The n-1 next iterations

Running through all phases ...

Iterating MP (1)

- Have one running use case: move
- Now we add the following UC's
  - EndTurn (i.e. switch player)
  - Buy (something for now)
  - Sell (-"-)
- **New design choice:** Piece.isMovable()
  - Can't move if not movable (i.e. user can click in GUI nothing will happen). Only choice for user is to click next player
  - Control class switches: movable = false/true
  - Add a test for movable
- A quick demo run of version MP 0.2

Iterating MP (2)

- Have 4 running use case
- Continue to add more UC's (or make existing more complete)
  - Income, expense, passing GO, ...
- Then !! ... Pick card.... Oooops!
- Design horror!!!!!
  - A card can move the piece! And possible a new card can move it again... the game running by itself!?!? Or?
  - A card can affect the piece and the player or all players (not just the actual player)
- Will this break the design ...??
  - A quick demo run of version MP 0.4

Adding a GUI to MP

- Having quite a few working use cases
- Time to add a GUI
  - Should have a preliminary one
- Design choice
  - We'll use the top-level class Monopoly as the control
  - Possible to move much of the command line version code to the class
  - GUI will call methods on Monopoly (should create interface in between)
- Also, ... should use MVC, how to?

GUI Technicalities

- State changes in model and other event possible updates GUI
- Code to update GUI resides in GUI
    - In listener
        - before call to control
        - after call to control
    - In observer-callback method
- Swing single threaded
    - Possible need SwingUtilities.invokeLater(...) or invokeAndWait(), (blocking)
- Time consuming method calls will block GUI
    - Use SwingWorker to run tasks in separate thread
    - Use Timer and TimerTask to run periodically in background

MVC Technicalities

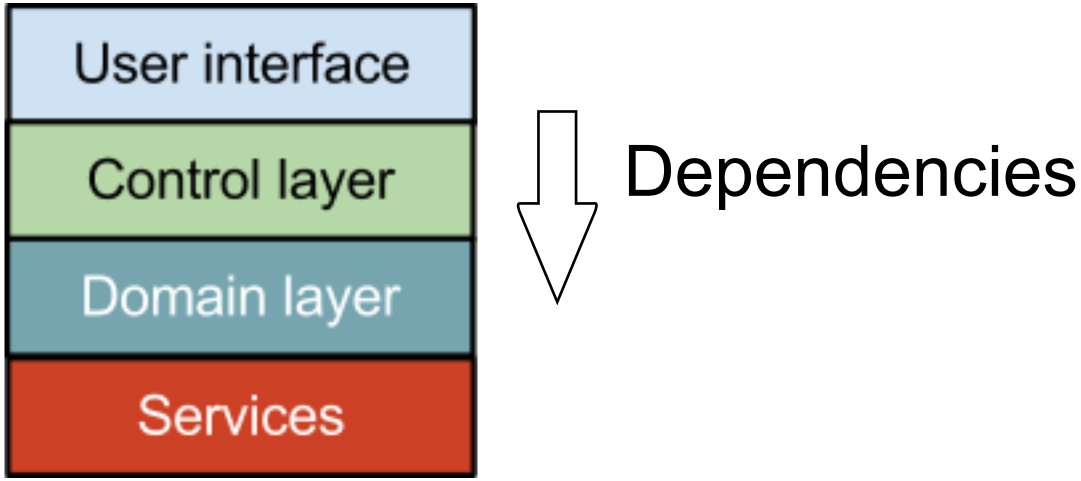
- Need Observer pattern to push state changes to GUI
- Many choices
  - Advanced: Google Guice, Context and Dependency Injection (CDI, a Java Standard)
  - Simple: Create a "in-house" EventBus
- **Design choice:** We'll use a simple EventBus
  - Excellent way to trace all events (they all pass the bus)
- How not to blur the model with event handling?
  - Events should be sent when state changes
  - State changes are normally in the set-methods
  - Use (possible internal, private) set-methods to separate event handling code from domain logic

