The n-1 next iterations

Running through all phases ...

Iterating MP (1)

- Have a few use cases running
  - Move
  - EndTurn (i.e. switch player)
  - Buy/Sell property

Deepening the MP Model (1)

- During the iterations we discover and hopefully deepening our understanding of the domain model.
- Example: There's an rule "only the actual player" should be able to move
  - How should this be realized?
- We can of course disable components in GUI but that's **not the way to do it**
  - GUI shouldn't control the domain logic
  - Better: Add attribute "moveable" (boolean) to class Piece (or similar)
  - Also: moveable probably affects many parts of GUI (so use Observer)

Deepening the MP Model (2)

- We previously stated: "All actions only affect the actual player"
- But that's not true!!!
  - A card can affect the piece and the player or all players (not just the actual player)
  - Also: A card can move the piece! And possible a new card can move it again... the game running by itself!?!? Or?
- Possible: Design horror!!!!!
  - Will this break the design ...??
- Suggested solutions
  - Pieces and Players must be able to react to cards, let both have actOnCard()-method.
  - Card's have an action and an argument (int). move 2, pay 4000, ...

Transition from Command Line to GUI

- Until now we used the "command line loop" as IO for the application
- Time to add a GUI
  - Should have a preliminary one

- **What** will replace the command line as controller?

- Design choice for MP
  - We'll use a top-level class Monopoly as the control
  - Possible to port much of the command line version code to the class
  - GUI will call methods on Monopoly (should create interface in between)

Keep the Model Clean

- GUI applications use the MVC pattern
- Will introduce foreign (service) code into model
- During the transition we must try to keep the model as clean possible
  - **Try** to put service code in design classes (non-domain)
  - And of course no GUI code in model (a remainder)

MVC Technicalities

- Need Observer pattern to push state changes to GUI
  - Game loops possible a bit different
- 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 in-house EventBus
  - Excellent way to trace all events (they all pass the bus)
  - Will minimize service code in model (see demos from course page)

GUI Technicalities

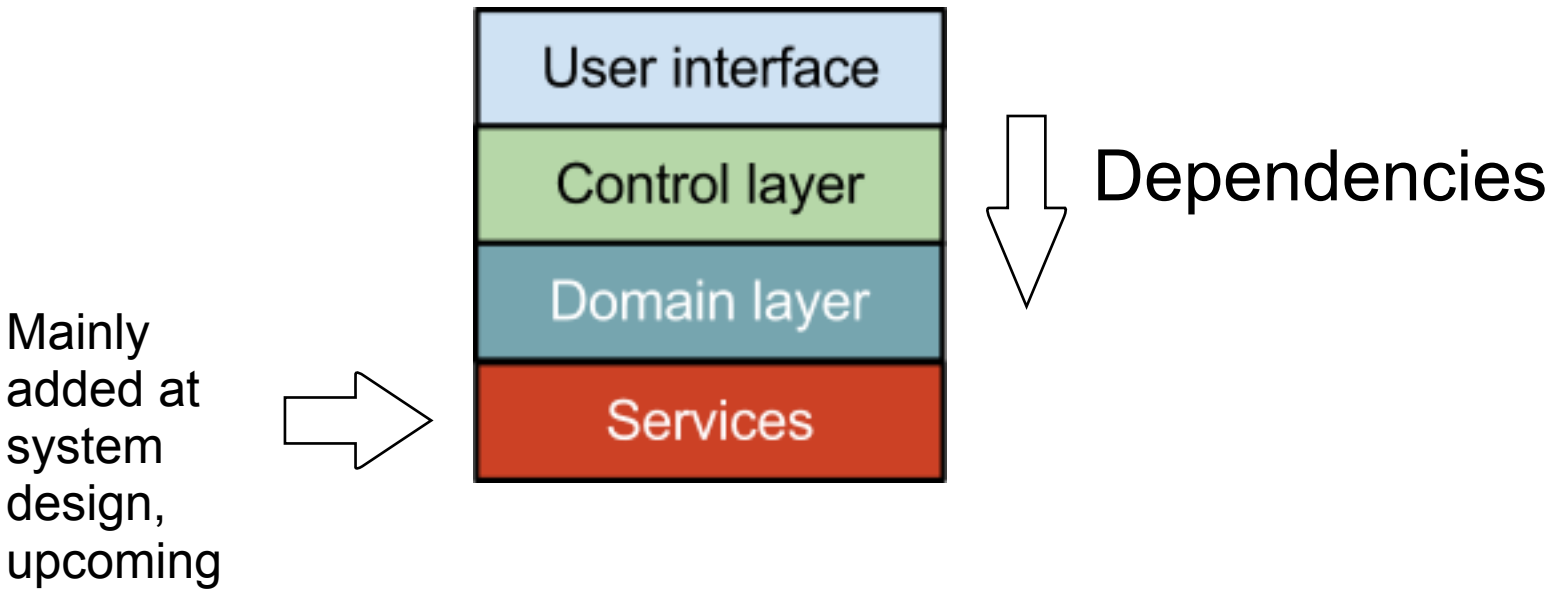
- State changes in model and other event possible updates GUI
- Code to update GUI resides in GUI (else testability issues)
  - In listener (actionPerformed)
    - before call to control/model
    - after call to control/model
  - In observer-callback method
- Swing single thread rule (all GUI code in Swing thread)
  - 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

