Detailed Design and Implementation

Phase 4 & 5

Detailed design

- During detaild design we try to refining and expanding the system design to the extent that the design is sufficiently complete to begin implementation
  - But as we know we have already started implementing
  - May cause architecture revision

- **Mandatory** design issue is **testability**

Remainder: Design mismatches

- All the way we have been using the OO paradigm
- **All software is not OO**...!
  - 3D graphics is built on an rendering pipeline
  - Relational databases are not OO!
  - The web is not OO
  - ...
- Have to incorporate different paradigms in one application
  - ...???...
- Optional **Not covered in course!**
  - Possible assistants can help...?

Documenting Detailed design

- No separate document
- Code and tests are the ultimate documentation
  - So please, write readable code
  - Adhere to standard style
  - Good naming!
  - Not to dense, use space, empty lines
- Comments
  - Class comment: What is this? Responsibility?
  - Method comments: -"-
    - What is happening!! (not how it's done)
  - Variables: -"-
  - Don't comment the obvious!

```
x = x + 1;   // Add one to x
```

Software Testability

- The degree to which a software artifact (i.e. a software system, software module) supports testing in a given test context

Software Testability in Course

- **Must** be possible to test classes (modules, subsystems)
  - **Must** use JUnit **4** (not 3) framework for unit tests
- Test **must** be automated (i.e. no human interaction, except for one mouse-click to run...)
  - Corollary: **Can't have** dependencies of GUI
- Dependencies
  - Few => Easy to test
  - Many => Cumbersome (complicated setup)
- We test implementations not interfaces
  - Possible need to cast in tests

Candidates for Testing

- Subsystems should be tested
    - Again: Testing implementation (not interface)
- Fat model classes: Yes
- Anemic model classes : Seldom
- Control classes: Yes
- public methods: Yes
    - void methods: Test side effects (state changed)
        - possible create methods to inspect state (not part if interface)
        - ..or use reflection (don't change test code)
- private methods, no (if in need; possible bad design)
    - If really need, use reflection (don't change test code)
- GUI: Yes,
    - ... more later

Implementing Testing (1)

- Create separate source folder "test" (put in Subversion)
- User same package hierarchy in test folder (as in src)
  - Possible to test
    - package private classes
    - protected methods
- Again: Testing should be automated
  - **No** System.out.println(),...need humans
    - Possible during development of test
  - Possible outcome: Pass or Fail
  - Possible exception, parts of GUI (dialogs, popups)
- Normally one JUnit test/class with...
  - ...many test methods (**not** one huge method)

Implementing Testing (2)

- Long names for test methods
  - Long name: public void selectAndCheckAllItemsForAdmin() {...}
- Fixtures: Setup for tests
  - Must have known state, fresh data, ...
    - @BeforeClass, run once before all test methods
    - @Before, run before every thest method
  - @Test, a test method to run
  - @Test(expected = IllegalArgumentException.class), test method with expected exception

Testability and Code Coverage

- Test should exercise major parts of (all) code
- How do we know?
- Use a coverage tool
  - In Eclipse EclEmma installed
  - Marks code as
    - run (green)
    - run partly (yellow)
    - not run (red)
- Technique: Run JUnit test them coverage

Dependencies

- Have UML diagrams for ocular inspection, but are we shore?
- Use a tool
    - JDepend checks for cyclic dependencies (and more)

Testability MoPro

- We'll inspect
  - monopoly-3.2.ep/test (anemic)
    - Note: Simple fixture
    - TestCore
    - TestControl
    - TestDialogs
    - Interpreter (not finished just a sketch)
  - monopoly-3.2-DDD/test (fat)
    - Note: More complicated fixture
    - TestPlayer
    - TestSinglePlayer
- Do some coverage
- Check with JDepend

