Overview of Design

15-413:
Introduction to Software Engineering

Jonathan Aldrich

Some ideas from David Notkin's CSE 503 class

Announcements

- Iteration 1 report due Monday
- Assignment 6 posted
  - Build 3-4 requirements models
  - Due Wednesday
- Iteration 2 plan due Friday

Announcements

- Stories and Tasks
  - To get a story to work may require several conceptual underlying tasks
    - Build a framework, write a UI, …
  - Can be useful to divide each story into tasks
    - Stories are for planning, tasks for implementation
    - Story should be completely done in an iteration; but different tasks can be done by different people
- Testing UIs
  - HttpUnit may be useful (Team 5)
  - UI with code underneath
    - Design hooks that allow you to drive the app without the UI for testing
  - Pure UI
    - Write a script for exercizing the UI
    - Execute the script by hand
    - Run every iteration & whenever relevant code changes

Announcements

- Testing Legacy Code
  - If you are changing or enhancing legacy code, write unit tests for the portion that you are going to change
- Functional Tests
  - Need one or more for each story
    - Written before the story is implemented
  - Automated if possible
    - Use manual script if not
  - Will ask for these in next project review

Software Complexity

- Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one…In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound."
  —Brooks, 1986

Benefits of Decomposition

- Easier to manage
- Modularity – solve a problem and use in other places
- Facilitates testing
- Easier to understand context & functionality of small parts
- Design is easier because problem is smaller
- Delegation / division of work

Benefits of Decomposition

- Smaller problem size
- Understanding
- Verification
- Reuse
- Evolvability
- Fault isolation
- Independent development

Criteria for Decomposition

- Look for common reusable components
- Maximize independence between modules – can alter parts independently, testing, etc.
- Each part solves one part of the problem – correspondence between problem & solution
- Performance

Criteria for Decomposition

- Conceptual integrity
- Reusable parts
- Performance
- Hide information likely to change

- Real world: need a balance of these

Software Change

- …accept the fact of change as a way of life, rather than an untoward and annoying exception.
  —Brooks, 1974
- Software that does not change becomes useless over time.
  —Belady and Lehman

- For successful software projects, most of the cost is spent evolving the system, not in initial development
  - Therefore, reducing the cost of change is one of the most important principles of software design

Information Hiding

Derived from definition by Edward Berard

- Decide what design decisions are likely to change and which are likely to be stable
- Put each design decision likely to change into a module
- Assign each module an interface that hides the decision likely to change, and exposes only stable design decisions
- Ensure that the clients of a module depend only on the stable interface, not the implementation

- Benefit: if you correctly predict what may change, and hide information properly, then each change will only affect one module
  - That's a big if…do you believe it?

Abstraction

Derived from definition by Edward Berard

- Noun: A representation of some object that focuses on more important information and leaving out less important information
  - The details (less important information) may be specified separately from the abstraction
- Verb: To come up with such an abstraction

- Distinct from information hiding
  - You're leaving out "less important" information, vs. information likely to change

Encapsulation

Derived from definition by Edward Berard

- Noun: a package or enclosure that holds one or more items
- Verb: to enclose one or more items in a container

- SE: a *language mechanism* for ensuring that clients of a module do not depend on its implementation
  - e.g. Java's public/private
- A matter of degree
  - C: must define structs used by clients
  - Java: hides all syntactic dependences
    - But semantic dependencies may remain

- Could I hide information in a language without encapsulation mechanisms?

Hiding design decisions

Decision	Mechanism
• Data representation	ADT/class
• I/O or comm proto	prototcol library
• GUI	GUI interface
• Algorithm	function

Hiding design decisions

- Algorithms – procedure
- Data representation – abstract data type
- Platform – virtual machine, hardware abstraction layer
- Input/output data format – I/O library
- User interface – model-view pattern

What is an Interface?

- Function signatures?
- Performance?
- Ordering of function calls?
- Resource use?
- Locking policies?

- Conceptually, an interface is everything clients are allowed to depend on
  - May not be expressible in your favorite programming language

Does an object-oriented design mean information hiding?

Does an object-oriented design mean information hiding?

- Objects hide data representation and the implementation of data operations
  - Assuming that fields are private and internal data structures stay internal
- Data representation is often likely to change—*but other things can change as well!*
- Need to think explicitly about what may change in order to hide it effectively

Cohesion

- The number of dependences within a module
- High cohesion is good
  - Changes are likely to be local to a module
  - Easier to understand a module in isolation

Coupling

- The number of dependences between modules
- High coupling causes problems
  - Change propagation
  - Difficulty understanding
  - Difficult reuse
- Coupling increases over time
  - I need to use that function over there…

Correspondence

- How well the design of the code matches the requirements
- If each requirement is implemented by a separate module, then a change in a requirement should only require changes to one module
  - Hard to achieve in practice
  - OO approaches design code after a model of the world
    - This helps, but some requirements crosscut the structure of the world as well!
- Separation of Concerns
  - Generalizes correspondence to "concerns" that may be implementation issues, not just requirements

Information Hiding Premises

- We can effectively anticipate changes
- Changing an implementation is the best change, since its isolated
- The semantics of a module must remain unchanged when its implementation is replaced
- One implementation can satisfy multiple clients

- Are these always true?