Lecture 9: Software Design

COSI 120b, Principles of Software Engineering
Now What?

- We have a set of functional and non-functional requirements
- We have managed to turn them into a specification document
- We have had the customer sign off on the requirements
- What’s the next phase?
Software Design

• Turn the requirements into software representations
  – Data structures
  – Class Diagrams
  – Software Architecture
  – Algorithms
  – Interfaces
  – Hardware Requirements
Software Design

• At the end of this stage the engineer should:
  – Have a clear picture of the software parts of the system
  – Have an understanding of their interactions
  – Understand the distribution of software to the hardware and the hardware requirements
  – Know the algorithms and their characteristics
Software Design

- Component-Level Design
- Interface Design
- Architectural Design
- Data / Class Model
Overview

• How do we ensure good software design quality?
• How do we go from requirements to a software design?
• How do we represent software design?
Design Quality

• A design must:
  – Implement all the requirements in the specification
  – Be readable and understandable for the team that is implementing the system
  – Complete picture of the software addressing each domain
    • Data
    • Functional
    • Behavioral
Design Quality

• Quality Guidelines
  – Recognizable styles or patterns
    • Do not reinvent notation
    • Design patterns
    • UML
  – Modular design
    • Ensure that parts of the implementation can be parallelized
  – Distinct, unambiguous representation
Design Quality

• More Guidelines
  – Appropriate data structures and algorithms
    • Fit the design
  – Independent functional characteristics for the components
  – Interfaces that limit interaction and reduce complexity
  – Repeatable Methodology
    • You may need to update your design later
  – Effective Notation
Design Quality

• Once we have a design, can we talk about the quality of the system?
• Quality Attributes (HPs \textit{FURPS})
  – Functionality
    • Feature set
    • Capabilities
    • Security
  – Usability
    • Human Factors
    • Aesthetics
    • Consistency
    • Documentation
Design Quality

• More Quality Attributes
  – Reliability
    • Frequency and Severity of failure
    • Recovery from failure
  – Performance
    • Processing speed
    • Resource Consumption
    • Throughput
    • Efficiency
  – Supportability
    • Can the system be extended?
    • Can it be maintained?
    • Adaptability, serviceability, extensibility
Requirements to Design

• So we know what makes a good design
• How do we go from the requirements to the design itself?
• What are the parts of the design?
• What is the representation of the design?
Abstraction

• Make the solution sufficiently general
• Data abstraction
  – A door
    • Contains a bunch of attributes
• Procedural abstraction
  – Methods to manipulate the door
• Talk about what the system should do at a conceptual level
  – Avoid talking about code
Architecture

• Consists of
  – Components
  – Their interaction
  – Structure of the data used by the components in their interaction

• The major components of the system and their interactions

• Represented through
  – Structural models
  – Framework models
  – Process models
Patterns

• A pattern is a well-known solution to a recurring problem
  – Solves a particular design problem in a way that has
    • Known consequences
    • Known implementations
    • Known limitations
  – Aids in reusability
    • Not only code, but conceptual reusability
Modularity

- Dividing software into separately named, addressable, and implementable components
  - Allowing parallel development
  - Allowing reuse of existing code modules
- Balancing cost of the software vs cost of integration
  - Integration is not free
  - Small modules may reduce complexity
Information Hiding

• How modules communicate with each other
  – The set of information contained within each module
  – The subset of that information that is shared between modules

• Reduces who can access information, thereby reducing the things that can go wrong
Functional Independence

- Modules that have a limited purpose and scope
  - Limited interaction with other modules
- Cohesion
  - Single task
  - Requiring little interaction with irrelevant parts of the system
- Coupling
  - Measure of interconnections with other parts of the system
- Aim for highest cohesion and lowest coupling
Refinement

• Movement from model to code
  – Starting to a high level of abstraction and ending at precise code statements
• Is this possible?
• Does this always make sense?
Refactoring

• Changing a software system so that the behavior of the code does not change, yet its internal structure does change
  – Why do we do this?
• How does this effect cohesion?
• Coupling?
• Complexity in general?
Design Classes

• Representations of different levels of abstraction
  – User Interface classes
    • Define all interaction with the user
    • Also defines the representation system / user metaphor
  – Business domain classes
    • Attributes and services that are required to implement the identified business functionality
Design Classes

• Process classes
  – Business abstractions necessary to manage the business domain

• Persistent classes
  – Data store
  – Databases
  – Persistent information beyond individual run-times of the software

• System classes
  – System management
  – Network, File I/O, etc
Design Classes

- Well-formed
  - How do we know these are the right classes?
  - Complete and Sufficient
    - Encapsulates all attributes and methods
  - Primitiveness
    - Perform one function
- High Cohesion
  - Well focused
- Low coupling
  - Independent
Representation of Software Design

• How do we represent each part of the system
  – Everyone on the team understands the representation
  – Unambiguous

• Universal Modeling Language
  – A “standard”
  – Not necessarily better than any other representation, but well documented and well known
Architecture Design

- The representation of the pieces of the system
  - How modules interact
    - Identify the modules
    - Identify their interaction
  - Refinement to class diagrams
  - Refinement to sequence diagrams
Architecture Design
Architecture Design
Interface Design

• Specification of the interactions into the architecture
  – How the user interacts
  – How other developers interact
  – How the system interacts with other systems
  – How other components interact
Component Design

• Detailed description of the individual modules
  – Detailed sequence and class diagrams
  – Psuedo-code
  – Method-level refinement
Deployment Design

• What are the parts of the system to be deployed in the physical computing environment
  – What components live on what computers in a distributed / enterprise system
  – How does that effect resources?
Conclusions

• Design helps you guide the implementation

• Design needs to follow from the requirements, and fully reflect the requirements

• Design needs to be agile enough to change as the requirements change