to each other? Which of these dependencies are "hard"? Which might change?
- What needs to be exposed? What should be hidden?
- Which constraints are set at compile-time? Which must be deferred until configuration-time? Which until run-time?

The great mistake

Software doesn't care how its designed

- Badly-written software still runs
- Might actually run faster and in less memory

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." [Martin Fowler]

Engineers care, as do managers and (ultimately) users

- Important software is mostly maintenance
- You need to understand code before you can maintain it
- The easier the code is to understand, the less hassle (and cost) in maintenance

Well-designed, well-documented software is more likely to stand the tests of time and change

A Software Engineer's maxim

Design hard - code easy

Coding may be picky, tricky, repetitive, awkward and annoying – and usually all five

But if it ever gets so you don't know what's happening, **the design is wrong**

Coding a good design just flows, as it is obvious what needs to go where; coding a bad design hides where the functions should go

The approach

All design approaches share a single common thread of *mastering complexity*

Deal with the system in small chunks rather than all together

- No one person can possibly understand every nuance of a 10 million line program – but someone still needs to maintain it

Separate concerns

- Identify sub-systems in the full system
- Define how the sub-systems should interact – and stick to it
- Deal with each sub-system individually – possibly a completely different team for each, with co-ordination

Design is essentially fractal: the same issues turn up in different guises at all scales of the process

Characteristics of good designs

Adequate
- Meets the requirements – or 95% of them, anyway…

Simple
- The simplest solution to the problem – and no simpler

Obvious
- Easy to work out what bit does what, and what each bit does

Changeable
- Change the way a bit does its thing

Extensible
- Add new bits

Re-usable
- Use bits in contexts other than those for which they were built

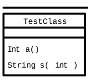
The issues

If you change something, what happens?
- How do small changes affect the whole system? How do you prevent changes affecting everything?

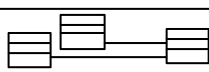
If someone else changes something, what happens?
- How do you isolate yourself from changes elsewhere? How far can you isolate yourself?

If you add something, what happens?
- Does new functionality fit into the scheme? Or does it hang off the edges?

Scales of design

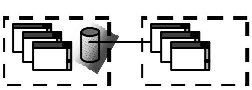


Class – what algorithms? What public methods?

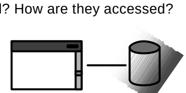


Component – what other classes are needed?

Application – what functions are included? How are they accessed?



Enterprise – can that application there access this database?



System – how do these applications interact?

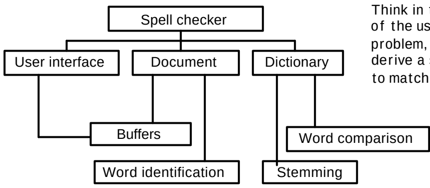
Issue scaling

The same issues appear in at different scales

Change management
- Individual classes – clients rely on a particular exposed data structure
- Packages – other packages use classes you might prefer they didn't
- Applications – other applications start using "undocumented" features that you then have to retain
- Businesses – users get attached to "features" that appeared by accident
- Enterprise – software hard-codes the source of a particular file to a particular machine

Top-down design

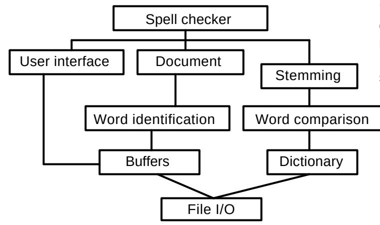
Decide on the top-level functions, sub-divide then recursively until you get something small enough to implement



Think in terms of the user's problem, then derive a solution to match

Bottom-up design

Find the shared functions, build them, then tie them together to form the application



Think in terms of modules needed to implement the solution

Comparison

Top-down design often fits well with domain analysis

- Take what you've learned about the problem and model it (as objects, modules, functions, …)

Bottom-up design tends to emphasise the code more than the user

- Can lead to better re-use, as common functions can be identified earlier

In practice, engineering uses a "middle out" approach

- Sweep down to define the problem from the user's perspective
- Fill-in the details from the bottom

This applies pretty much equally from small-scale design (classes) up to enterprise design (systems)

Change

Change occurs all the time

- Bug fixes, new features, new optimisations, …

If one part of a system changes, how does this affect the other parts?

- Need to be re-coded?
- Need to be completely re-designed?
- Can you determine which, *a priori?*

Seek to minimise the effects of changes

- Affect the smallest possible part of the system

Meyer’s notion of “linear” change

Bertrand Meyer (O-O Software Construction, Eiffel) noted that there are two kinds of changes:

- Those where the complexity of the change is proportional to the size of the change
- Those where the complexity is proportional to the size of the system

As system size increases, the first class (“linear” changes” stay tractable whilst the second (“non-linear” changes) rapidly become monsters

Without good design, *all* changes must be assumed to be non-linear

Why do changes hurt?

The effects of changes propagate for two reasons

1. Because the change has a functional effect – the changed system behaves differently
2. Because the change has an implementational effect – the changed system provides its behaviour in a different way

The first kind of change is just fundamentally hard

The second kind is only hard if other parts of the system rely on *how* something happens rather than just on *what* happens

If we restrict dependencies to *what* rather than *how* a complete class of hard changes is eliminated

- Concentrate on keeping the behaviour consistent

Dependencies

Classes which rely heavily on each other are said to be *strongly coupled*; those which don’t are *weakly coupled*

Suppose we have a list data structure and a sorting algorithm: can we sort without changing the list? Can we change the list without changing the algorithm?

Effects of strong coupling

- ✗ Changes propagate – again
- ✗ Can’t re-use one class without taking the one it depends on
- ✗ Can’t easily replace the class with another
- ✓ Implementations can be optimised to the specific features of the other class

Factoring

If something appears twice, what do you do?

- Repeat it?
- Share it?

This decision is called *factoring*

- Repeat it – if it changes, every occurrence must be changed, some might be forgotten
- Share it – change it in one place, but everyone *has* to use the new version whether they want to or not

Cohesion – 1

As systems grow, it’s important to know where functionality is provided

- How was the code factored? What provides what?

The simplest way to achieve this is if all aspects of one feature are provided in one class/package/application

- Keep a function together with its parts, and keep functions separate

A class/package/application that provides a single abstraction is said to be *cohesive*

- `java.lang.reflect` is strongly cohesive – provides reflection into classes: if you import it, that’s what you’re doing
- `java.util` is quite weakly cohesive – provides a load of stuff that doesn’t fit anywhere else: if you import it, that tells you nothing

Cohesion – 2

Different kinds

- Co-incidental – functions just happen to be collected
- Logical – all perform similar tasks, like a set of I/O routines
- Sequential – all involved in modifying some state in strict sequence
- Functional – all contribute towards a single well-defined task, for example reflection
- Temporal – all get called together, such as all initialisation routines
- Communication – all share common on data

Good design can actually generate all these (apart from the first)

- `java.lang.reflect` – functionally cohesive
- `java.util` – largely co-incidental, although the collections classes are somewhat logically cohesive