Software Engineering 301:
Software Analysis and Design

Design principles

Design principles

- Rules of thumb regarding good things and bad things to do in a design
  - may not *always* apply, but you should have a strong reason for violating them
  - may not be sufficient to select a design
    - when you are sure that two choices will meet your needs, the simpler choice is usually best
    - make sure that you have considered the consequences of all non-functional properties of the system
- Design principles generally arise from typical non-functional properties

Six key principles

- We will look at these:
  - Divide and conquer
  - Aim for high cohesion
  - Aim for low coupling
  - Hide information
  - Keep it simple
  - Anticipate problems
- There are likely others, and these aren’t always considered “standard”

Principle #1: Divide and conquer

- When a “part” is too big or too complex …
  - For one person to understand it all
  - For one person to implement on their own
  - To be easily replaced
- … break it down into smaller pieces

Principle #2: Aim for high cohesion

- Don’t split up details that should stay together
- Don’t lump together details “just because it has to go somewhere”
- Different kinds of cohesion:
  - Providing a single computation, providing a related set of services, interacting with specific data, …
- Parts with poor cohesion are more likely to change, and to be changed incorrectly (because their purpose is unclear)
  - e.g., java.lang.System has fairly low cohesion: collection of random utilities, all static
    - BUT this doesn’t mean it absolutely has to be changed

Principle #3: Aim for low coupling

- Coupling is a description of how interconnected the “parts” in a system are
  - naming another part, making a method call, accessing a field, …
- When parts know too much about each other, the system is harder to change
- Use a more abstract type when it will suffice
- Coupling *cannot* be eliminated altogether

Principle #4: Hide information

- Isolate volatile details as much as possible
  - The fewer parts of a system that know about a fact, the fewer parts that will need to change if that fact changes
- If information changes, system should change in fewer places: errors less likely to occur, effort required likely less
- Related to low coupling
- But:
  - Not all information can be hidden
  - Information cannot be perfectly hidden (changes to it can still propagate beyond its interface)

Principle #5: Keep it simple

- When two designs provide what is needed, choose the simpler design
- A simpler design implies:
  - less work for the implementors and testers, fewer bugs
  - easier to understand
- Avoid the temptation to add features and functionality “just in case”

Principle #6: Anticipate problems

- All code has errors
- The design can make it easier/harder to test
  - E.g., ensure that GUI elements can be executed programmatically
  - E.g., ensure that state can be accessed and set
- Be explicit about interface contracts
  - Pre-conditions, post-conditions, invariants
  - Adhere to the contracts

Next time

- Design patterns