What is Design?

What is Design? (My answer)

- Reasoned choice
  - Determining alternatives
  - Predicting consequences
  - Balancing tradeoffs

Design is not a phase!
Design Everything

- Design is not a phase
- Everything is design
  - Design the product
    - Its specification, structure, and implementation
  - Design the process
  - Design the organization

Designing Software Structure

- Key Goals
  - Design for change
  - Design to schedule
  - Design for risk control
- Approach
  - Information hiding

Design for Change

Why does software change?

- External environment (e.g., Lira -> Euro)
- User audience - new requirements
- Technical environment
  - XP to Vista; static html to dynamic to ajax; web-enabled office software; ...
In recent months, the department has used a computerized test that was known to have some answers that conflicted with state law or the state-issued study guide. [...] Some 97,414 tests were administered during the three months, with 36,391 failures reported.

Some computers used to administer the tests are still marking correct answers as wrong, said department spokeswoman Suzanne Taylor. The problem lies in getting updated software for the computers’ testing program, she said.

Information Hiding for Change

• Identify design secrets
  – A secret is a potential change that nothing else should depend on
  – A module or subsystem hides a secret
    • Localizes an anticipated change

Hiding and Abstraction

• What is “abstraction”
  – or “an abstraction”
  – or “an abstract interface”?
• Hint:
  – “Abstract” does not mean “vague”
  – “Abstract” does not mean “mathematical”

How hard can this be?

• How can it happen that Washington has to wait for updated test software with correct answers?
• How would you design that system? Would your design be vulnerable to this problem?
Abstract

- X is an abstraction if it can be realized (implemented) in multiple, different ways
  - X is “more abstract than” Y if implementations(X) ⊇ implementations(Y)
- An interface can “abstract over” a set of possible implementations
  - the variation is the design secret

Hiding is Not Free

- Hiding means pretending not to know
  - Not taking advantage of information that is “hidden” in other modules or subsystems
  - Not using the faster special case (optimizing)
  - Coding for all cases, not just those that can actually occur
- So we hide some things and reveal others

Abstract Interface Examples

<table>
<thead>
<tr>
<th>Interface</th>
<th>Provides Abstract Service</th>
<th>Abstracts over (secret)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>Reliable communication stream</td>
<td>Routing, transport (ethernet, PPP, ...)</td>
</tr>
<tr>
<td>IDE</td>
<td>Addressable block storage</td>
<td>Storage media, device characteristics</td>
</tr>
<tr>
<td>SQL</td>
<td>Relational database manipulation</td>
<td>Storage structure, concurrency control</td>
</tr>
<tr>
<td>Java Swing</td>
<td>GUI widgets &amp; interaction</td>
<td>Window system, graphics platform (Win32, Cocoa, X, ...)</td>
</tr>
</tbody>
</table>

What to Hide?

- Distinguish “likely changes” from “fundamental assumptions”
  - A “fundamental assumption” can be spread throughout system
  - A “likely change” should be isolated (hidden)
- Example
  - SQL databases hide many details
    - index structure, locking or versioning protocol, ...
  - But relational (vs. OODB, etc.) is a “fundamental assumption”
Change Time Scale

- Minutes? Hours? Weeks? Years?
  - Binding mechanism may be different in each case, from table-driven programming to modular organization
- Binding time is a design tradeoff
  - Usually, “more dynamic” is more expensive in complexity and/or performance
    - but not always (e.g., video games, Washington driver test)

Designing Software Structure

- Key Goals
  - Design for change
    - Design to schedule
  - Design for risk control
- Approach
  - Information hiding

Information Hiding and Design to Schedule

- Fixed schedule, flexible feature set
  - Expand or contract as time allows
  - Deliver incremental releases
- Three considerations
  - Independence of optional features
  - Parallel development
  - Staging and replacement

Dependence

- If module A depends on module B
  - B must be developed first
  - A cannot be delivered unless B is delivered
  - Changes to B may require changes to A
- We would like to break (some of) these dependencies
Dependence is Semantic, not (only) syntactic ...

- Dependence may not equal “calls”
- Dependence may not equal “reads from”
- Dependence may not equal “import”

Breaking Dependence with an Interface

Examples:
- Java Interface
- Device Driver
- Protocol
- Virtual Machine API

Expand and Contract

- Minimize dependence on optional features
  - Hide presence/absence to the extent possible
- Dependence should be consistent with build order
  - Develop as a series of releases
Design to Control Risk

- Be explicit: What are you worried about?
  - You should have a “risk plan,” including technical and non-technical risks
- Risk implies potential change
  - So the design-for-change techniques apply: Information hiding, abstraction

Object-Oriented

- Myth: Object-oriented = information hiding
- Reality:
  - OO is based partly on information hiding principles
  - OO is often good for hiding data structures
  - OO may not help hide other design secrets
    • and can even get in the way, sometimes

Language Support for Information Hiding

- Visibility control
  - Private classes, private methods, packages ...
  - Helpful for information hiding
- “Abstract” interfaces (abstract classes, interfaces, ...)
  - Usually: Encapsulate data structure
  - Possibly allow multiple implementations
    • but only if well designed

Where Language Support Fails

- Hiding memory management policy
  - especially problematic in C++; less in Java
- Hiding concurrency policy
  - although Java has made some progress here
- Factoring information from control
  - what Washington driver test needed
- Large-scale structure
  - Packages are (just) a start
Approaching Design

- Start with explicit consideration of risks, schedule, and likely changes
- Then consider gross organization, with explicit design secrets
- Then (and only then) choose an appropriate design approach
  - OO design, data flow, layered, ...
  - Probably more than one, at different levels

Applied to 07F project ...

- Overall project:
  - Risk identification: Java graphics might be unsuitable for map display/interaction
    - Tracking mouse hits too slow or complex?
    - Jumping mouse unworkable?
  - Separation of concerns / info hiding:
    - Relative independence (?) of GIS style mapping from display & interaction
    - But display & interaction is a big chunk - all interaction will need reimplementing in Java

Applied to 7f Project (cont)

- Design decision:
  Interpretation/translation approach
    - Directly interpret XML + styles ?
    - Translate XML + Styles to Java code?
    - Translate to data and interpret that?
- A “binding time” issue
  - Very common design dimension, often trading performance and flexibility
  - Different tradeoffs from drivers’ test problem

Product Design & Process
What Else Can You Factor? (and why?)

Intermediate form or ??

Objectives: Parallel work, independent choice, precise interface definition