CITS4401 Software Requirements and Design
Week 11 – Design – review

Lecturer: Arran Stewart

Important concepts

- Interfaces, preconditions, postconditions
- Coupling, cohesion, partitions
- Design patterns
- Design documentation
- Design activities
  - Making use of our requirements & analysis artifacts
  - Dealing with constraints
- Event-driven systems
- Architecture
- Non-OO designs/agile methodologies

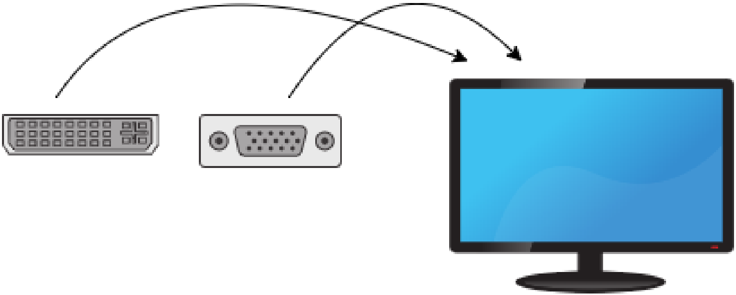
Interfaces

- We said that interfaces are the *boundaries where two things meet*.
  - We can have user interfaces; APIs (programming interfaces); hardware interfaces
- Coupling and interfaces
  - When we reduce coupling, we reduce the "surface area" between two things

Multiple interfaces

Just as something can have more than one “surface” in different directions (e.g. a cube has six faces, a dodecahedron 12), a component or system can present different interfaces.

We might say it presents different interfaces to different “audiences” (groups of external entities).



Multiple interfaces

An online service (like GitHub) can have multiple interfaces it presents – e.g. a web interface, and a command-line interface (accessed by using the `git` command).

Public and private interfaces

If a component is part of a package or subsystem – it might present different interfaces, depending on whether it's being accessed from *within* the package, or *from outside*.

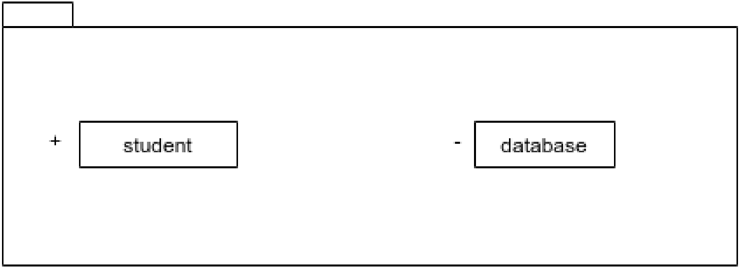
Public and private interfaces

We know this is true for objects – an object can have *private* methods and instance variables, which can only be accessed by other objects of that type – and *public* methods and instance variables, which can be accessed by anyone.

Public and private interfaces

And it is true for collections of objects or classes (systems, subsystems, and packages).

If we need to show this in UML, we can show *public* things with a plus (“+”), and private things with a minus (“-”).



Public and private interfaces

Once we've made something *public*, we've made it available to users (or programmers), and they'll complain if we later remove or change it.

Making things *private* prevents this – and by reducing "surface area" exposed, we reduce coupling.

Coupling

Recall that (at a subsystem level):

- High coupling means two subsytems depend on each other closely
  - Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)
- Low or "loose" coupling means two subsystems depend on each other less closely
  - A change in one subsystem does not affect any other subsystem

Design patterns

We saw a number of different design patterns – structural, behavioural, creational.

The *structural* patterns, in particular, often tend to focus on ways of *reducing coupling*.

There *are* disadvantages to reducing coupling:

- Code produced may be less efficient – there are more "layers of indirection" between objects
- It may become more difficult to see what the flow of control is (recall that this is so for event-driven systems, which are often fairly loosely coupled)
    - You can't just look at a class diagram and see "what interacts with what"
    - that information may now not be apparent from a class view

Design documentation

We saw why design documentation is important:
someone will need to *maintain* your system!

And as changes need to be made, they will want to know *why* you made particular design choices (and what the consequences might be of altering them).

Design documentation

We discussed whether you need to document *every* single design decision (no), and how you decide which ones to document and which ones not.

Design activities

We looked at some of the activities involved in system design – which use as "input" the artifacts of your requirements elicitation and analysis activities:

Transform the analysis model by

- defining the design goals of the project
- decomposing the system into smaller subsystems
- selection of off-the-shelf and legacy components
- mapping subsystems to hardware
- selection of persistent data management infrastructure
- selection of access control policy
- selection of global control flow mechanism
- handling of boundary conditions

Design activities

For each of these activities, we saw which bits of your analysis model you can use:

Making use of our analysis

- Nonfunctional requirements →
  - Design Goals Definition
- Functional model →
  - System decomposition (Selection of subsystems based on functional requirements, cohesion, and coupling)
- Object model →
  - Hardware/software mapping
  - Persistent data management
- Dynamic model →
  - Concurrency
  - Global resource handling
  - Software control
- Subsystem Decomposition
  - Boundary conditions

Design activities

For each of these activities, we saw ways you might do it . . .

Heuristics to Identify Subsystems

- Consider the objects and classes in your requirements analysis models.
- Try grouping objects into subsystems by
  - assigning objects in one use case into the same subsystem
  - create a dedicated subsystem for objects used for moving data among subsystems
  - minimizing the number of associations crossing subsystem boundaries
  - ensure all objects in the same subsystem are functionally related

. . .

Design activities

Access control questions

- Does the system need authentication?
- If yes, what is the authentication scheme?
  - User name and password? Access control list
  - Tickets? Capability-based
- What is the user interface for authentication?
- Does the system need a network-wide name server?
- How is a service known to the rest of the system?
  - At runtime? At compile time?
  - By communication port number?
  - By host name?

Design activities

Guidelines for choosing control flow

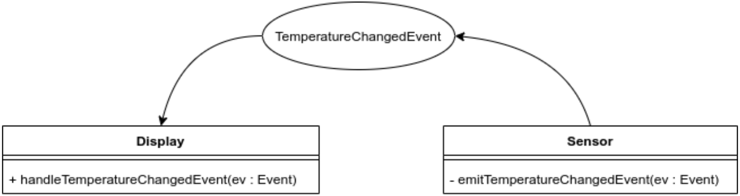
- activities must occur in a fixed order with little time overlaps between activities
  $\rightarrow$ choose procedural control
- activities may occur in different orders, as determined by external requests, but usually one activity at a time
  $\rightarrow$ choose event driven control (+ central controller)
- activities are largely independent and can be time overlapped
  $\rightarrow$ choose threads

Design activities

. . . etc.

Event-driven systems

We looked at one very useful sort of system, *event-driven* systems



And we noted that this sort of system often *decouples* classes from each other (`Sensor` and `Display` don't directly use each other).

Architecture

We looked at different *architectures* – high-level ways of structuring a system. e.g.

- layered architecure
- pipe-and-filter
- blackboard / repository
- client/server

We discussed when different architectures might be appropriate.

Alternative approaches to design

We looked at some alternative techniques for design. e.g.:

- data-oriented design
- use of formal methods

Agile methodologies

We discussed agile methodologies, which *de-emphasize* planned, up-front design, in favour of *incremental* design.