Connecting GUI to Model

- Inspection and demo run of MP 0.4

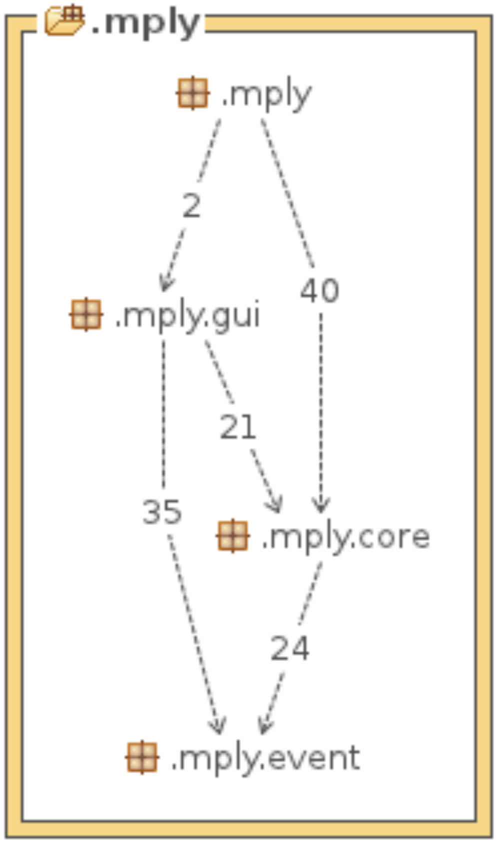
Dependencies

- High quality software is composed of loosely composed modules, i.e. few and controlled dependencies between modules (packages)
- Dependencies going down towards lower abstraction levels
- Typical layering



Dependency Analysis of MP 0.4

- Have UML diagrams for ocular inspection, but are we shore? Use a tool!
  - Eclipse has STAN plugin
  - Possible need to remove test classes (.class files from JUnit-tests)



Seems very good!

Exception handling

- Handle exceptions where it's possible
- If not possible in this class let caller try to handle it
  - Should normally be in domain or control layers
- Possible create central ExceptionHandler
  - Methods: ignore(e), retrow(e)
  - Possible to log all exceptions from one place
- Often need to inform GUI (show dialog), use EventBus
  - **Don't propagate exception** all the way to GUI

More Design Issues

- Mutability
- Handling of Resources (texts, images, config data, ...)
- State
- Swapping algorithms
- Canonical form for objects
- Immutable objects?
- Reducing dependencies (a constant design issue)
  - Interfaces

Mutability

- Always try to use immutable objects
    - Safe to share
- Use **final** all over as much as possible

Resources

- How to find/organize?
  - Use Resource Bundles
    - java.util.ResourceBundle
    - A map as a text file. Automatically read and converted to Java object
  - For images use ClassLoader class
    - getResource(s), findResource(), ...
  - Possible XML, use Java JAXP (API For XML processing)
    - Even better (simpler) XStream library, see sample on course page.

Application/Object States

- Object state
  - MP: Have movable (as state for Piece)
  - ..other?
- Identifying distinct states (modes) for application or objects
- Outcome depends on input **and** state
  - Example: Game character in state "dead" will not react to damaging input (events)
- Design pattern "State"

Changing the algorithm (behavior)

- To be able to swap algorithms use "Strategy pattern"
  - Example: Useful for game levels (all objects having same interface)
    - Level 1, simple algorithm (An object)
    - Level 2, a bit smarter (Other object)
    - ...
    - Level N, can't beat this (Yet other object)
- Also possible "Template Pattern"
  - If much of algorithm common to all objects

Canonical Object Form

- Do the object(s) need to be
  - Compared?
    - Override Object.equals()
      - If so also override hashCode()
  - Sorted?
    - Implement Comparable, Comparator
  - Cloned?
    - Override Object.clone()
  - Other general behavior...?

Summary

- We have done a few iteration and added a primitive GUI
- Have solved some design problems
- Hopefully the design is stable
  - If so, ... we start furiously to implement everything
- Our process have some weakness, the over all picture...

Next: System design...