Connecting GUI to Model

- Inspection and demo run of MP 0.4

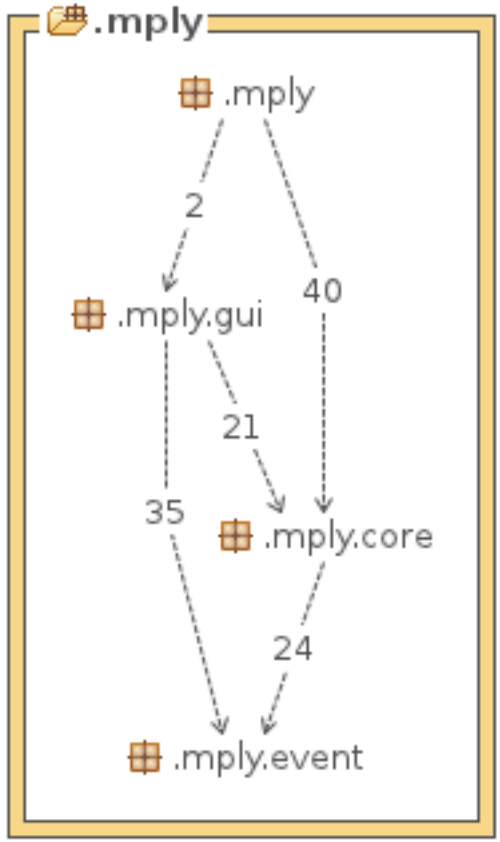
Dependencies

- 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
  - Low abstraction levels often uses primitive types
- Typical layering



Dependency Analysis of MP 0.4

- Have UML diagrams for ocular inspection, not fool proof? 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 actual method 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 Model Design Issues

- Still working with the model albeit we have a GUI

- A few examples
  - Mutability
  - State
  - Swapping algorithms
  - Canonical form for objects
  - Reducing dependencies (a constant design issue)
  - Interfaces

Mutability

- Always try to use immutable objects
  - Safe to share and more...
- Use **final** all over as much as possible
- Set initial state in contructor

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()
    - Very common to get the standard Java collections to work as expected
  - Sorting?
    - Implement Comparable, Comparator
  - Cloned?
    - Override Object.clone()
  - Other general behavior...?

Summary

- We have done a few iteration, thereby deepened our understanding of the model.
- Have solved some design problems
- Have added a (primitive) GUI
- Hopefully the design is stable
  - If so, ... we start furiously to implement everything
- Our process have some weakness, the over all design i.e. "the big picture"...

Next: System design...