Software design and modelling

Nigel Goddard

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 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)

(Some) criteria for a good design

- It can meet the known requirements (functional and non-functional)
- It is verifiable: that is, it is practical to check that it meets the requirements

(Some) criteria for a good design

- It can meet the known requirements (functional and non-functional)
- It is verifiable: that is, it is practical to check that it meets the requirements
- It is straightforward to explain to implementors

(Some) criteria for a good design

- It can meet the known requirements (functional and non-functional)
- It is verifiable: that is, it is practical to check that it meets the requirements
- It is straightforward to explain to implementors
- It is maintainable: that is, it will be possible to adapt it to meet future requirements

(Some) criteria for a good design

- It can meet the known requirements (functional and non-functional)
- It is verifiable: that is, it is practical to check that it meets the requirements
- It is straightforward to explain to implementors
- It is maintainable: that is, it will be possible to adapt it to meet future requirements
- It makes appropriate use of existing technology, e.g. reusable components

(Some) criteria for a good design

- It can meet the known requirements (functional and non-functional)
- It is verifiable: that is, it is practical to check that it meets the requirements
- It is straightforward to explain to implementors
- It is maintainable: that is, it will be possible to adapt it to meet future requirements
- It makes appropriate use of existing technology, e.g. reusable components

(Some) criteria for a good design

- It can meet the known requirements (functional and non-functional)
- It is verifiable: that is, it is practical to check that it meets the requirements
- It is straightforward to explain to implementors
- It is maintainable: that is, it will be possible to adapt it to meet future requirements
- 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;

(Some) criteria for a good design

- It can meet the known requirements (functional and non-functional)
- It is verifiable: that is, it is practical to check that it meets the requirements
- It is straightforward to explain to implementors
- It is maintainable: that is, it will be possible to adapt it to meet future requirements
- 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.

A quotation from Donald Schön

Designers put things together and bring new things into being, dealing in the process with many variables and constraints, some initially known and some discovered through designing. Almost always, designers' moves have consequences other than those intended for them. Designers juggle variables, reconcile conflicting values, and maneuver around constraints – a process in which, although some design products may be superior to others, there are no unique right answers.

Donald A. Schön
Educating the Reflective Practitioner
Jossey-Bass, San Francisco, 1987.

Levels of design

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

- what programming languages will your system use?

Levels of design

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

- what programming languages will your system use?
- how is your system split up into subsystems? (high-level, or architectural, design)

Levels of design

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

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

Levels of design

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

- what programming languages will your system use?
- how is your system split up into subsystems? (high-level, or architectural, design)
- what are the classes in your system? (low-level, or detailed, design)
- what are the responsibilities of each bit? (both levels)

Levels of design

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

- what programming languages will your system use?
- how is your system split up into subsystems? (high-level, or architectural, design)
- what are the classes in your system? (low-level, or detailed, design)
- what are the responsibilities of each bit? (both levels)
- what are the interfaces? (both levels)

Levels of design

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

- what programming languages will your system use?
- how is your system split up into subsystems? (high-level, or architectural, design)
- what are the classes in your system? (low-level, or detailed, design)
- what are the responsibilities of each bit? (both levels)
- what are the interfaces? (both levels)
- what messages are exchanged, in what order? (both levels)

What is architecture?

Many things to many people.

> The way that components work together

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

What is architecture?

Many things to many people.

> 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 (Stuart Anderson).

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?

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?

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, but beyond scope of this course.

More examples of architectural decisions

- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)

More examples of architectural decisions

- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- what conventions do components have about error handling?

More examples of architectural decisions

- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- what conventions do components have about error handling?
- what framework does the product use?

More examples of architectural decisions

- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- what conventions do components have about error handling?
- what framework does the product use?

More examples of architectural decisions

- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- what conventions do components have about error handling?
- what framework does the product use?

Clean architecture helps get reuse of components.

By some definitions parts of the architecture can *be* components – frameworks, product-line architectures etc.

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.

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.)

Which of these two designs is better?

Clickers out!

Which of these two designs is better?

Clickers out!

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

Which of these two designs is better?

Clickers out!

