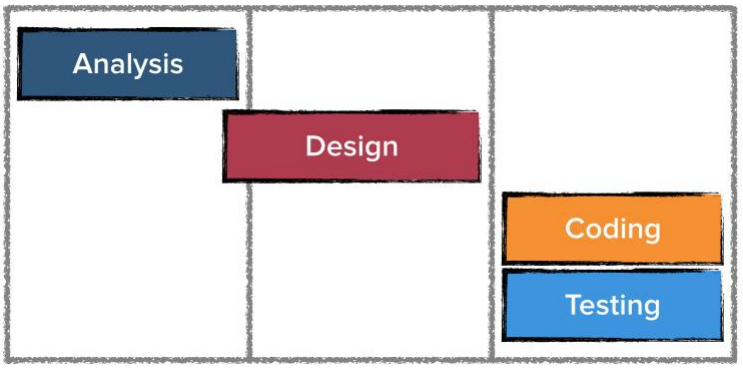
More Model and Application Design

Slide Series #5

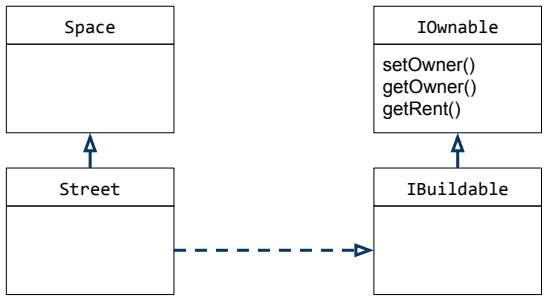
Starting out Iteration 2



We'll run a complete cycle again (not much of requirement)!

- Using MP as illustration

MP : Extending the design model




```
if (s instanceof IOwnable) {
    IOwnable o = (IOwnable) s;
    return o.getOwner() … // Or getRent()
} else {
    ...
}
```

Remember: Design model is an adaption of the domain model, more suitable to implement.

Need some more classes to implement alternate flow for UCs Move

- Pay rent (landing on property owned by other)
  - NOTE: This use case potentially will have a lot of user interaction (user possibly must sell, broke, …)
- Go to jail (landing on Go to Jail)
- Speeding (3 consecutive double-sixes, must go to jail)
- Update domain (Street) and design model (Street, IOwnable, IBuildable)

NOTE: Would like to have uniform handling of Spaces (Street, Chance, Community Chest, etc)

- Easy to send List<Space> to GUI for rendering

Hmm, forgot: If in Jail when starting use case Move?

- Should show dialog, present possible actions … or?
- … go back and modify use case text!

MP: Update Use Case Move

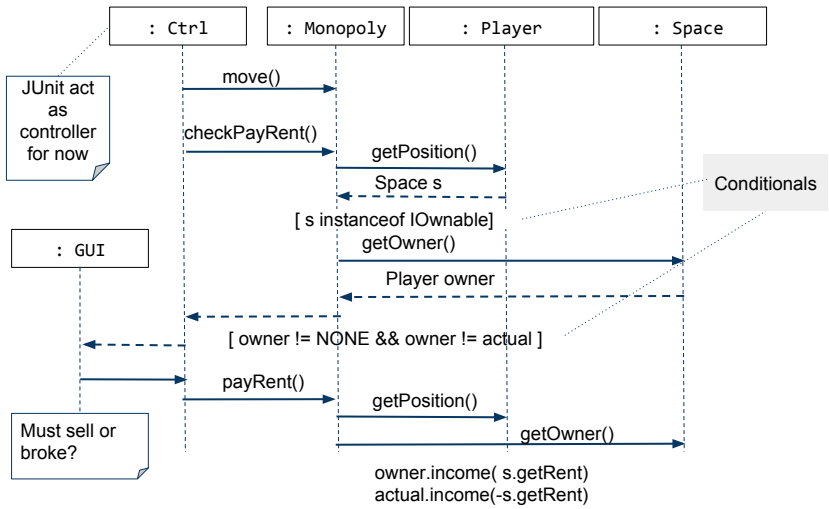
1. Move

Summary: The game has started. Actual player moves piece on the board, **Player not in Jail (see use case "In Jail")**
Priority: High
Extends: DoTurn
Includes: RollDice
Participators: Player

	Actor	**System**
1	Click Roll button	
2		Result for two dices shown Piece removed from actual position and put in new position Roll button disabled
2.1 Passed Go		If player passed go, player balance flashes (updated) and a "cash"-sound is played

We forgot so have to updated use case text!

