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.

Outputs of design process

Outputs include

- models.
  - E.g. using UML or SImulink
  - Often graphical
  - Can be executable
- Written documents
  - Important that these record reasons for decisions

(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?

Examples of architectures

- Client-server
- Peer to peer
- Message bus

See the *Suggested Readings* for more on these and other examples.

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.:

- System architecture has been settled by a small team written down, and reviewed.
- You are in charge of the detailed design of one subsystem.
- 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.

Software Design Principles

Key notions that provide the basis for many different software design approaches and concepts.

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
- encapsulation / information hiding
  Grouping and packaging the elements and internal details of an abstraction and making those details inaccessible'
- separation of interface and implementation
  Specifying a public interface, known to the clients, separate from the details of how the component is realized.

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.

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 UML standard;
- 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.

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.*