



CS1530: Lecture 10

Software Design

Bill Laboon

The Road Behind

- ❖ Previously, we talked about the tactics of developing in a TDD manner
 - ❖ Red-green-refactor
 - ❖ Developing easily testable methods
 - ❖ Preparing for change

The Road Ahead

- ❖ Now we're going to discuss strategy
 - ❖ Designing Systems
 - ❖ System Architecture
 - ❖ Patterns of Design

Keep In Mind...

- ❖ As we go through, think about how to apply the lessons in Code Complete, especially:
 - ❖ Software design is a "wicked" process. "Wicked": Sometimes the only way to know how to do it is to do it! Example: Tacoma Narrows Bridge
 - ❖ Heuristics. We have yet to come up with a one-size-fits-all approach
 - ❖ Sloppy - you will make mistakes. Agile / Scrum takes this into account!
 - ❖ Trade-offs! Everything is a trade-off, including your architectural pattern.

Software Is Difficult and Complex

- ❖ Accidental and essential difficulties
 - ❖ Essential difficulties: inherent to the problem software is trying to solve
 - ❖ Accidental: Just happen to be part of the problem
- ❖ Examples:
 - ❖ Essential difficulty: Ugh, this machine learning algorithm is hard to understand
 - ❖ Accidental difficulty: Ugh, why are we programming this ML algorithm in Assembly?

Complexity

- ❖ Complexity is almost an essential difficulty in writing software, trick is to manage it
- ❖ Minimize essential complexity (through abstraction, patterns, info hiding, encapsulation, etc)
- ❖ Minimize accidental complexity (through good decisions on software development)

Subsystems

- ❖ One thing that architectural patterns need to do is break a system down into subsystems
- ❖ Should be able to whiteboard these out, high level of abstraction
- ❖ Software that is not easily whiteboardable is a code smell
- ❖ *Code smell* - a hint that something might be wrong. Does not mean that something is DEFINITELY wrong, but just a hint.

Subsystems

- ❖ These subsystems should be cohesive and loosely coupled:
 - ❖ Business rules
 - ❖ User Interface
 - ❖ Database Access

Tactics vs Strategy

- ❖ Think of subsystems as like larger “groups” of objects or classes!
 - ❖ Example: packages in java
- ❖ Remember that classes should have very specific input/output, can have complex innards that are encapsulated / use information hiding
- ❖ Just like objects, there should be loose coupling / high cohesion!

Top-Down Design

- ❖ Top-down
 - ❖ Think of system as a whole first
 - ❖ Fill in general outline of what you need (subsystems)
 - ❖ For each subsystem, fill in with elements
 - ❖ For each element in the subsystem, start filling in more details, etc.

Bottom-Up Design

- ❖ Work on small pieces, making them independent and cohesive
- ❖ Start putting them together to make more complicated patterns and subsystems
- ❖ Put these subsystems together to make large systems

Top-Down vs Bottom-Up Design

- ❖ Which is better? It depends.
- ❖ Top-down: good for well-defined systems where you know the end goal
- ❖ Bottom-up: good for flexible systems where the end goal is unknown or more liable to change

It's Turtles (Abstractions) All The Way Down

- ❖ Wheeler's Law: All problems in computer science can be solved by another level of abstraction.
- ❖ Henney's Corollary "...except for the problem of too many layers of indirection."
- ❖ Fun fact: David Wheeler was the first person in the world to get a PhD in computer science (1951)

Design Heuristics

- ❖ If possible, map from real world or some other tangible concept
- ❖ Example: Super Mario Brothers video game
- ❖ User Interface: controller input, screen output
- ❖ World: contains one instance of Mario, instances of Question Mark block, Brick Block, Goombas, etc.
- ❖ Data Storage: Stored level info, saved games, high scores, etc.

Standardize

- ❖ Use standard..
 - ❖ Terminology
 - ❖ Abstractions and abstraction levels
 - ❖ Design patterns
 - ❖ Architectures

Avoid Leaky Abstractions

- ❖ If you need to know about the innards of a class, this is a fail
- ❖ If you need to know about the innards of a subsystem, this is a fail

Software Architecture

- ❖ The structure of software and how different subsystems interact
- ❖ Fundamental factors
- ❖ Large-scale - absolute smallest detail is objects, functions, or methods, but usually done at even a higher level (subsystems)

Software Architecture

- ❖ Software Architecture Patterns book (from O'Reilly) available online - highly recommended!
- ❖ <http://www.oreilly.com/programming/free/files/software-architecture-patterns.pdf>

Common Architectural Patterns

- ❖ These are not the only ones!
- ❖ There are some really neat but obscure ones that I would love to talk about if we had more time... but sadly you all did not sign up for the five-hours-a-day, five-days-a-week version of this course

Layered Architecture

- ❖ Think of a layer cake, or different geological layers
- ❖ Can only refer to classes below you in abstraction hierarchy
- ❖ You cannot call above your layer of abstraction - the kernel does not call code in Microsoft Word, for instance
- ❖ Each layer is a deeper abstraction level
- ❖ Separation of concerns

Layered Architecture

- ❖ Think of your computer right now - layered abstraction maps very well to it
 - ❖ Application
 - ❖ Operating System tools / libraries
 - ❖ Kernel
 - ❖ Processor / firmware / hardware / etc.

Layered Architecture

- ❖ Very common layout:
- ❖ Presentation Layer - Display data
- ❖ Application Layer - Perform business logic on it and write to data layer
- ❖ Data Layer - Get/save data

Model-View-Controller

- ❖ Model: How the information is stored, business logic
- ❖ View: How the user sees the data
- ❖ Controller: How the model and view communicate
- ❖ Model updates View, View uses Controller, which manipulates Model, loop forever

Model-View-Controller

- ❖ Very common architecture for web apps and other user-facing systems
- ❖ Ruby on Rails, Ember.js, Spring, Django

N-tier

- ❖ “Distributed layers”
- ❖ Tiers are basically physical instantiations of layers
- ❖ Run and separated on actual machines or hardware

N-Tier Example

- ❖ Front-end: the local web client (e.g. Internet Explorer, Chrome, Firefox)
- ❖ Application Server: Ruby on Rails or other web server
- ❖ Data Server: Database or other data store

Client-Server Architecture

- ❖ Special case of n-tier where $n = 2$
- ❖ Client (user interfaces with this)
- ❖ Server (user does not, only goes through client)

Pipeline

- ❖ Data moves forward through processes in a directed manner
- ❖ Assumes you have well-defined input and output, not much muddling (or none) in the middle by the user
- ❖ Input > foo > bar > baz > quux > OUTPUT

Pipeline Example

- ❖ Best example - Unix pipes
- ❖ `ls -l | grep -v "~" | grep "\.java$" |
grep Demo | sed 's/e/3/g'`

Event-Driven Architecture

- ❖ Flow of application driven by events
- ❖ Usually a communications layer and multiple processes "listening" and "talking"
- ❖ Example: fire alarm system, Java Swing framework

Big Ball of Mud

- ❖ Most popular kind of architecture!
- ❖ Usually starts out as another kind of architecture, slowly morphs
- ❖ Add something here! Ugly hacks! Etc.
- ❖ Lots of spaghetti code, leaky abstractions, held together with duct tape and gum
- ❖ <http://www.laputan.org/mud/>

How To Avoid The Big Ball of Mud?

- ❖ Avoid integrating throwaway / prototype code
- ❖ Strict code reviews
- ❖ Modularize ruthlessly
- ❖ Figure out what architecture you want to use and use it
 - ❖ Can be difficult in an agile environment!
 - ❖ Much easier with BDUF