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
}
```

Which of these two designs is better?

Clickers out!

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

Which of these two designs is better?

Clickers out!

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

Which of these two designs is better?

Clickers out!

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

Which?

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

A how similar the parts of a component are

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

A how similar the parts of a component are

B the degree to which a component reflects a functional abstraction

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

A how similar the parts of a component are

B the degree to which a component reflects a functional abstraction

C how clear the organisation of a component is

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

A how similar the parts of a component are

B the degree to which a component reflects a functional abstraction

C how clear the organisation of a component is

D the density of interaction between parts of a component

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

A how similar the parts of a component are

B the degree to which a component reflects a functional abstraction

C how clear the organisation of a component is

D the density of interaction between parts of a component

Which?

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

A how similar the parts of a component are

B the degree to which a component reflects a functional abstraction

C how clear the organisation of a component is

D the density of interaction between parts of a component

Which?

A better design has:

A higher coherence

B lower coherence

Design principles 1: Coherence

Coherence of a component (e.g., class, package) is:

A how similar the parts of a component are

B the degree to which a component reflects a functional abstraction

C how clear the organisation of a component is

D the density of interaction between parts of a component

Which?

A better design has:

A higher coherence

B lower coherence

Which?

Design principles 2: Coupling

Coupling between two components is:

Design principles 2: Coupling

Coupling between two components is:

A how similar the two components are

Design principles 2: Coupling

Coupling between two components is:

A how similar the two components are

B the degree of interaction between the components

Design principles 2: Coupling

Coupling between two components is:

A how similar the two components are

B the degree of interaction between the components

C how dependent one component is on the implementation of the other component

Design principles 2: Coupling

Coupling between two components is:

A how similar the two components are

B the degree of interaction between the components

C how dependent one component is on the implementation of the other component

D the extent to which one component depends on the other

Design principles 2: Coupling

Coupling between two components is:

A how similar the two components are

B the degree of interaction between the components

C how dependent one component is on the implementation of the other component

D the extent to which one component depends on the other

Which?

Design principles 2: Coupling

Coupling between two components is:

A how similar the two components are

B the degree of interaction between the components

C how dependent one component is on the implementation of the other component

D the extent to which one component depends on the other

Which?

A better design has:

A higher coupling

B lower coupling

Design principles 2: Coupling

Coupling between two components is:

A how similar the two components are

B the degree of interaction between the components

C how dependent one component is on the implementation of the other component

D the extent to which one component depends on the other

Which?

A better design has:

A higher coupling

B lower coupling

Which?

Design principles 3

So... if you had to pick two design principles, they'd be:

- maximize coherence
- minimize coupling

Why?

Design principles 3

So... if you had to pick two design principles, they'd be:

- maximize coherence
- minimize coupling

Why?

(human and compiler) understandability, maintainability

Design principles 4 (from GSWEBOKCh3)

- abstraction - procedural, data
  "the process of forgetting information so that things that are different can be treated as if they were the same"

Design principles 4 (from GSWEBOKCh3)

- abstraction - procedural, data
  "the process of forgetting information so that things that are different can be treated as if they were the same"
- decomposition, modularisation
  splitting up, "usually with the goal of placing different functionalities or responsibilities in different components"

Design principles 4 (from GSWEBOKCh3)

- abstraction - procedural, data
  "the process of forgetting information so that things that are different can be treated as if they were the same"
- decomposition, modularisation
  splitting up, "usually with the goal of placing different functionalities or responsibilities in different components"
- encapsulation
  "grouping and packaging the elements and internal details of an abstraction and making those details inaccessible"

Design principles 4 (from GSWEBOKCh3)

- abstraction - procedural, data
  "the process of forgetting information so that things that are different can be treated as if they were the same"
- decomposition, modularisation
  splitting up, "usually with the goal of placing different functionalities or responsibilities in different components"
- encapsulation
  "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 (from GSWEBOKCh3)

- abstraction - procedural, data
  "the process of forgetting information so that things that are different can be treated as if they were the same"
- decomposition, modularisation
  splitting up, "usually with the goal of placing different functionalities or responsibilities in different components"
- encapsulation
  "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."*
- sufficiency, completeness, primitivity
  "all the important characteristics of an abstraction, and nothing more."

Design principles 4 (from GSWEBOKCh3)

- abstraction - procedural, data
  "the process of forgetting information so that things that are different can be treated as if they were the same"
- decomposition, modularisation
  splitting up, "usually with the goal of placing different functionalities or responsibilities in different components"
- encapsulation
  "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."*
- sufficiency, completeness, primitivity
  "all the important characteristics of an abstraction, and nothing more."

Design principles 4 (from GSWEBOKCh3)

- abstraction - procedural, data
  "the process of forgetting information so that things that are different can be treated as if they were the same"
- decomposition, modularisation
  splitting up, "usually with the goal of placing different functionalities or responsibilities in different components"
- encapsulation
  "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."*
- sufficiency, completeness, primitivity
  "all the important characteristics of an abstraction, and nothing more."

Note crucial role of *interfaces*. This whole family of principles is about fitting very complex software into limited human brains.

Modelling

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 proprties you want.

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

Both can be done in different styles...

Pros and cons of BDUF

Pros and cons of BDUF

Big Design Up Front

- often unavoidable in practice

Pros and cons of BDUF

Big Design Up Front

- often unavoidable in practice
- if done right, simplifies development and saves rework;

Pros and cons of BDUF

Big Design Up Front

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

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.

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.

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: GSWEBOK2004 Ch3 (see web), for an overview of the field of software design

Suggested: Stevens Ch3, a simple case study; Somerville Ch14 on OOD (and nearby chapters, maybe)

Suggested: Browse SEI's collection of architecture definitions at `http://www.sei.cmu.edu/architecture/definitions.html`

Suggested: (architecture) Somerville ch 11-13

Quotes of the day

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.*

C.A.R. Hoare

> *Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.*

Eoin Woods, software architect, co-author of Software Systems Architecture : Working With Stakeholders Using Viewpoints and Perspectives