MP : Dry Run Pay Rent



We continue to use JUnit tests

- This UC need user interaction
  - We know we will have a control layer because we use MVC (not all applications do it like this)
  - Interaction handled by control layer
- Control for now is the JUnit test.

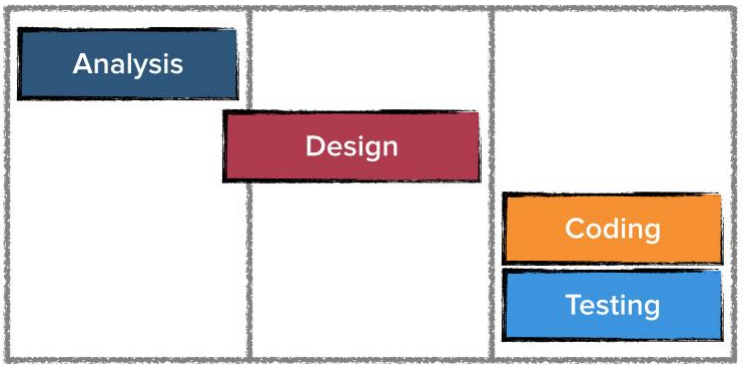
If getting complex … !

- Simplify: Use notes, pseudo code, …

MP : Monopoly-0.2

Download from course page, inspect and run!

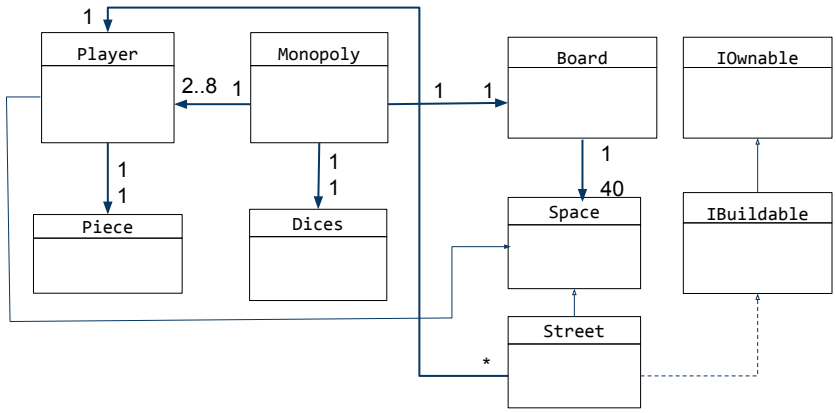
Starting out Iteration 3



This iteration will produce a complete application with GUI

- So for now will focus on system design (i.e. vs model design)

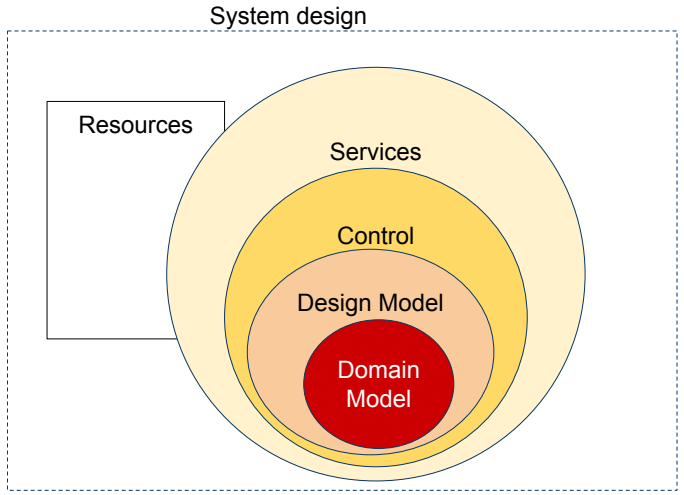
MP : Design Model v 0.3



Design model at end of iteration 3.