Detailed Design Implementation Overview

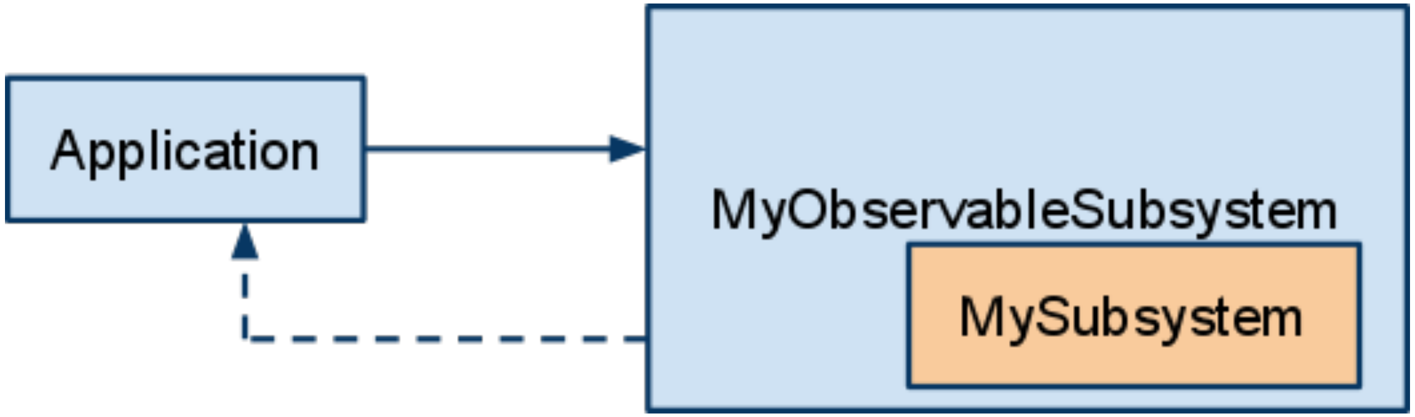
- Subsystem implementation
    - We have the interfaces!
- Event model implementation
- GUI implementation
- MVC implementation
- Entry/Exit
- How to wire together
- Exception handling
- Lookups
- Resources
- ... know you design pattern!

Subsystem Implementation 1

- "In house" (code yourself) or find existing implementation
    - In house implementation
        - Standard: Facade + Factory method (DP)
        - Interface to subsystem delivered by factory method
        - See monopoly-3.2.ep-DDD/io/FileReader
    - Existing
        - ... find, possible wrap using Adapter DP to match our interface
- Application always references the interface of the subsystem
    - Exception: The class with the Factory method

Subsystem Implementation 2

- Subsystem has single responsibility
- Add features by wrapping (decorator pattern)
  - Make it a singleton
  - Make it observable
  - ...



Subsystem to Search For

- Typically you don't implement subsystems for
  - Graphics
  - Sound
  - Data handling, XML, ...
  - Networking
  - ... find somewhere!
- Always look for high level
  - Network: Sockets, NO!
    - XML-RPC, RMI, ... probably better
  - Resources: Low level file handling, NO!
    - Resourcbundles, java.xml.*, ...better for some tasks

Event Model Implementation

- In house implementation
  - Standard: Observer pattern, more to come...
  - Advanced: Implement "event buses" or "messaging"
    - Messaging for asynchronous events (message queues)
- Using frameworks
  - Pro
    - Much work done
    - Will probably get at (very) good design (few dependencies)
  - Cons
    - Time to learn
    - Possible surprises (bugs)
- Frameworks for events
  - WELD (= CDI, Java Context and Dependency Injection)
  - Quick look at testweld.ep (on course page)

GUI

- Primitive GUI
  - Monolithic (one huge frame),
  - Hard coded data; Positions (234,15), icons, colors, texts, ...
- Advanced GUI
  - Modular, composed of panels in layers
  - Uses Layout managers
  - Data externalized
  - Possible I18N (internationalization)
- GUI Testing
  - Automation possible but optional (no tool in course)
  - Possible to create semi-automated JUnit tests
    - Example: monopoly-3.2.ep/test/.../TestDialogs

Event Handling and Updates in GUI

- State Changes and Events possible updates GUI
  - Simple: GUI is Observer
  - If complicated GUI add other "handlers" as observers
    - Example MoPro: DialogHandler class (non GUI, invisible) an observer, shows dialogs in response to events/state changes
- Code to update GUI resides in GUI
  - In listener
    - before call to control/model
    - after call to control/model
  - In observer-callback method

