CS1530, LECTURE 8

SOFTWARE DESIGN AND ARCHITECTURE

SOFTWARE DESIGN

- In this phase, focus moves from “what” of requirements and the SRS to the “how” of the actual design
  - (NB - the “what” and the “how” often bleed into each other!)
- That is, given an SRS, how do we plan out how our software will be constructed?
- Design includes - basic architecture and patterns to use, layout of system, classes/packages/modules, interfaces between subsystems, etc.

PHASES OF DESIGN

- **Architectural design phase** - A high-level overview of the system, including detailing main components, their relationships, and relationships to the outside world. The requirements (functional attributes and quality attributes) as a whole guide this, but there is usually not a 1:1 correspondence between requirements and architectural design decisions.

- **Detailed design phase** - With guidance from architecture, components are detailed (usually down to class level, perhaps to method). There is a direct mapping here to ensure all requirements are met (although note that some aspects of the system will not be covered by individual requirements (may be system-wide).

PHASES OF DESIGN

- In traditional development methodologies, detailed design is done up-front. Changes after the initial design are (relatively) rare and discouraged.
    - The more heavyweight the process, the more detailed the detailed design
- In Agile methodologies, very little detailed design. Individual developers often do this during, and as part of, the development process.

OBJECT-ORIENTED DESIGN

- Designing a system in terms of its classes, objects, and their relationships and interactions with each other
- The most popular, but far from the only orientation!
  - **Structured design** (systems designed via the structures of data passed in and out)
  - **Algorithmic / functional design** (systems designed by determining algorithms/functions used)
  - **Transformational design** (systems designed by in-depth specification of the problem itself, through which the solution is manifest)

OO AND ITS DISCONTENTS

- *"Object oriented programs are offered as alternatives to correct ones."* - Edsger Dijkstra
- *"Object-oriented programming offers a sustainable way to write spaghetti code."* - Paul Graham
- *"Object Oriented Programming is an expensive disaster which must end."* -Lawrence Krubner

ARCHITECTURAL DESIGN

DONEC QUIS NUNC

- Software architecture - the basic and underlying structure of the software system
- This is a high-level abstraction of the system
- Every system has an architecture, even if it is not documented or planned!
- Multiple architectures can exist in one system

UNDERSTANDING AN ARCHITECTURE

- Software can be understood in different ways via different viewpoints
    - **Logical view** - the classes of the system and their relationships
    - **Process view** - the run-time components and how they communicate and otherwise interact with each other
    - **Subsytem decomposition view** - the modules and subsystems and how they import / export data (NB: this is static, as opposed to process view which is dynamic)
    - **Physical architecture view** - how the system is laid out in the physical (not virtual) world (e.g. computers, wires, servers, etc.)

META-ARCHITECTURAL KNOWLEDGE

- **Architectural styles or patterns** - basic “strategic” outlines of how software should be constructed (e.g. pipe-and-filter, event-driven, client-server, layered, MVC, etc.)
- **Architectural tactics** - Smaller, “tactical” methods of solving similar problems
- **Reference architectures** - Complete architectures that can be used as templates for a class of systems (include an architectural style, different tactics used in it, and specific pieces of functionality and problems solved)

ARCHITECTURAL STYLE: PIPE-AND-FILTER

- Data moves forward (never backwards) through a series of pipes, and get modified/removed until you have output
- input -> (FILTER SOME DATA OUT) -> (MODIFY DATA) -> output
- Assumes well-defined input and output
- Assumes little interaction after initial input before output
- Examples: UNIX command line, map/reduce, data (and audio/ video) compression

ARCHITECTURAL STYLE: EVENT-DRIVEN

- Flow of application is driven by events, which can occur at any time
- Usually there is a communications layer and other elements which can “talk” to each other
- Good for when there is repeated input/output
- Examples: GUIs (Java Swing framework), video games

ARCHITECTURAL STYLE: CLIENT-SERVER

- A style where there is a clear difference between servers and clients of those servers, which are physically distinct. Servers store data and business logic, and clients interact with them. Usually there are many more clients than servers.
- Good for when interactions are computationally cheap and you want a centralized location for data storage and business decisions
- Examples: (simple) web sites, ATM networks, MMORPGs

ARCHITECTURAL STYLE: DISTRIBUTED

- Every system acts as a full node (compare to client-server, where only some systems, servers, are “full” nodes) and can interact with any other individual node. All nodes are in this sense “equal”.
- Note: not decentralized! Decentralized means that there are multiple nodes that interact with each other, but some nodes cannot connect directly with each other.
- Good for reliability, although you often pay for this in terms of performance
- Examples: Bittorrent, (core) Bitcoin protocol, Ethernet

ARCHITECTURAL STYLE: LAYERED

- Components are grouped into layers (like a layer cake) and the components only communicate with other components in their layer, or one up or one down
- Example: Presentation Layer (displays data), Application Layer (business logic), Persistence Layer (stores/retrieves data in database)
  - There is no direct interaction between the Presentation and the Persistence layers - interactions go through Application layer
- Examples: JVM, many desktop applications

ARCHITECTURAL STYLE: MVC

- Model - how the data is stored, View - how data is viewed, Controller - how these two communicate
- Allows different ways of viewing similar data
- Views interact with Controller, Controller interacts with Model (similar to Layered architecture)
- Examples: Ruby on Rails, Django, Phoenix web framework

… and the most popular

architecture of them all….

ARCHITECTURAL STYLE: BIG BALL OF MUD

- .. a “haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.“
- May start out as a more well-defined architecture, but through piecemeal replacement, quick-and-dirty hacks, throwaway code being committed, different people with different ideas, problems being swept under the rug, etc., the program becomes an unmaintainable, incomprehensible mass
- Try to avoid this style
- http://www.laputan.org/mud/

ARCHITECTURAL TACTICS

- Lower level methods to solve particular issues
- Example: heartbeat vs ping
- Example: redundancy
- Example: pub/sub (publish and subscribe)

REFERENCE ARCHITECTURE

- "A full-fledged architecture" which can serve as a template for a class of software products
- "Convention over configuration" - By using a reference architecture itself, you are standing on the shoulders of giants (or at least people who have already worked on these kinds of projects)
- By studying a reference architecure, you can understand how to make your own software engineering decisions better, and know when to configure those conventions

EXAMPLE: REST

- REpresentational State Transfer - a reference architecture used often for communication among decentralized web applications (a common issue!)
- Overall client-server design style for the system as a whole
- Constraint: all communication done via network message-passing (allows easy horizontal scalability)
- Individual components use a layered design style
- Uniform interface between components (tradeoff: increases simplicity, but decreases efficiency and versatility)
    - HATEOAS (Hypermedia As The Engine Of Application State) - Using standard HTTP message type (GET, POST, DELETE, etc.) for communication along with descriptive URLs, the entire system can be discovered and interacted with

DETAILED DESIGN

- After determining the general architecture, more detailed work needs to take place
- This is usually down to the class level, sometimes the method level.
- Rarely would one talk about anything below the method level at this point (that would be implementation)

FUNCTIONAL DECOMPOSITION

- A top-down (as opposed to bottom-up) way of developing the architectural design
- For each module or function, decompose it into its own pieces which can be worked on
- This may be by particular actions or functionality or by entity

FUNCTIONAL DECOMPOSITION EXAMPLE (ATM)

- Deposit
    - Cash, Check
- Withdraw
    - Cash
- View Balances
    - Checking, Savings, All Accounts
- Cancel