L12 - Design principles

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

UNIVERSITY OF ABERDEEN

12.1 Reminding past issues and mapping them to current topics

Where are we now?

Software development paradigms

⇒ The Unified Process (UP) paradigm

⇒ UP phases and UP disciplines (activities) within each phase

⇒ Elaboration (second UP phase)

⇒ ······

⇒ Design

⇒ Design principles

⇒ ······

12.2 Software design principles

Explicit commitments at design time

Design is a creative stage where a SW solution is devised for a problem. This means choosing 'classes' of SW technologies to deliver such solution:

- Programming paradigm (*e.g.*, object oriented, functional, . . .)
- Data management framework (*e.g.*, RDBMS, OODBMS, files, . . .)
- Standards (*e.g.*, communication protocols, data structures, . . .)

Design avoids committing to anything whose choice can be deferred (*e.g.*, specific programming languages, algorithms, implementation technologies)

What are software design principles?

1. Basic principles stating desirable **design characteristics** *that meet stakeholder needs and expectations:*
   - **Feasibility** - a design is acceptable only if it can be realised
   - **Adequacy** - designs that meet more stakeholder needs and desires, subject to constraints, are better
   - **Economy** - designs that can minimise costs, dev. time, risks, are better
   - **Modifiability** - designs that make a program easier to modify are better
2. Constructive principles stating desirable **SW quality requirements** *based on past development experience:*
   - **Modularity** - forming good 'modules' is essential in software design
   - **Implementability** - designs leading to easier software are better
   - **Aesthetic** - Beautiful (simple and powerful) designs are better

What is a module, precisely?

'Module' is a general term used to denote a conceptually simple and independent SW unit that communicates through well-defined interfaces.

- Although the term module could be used to denote a sizeable, heterogeneous part of a large SW system (*e.g.*, a GUI), emphasis is on (1) conceptual single-mindedness and (2) small size
- A module is intended to deliver a single (functional) requirement or a single, specific part of it. For the avoidance of doubt, a library is composed of many modules
- A module is a design concept – hence it indifferently maps into either source code or binary code at programming level
- A module is a design unit of work for all what follows design – it is a self-contained amount of SW that a single programmer/tester can develop/verify in between a few hours and a few days.

Top constructive principle: modularity

Modularity principles are design guidelines and evaluation criteria:

- **Small modules** – designs with small modules are better
- **Information hiding** (aka encapsulation) – each module should shield the details of its internal structure and processing from other modules
- **Least privilege** – modules should not have access to resources they do not need to perform their job
- **Minimal coupling** – the degree of connection between pairs of modules should be minimised.
- **Maximal cohesion** – the degree to which each part of a module is related to the other parts should be maximised

Modularity principles: min coupling and MAX cohesion

- **Coupling**: describes the degree of interconnectedness between design units (packages, modules, subsystems)
- Coupling measures how interdependent these elements are in a system. The higher the independency, the more likely a change to on element affects the others
- Coupling is reflected by (i) the number of links and (ii) the degree of interaction that an architectural unit has with other architectural units — HENCE, coupling should be minimised
- **Cohesion** is a measure of the degree to which a design unit contributes to a single purpose, *i.e.*, how specific and 'single-minded' a design unit is — HENCE, cohesion should be maximised

Implementability and aesthetic principles

Both principles derive from empirical observations and field experience

Implementability principles are design guidelines and evaluation criteria to achieve design economy:

- **Simplicity** - simpler designs are better (easier, cheaper, faster to develop and more likely to work)
- **Design WITH reuse** - designs that reuse existing assets are better
- **Design FOR reuse** - designs that create reusable assets are better

12.3 Architectural design issues

Architectural design: the starting point

- Architectural design should actually begin during inception as a sketch of the envisaged 'logical' (*i.e.*, conceptual) subsystems
- There is no clear boundary between architectural design (*i.e.*, *modular design*) and detailed design (*i.e.*, *class design*)
- There are no accepted strict standards for the abstraction level of an architectural design specification
- Software architectures must provide and specify structures that meet both functional and non-functional system requirements
- SW engineers must pay special attention to the way alternative structural solutions affect quality attributes (URPS+, *i.e.*, Usability, Reliability, Performance, Supportability, security, reusability, ...)

Architectural design: the outcomes

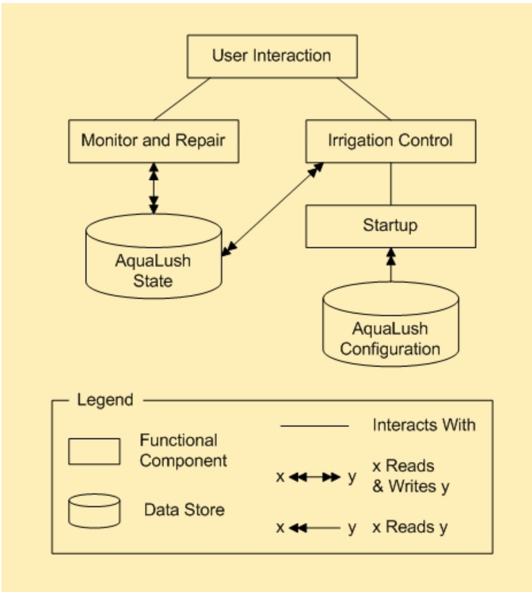
Architectural design leads to the specification of:

- **Modular decomposition** – how to divide a system into smaller units
- **States and transitions** – how relevant unit states change over time
- **Collaborations** – how units collaborate to provide services
- **Responsibilities** – units responsible for providing each service
- **Interfaces** – unit interfaces and communication protocols
- **Properties** – non-functional requirements 'met' by each unit
- **Relationships** – structural and functional links among units
  These are often called *dependencies*

12.4 Architectural design notations

Type of specification	*Notations*
Decomposition	Box-and-line diagrams, class diagrams, package diagrams, component diagrams, deployment diagrams
States	State diagrams
Collaboration	Sequence and communication diagrams, activity diagrams, box-and-line diagrams, use case models
Responsibilities	Text, box-and-line diagrams, class diagrams
Interfaces	Text, class diagrams
Properties	Text
Transitions	State diagrams
Relationships	Box-and-line diagrams, component diagrams, class diagrams, deployment diagrams, text

Architectural design: box-and-line diagrams



- Box-and-Line diagrams are NOT part of the UML set
- B&L diagrams have no formal semantics - hence should be clearly explained
- Keep boxes and lines simple; Make symbols for different things look different
- Use symbols consistently in different B&L diagrams
- Adopt usual conventions to name elements

Architectural design: specifying interfaces

An interface is a communications boundary between units.
An interface specification describes the unit mechanism to communicate with its environment. The following template is used:

1. **Services provided** — for each service provided specify its
   1. Syntax (elements of the communications medium and how they are combined to form messages)
   2. Semantics (the meaning of messages, using preconditions and postconditions)
   3. Pragmatics (how messages are used in context to accomplish tasks)
2. **Services required** — specify each required service by name. A service description may be included
3. **Usage Guide**
4. **Design Rationale**

12.5 Preparing for the topic ahead

Next week. . .

Architectural patterns:

More specifically, we will focus on:

- Pattern principles and taxonomy
- Detailed examples of architectural patterns