GUI implementation techniques

- Hand code... tedious, probably bad idea ...
- Draw Swing components
    - NetBeans/Matisse
    - Eclipse/Jigloo
    - ...
- Build GUI from XML (a simple structured text file format)
    - SwiXML
    - Beryl XML (incl. a GUI builder)
    - ...
- In any way
    - Separate out the GUI **construction code** (JButton b = new JButton()), from event handling/listeners
    - Done automatically in SwiXML

Concurrence in GUI

- Swing single threaded
- All updates of GUI in event dispatch thread (EDT)
  - Example: Incoming network (other thread) must handle over to EDT
    - Use SwingUtilities.invokeLater(...)
- Time consuming method calls will block GUI
  - Use SwingWorker to run tasks in separate thread
- Also possible
  - Use Timer and TimerTask to run periodically in background

Observer implementation techniques

- Use Java interface java.util.Observer and subclass java.util.Observable,
  - Uses implementation inheritance, primitive callback
  - ...avoid
- Classes from java.beans.* package; PropertyChangeEvent, PropertyChangeSupport and PropertyChangeListener
  - More fully fledged
  - Demo on course page
- As previously demonstrated CDI/WELD

Control Implementation Techniques

- Controls often use Command Pattern
  - Interface with single "execute"-method
  - Parameter passing through constructor
  - Example monopoly-3.2.ep/ctrl
  - Possible to store commands (undo!)
- Often a Factory to produce controls
  - Possible pure static class
  - Example CF.java/MustSellDialog.java

Application Entry/Exit Points

- Entry: Standard is (use)
    - A "Main"-class with
        - public static void main(String[] args){

          ...

- Possible: Exit-class
    - Handling cleaning up
    - Calls System.exit(0)

Wiring It Togheter

- Where and when to wire together the application?
  - Static wiring; fixed references
  - Dynamic wiring; changing references
- Ad-hoc (non general)
  - A creates B creates C, ...
    - Creation all over!
    - Dependencies..?!
- Centralized creation
  - If simple, create/wire in Main class
  - Else, Builder pattern or similar
- Use a framework
  - A framework can "inject" objects into other objects
  - Very loose coupling
  - Quick look at testweld.ep, testguice.ep (on course page)

Wiring In MoPro

- Builder class for model, GameBuilder
    - Model built outside of model
- Builder class for view, GUIBuilder
    - Uses model builder to construct static parts of GUI
- A look at monopoly-3.2-DDD.ep

Exception handling

- Try to find a general pattern
- Possible ExceptionHandler class
  - Very convenient to let all Exceptions pass through one known location
  - Possible to decide later how to handle different exceptions
  - Possible add logging
- If long chain of method calls, possible use "exception tunneling"
  - Wrap checked exception in runtime exception

Lookups

- Very common need for lookup
    - Singletons
    - Resource Locator
        - Singleton with methods to locate objects
        - Read only
    - Global maps
        - Enum as keys (no misspelling)
        - Read only

Resources

- How to find/organize?
  - Standard: Use Resource Bundles
    - java.util.ResourceBundle
    - A map as a text file. Automatically read and converted to Java object
  - ResourceBundle demo on coures page (FAQ page)

Summary

- We got an idea (Monopoly)
- We gathered the requirements
  - Scope, use cases, functional/non-functional, GUI
- From the requirements we analyzed and built a model
  - The analysis (domain) model (class diagram)
  - A dynamic model of a high priority use case (sequence diagram)
- From the analysis we implemented a running use case
- During system design we created the system architecture
  - Spaces, event bus, interfaces, subsystems, file format,..
- We did detailed design, implemented and tested
  - Respecting the previously created design
- Final: monopoly-3.2

Final Prototype

- Inspection of monopoly-3.2-DDD-ep (final prototype)
  - Visitor design patterns for spaces
- A demo run of final protoype

Hmm...

```
15:33
com.yahoo.go.app.YGo
Null Pointer
java/lang/NullPointer
Exception
                    Quit
```

- Murphy's law
  - Anything that can go wrong, will go wrong **...**
- Finagle's Corollary
  - ... at the worst possible moment

- Ontological indifference law
  - The universe is not indifferent to intelligence, it is actively hostile to it