Complexity
Core Challenge: Complexity

• “But when projects do fail for reasons that are primarily technical, the reason is often uncontrolled complexity. ... When a project reaches the point at which no one completely understands the impact that code changes in one area will have on other areas, progress grinds to a halt.”

• “Software's Primary Technical Imperative has to be managing complexity.”
  • McConnell, 5.2

• Architecture is key to managing complexity
  • Architecture provides a guide
  • Good architecture controls interaction
    • which allows considering subsystems independently
Dealing with complexity

• We can’t comprehend the entire system in detail, so we use **information hiding** via
  • Modularisation
  • Abstraction

to be able to effectively deal with complexity
Modularity and abstraction are major aids to understanding

- Using modularisation and abstraction, we have the intellectual leverage to **understand** and (informally) **reason** about systems
- We can apply these concepts at **different levels** to aid our understanding of a system
- In turn understanding enables us to
  - Go about constructing systems
  - Maintain them
  - Extend them
Levels of Design

• Modularity
  • Confines the details
  • So they don’t matter “from outside”
  • Facilitates abstraction

• As we move up levels
  • We lose detail, and
  • Expand scope of what we can understand

• Good design and construction means that the detail can be safely ignored at higher levels

Diagram source: McConell
Structure and dynamic behaviour

Are abstraction and modularisation only good in thinking about static structural things?

- Lets think about run-time, abstraction and modularisation help there too
  - Run-time structure
  - Dynamic behavior
    - Flow of control
    - Invoked computation - Parameters and results
    - Parallelism or no parallelism
    - Dynamically changing structure
    - ....

Thursday, 13 October 16
Example: Components

McConnell, 5.2: Figure 5-3. An example of a system with six subsystems
Example: Complexity via unconstrained communications

McConnell, 5.2: Figure 5-4. An example of what happens with no restrictions on intersubsystem communications
Example: Low coupling is better

McConnell, 5.2: Figure 5-5. With a few communication rules, you can simplify subsystem interactions significantly
• Notice modularity, encapsulation and interfaces at different levels
  • Subsystem
  • Package
  • Object
Design in general, design as an activity
The activity of design and when we do it

- Design is an activity in many fields
  - Eg: Architecture (for buildings), computer architecture to code and test construction

- Characteristics of software design
  - Knowledge in three kinds of domain: Application, technical domain and design domains
  - Requires motivated choices and tradeoffs
  - Knows what to take account of, and what to ignore
  - Multi-faceted and sometimes multi-level
Design is a Wicked Problem

*Horst Rittel and Melvin Webber defined a "wicked" problem as one that could be clearly defined only by solving it, or by solving part of it (1973).* McConnell, 5.1

- Change is a reality
  - Requirements and problem definitions change
    - Exogenously: the external world changes
      - E.g., a regulation is passed during development
    - Endogenously: triggered by the evolving system
      - E.g., people learn that they misunderstood the problem
- Software development must cope
  - Methodologically: E.g., agile methods respond well to change
  - Architecturally: E.g., modularity lets us replace modules
  - Constructionally: E.g., robust test suites support change
Direction of Design

• Top down
  • Start with the **general** problem
  • Break it into **manageable parts**
    • Each part becomes a new problem
    • Decompose further
    • Bottom out with concrete code

• Bottom up
  • Start with a **specific** capability
    • Implement it
    • Repeat until confident enough
      • to think about higher level pieces

---

System Software Architecture

1. Software system
2. Division into subsystems
3. Division into classes with
4. Division into data and
5. Internal routine design

Software construction

Thursday, 13 October 16
Opportunistic focus

• Top down and bottom up aren’t exclusive
  • “Thinking from the top”
    • Focuses our attention on the whole system
  • “Thinking from the bottom”
    • Focuses our attention on concrete issues
• Being able to choose where you focus your attention opportunistically is a great help
• Eg working at the top level, you may wonder will this really work, so you consider realisation at a lower level of detail
  • Will have to rework the top level if doesn’t work at a greater level of detail
Exploring the Design Space

• Wickedness suggests
  • we need to *do* stuff early
  • *build* experimental solutions

• Three common forms:
  • Spikes
  • Prototypes
  • Walking skeletons
Spikes

- Spike
  - Very small program to explore an issue
    - Scope of the problem is small
  - Often intended to determine specific risk
    - Is this technology workable?
  - No expectation of keeping
Prototype

• Can have some small or large scope
• Intended to demonstrate something, rather than ‘just’ find out about technology (a spike)
• Mock ups through working code
• Can be “on paper”!

• Prototypes get thrown away
  • Or are intended to!
Walking Skeleton

- **Small** version of “complete” system
  - “tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.” - Alistair Cockburn

- Walking skeletons are meant to evolve into the software system
- Consider miniwc.py!
Coursework
Q2 and SE1 and CW3

• Q2
  • Another Quiz!

• SE1
  • First essay!
  • Make sure you understand it!
  • Plot an outline

• CW3
  • Catch up!
  • Just CW2 again to get everyone to the next level