- Later put in SDD see upcoming slides

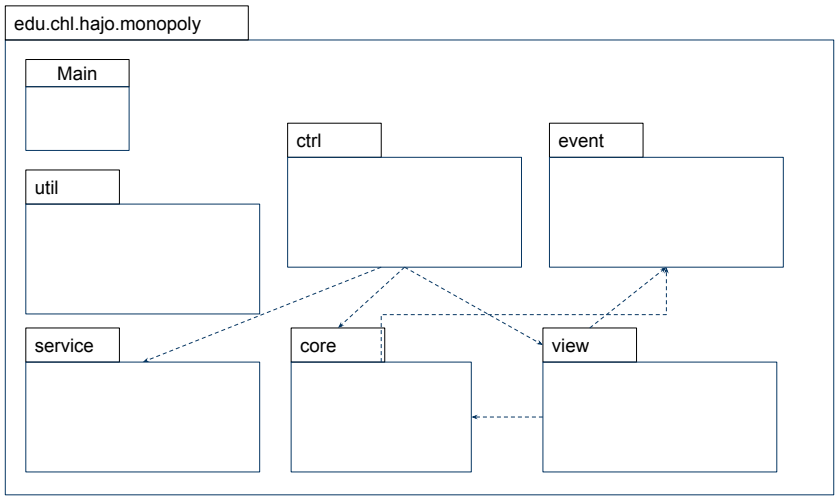
Designing the full Application



This is an abstract view how an OO-application/system should “look”

- Domain model is the core classes from the analysis
- Design model is the domain model adapted for implementation
  - Extended with “technical”-support classes
  - For MP: IOwnable, IBuildable (so far)
- Control is a layer coordinating the flow between the model and services
  - So far handled by JUnit tests
- Services are everything supporting the model (no services so far)
  - GUI
  - Handling of resources
  - Persistence (save to file, database)
  - Communication (network, …)
- Resources
  - Data for configuration, initialization, ...
  - Images, sounds, …
  - i18n data

Concrete Structure



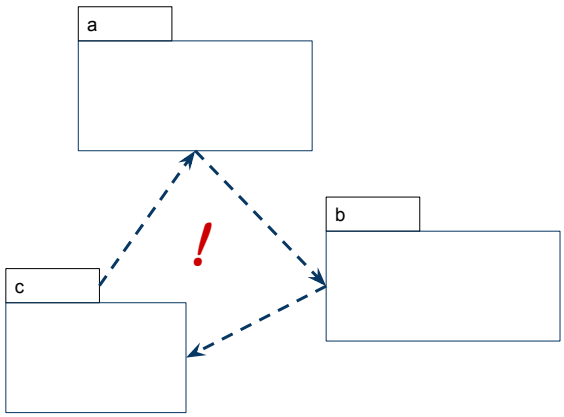
Application should be partitioned into packages.

- Will organize the overall structure of application.
  - Each package should have a well defined purpose
- NOTE: Arrows shows dependencies
  - util and config used by many but uses NONE (only incoming arrows)
    - Arrows for util and config not shown, would clutter up
- NOTE: Model not dependent on services (used via ctrl)
- Package structure should guarantee unique qualified class names
- Use UML package diagram

Packages

- edu.chl.hajo.monopoly: (nested) package(s) for full application. Using approx. reversed internet domain
  - Only class (for now) Main. Application start class (main method)
- util: non-application specific classes (possibly reusable)
- service: classes for file handling, etc.
- ctrl: control classes
- event: event handling inside application (not Swing events) more to come
- view: GUI classes
- core: the model

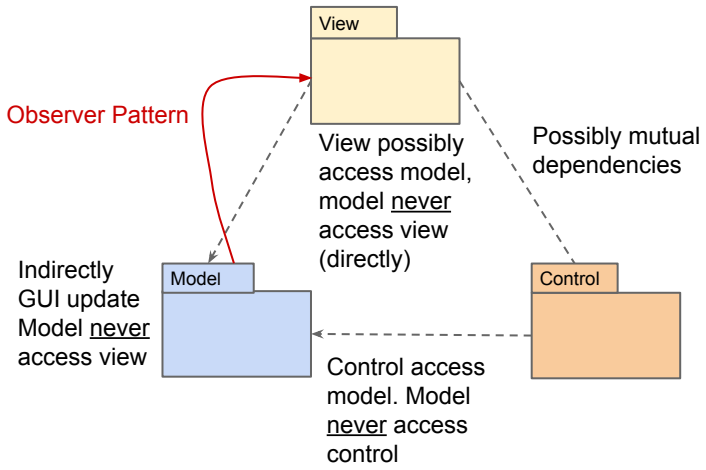
Circular Dependencies



Circular dependencies between packages

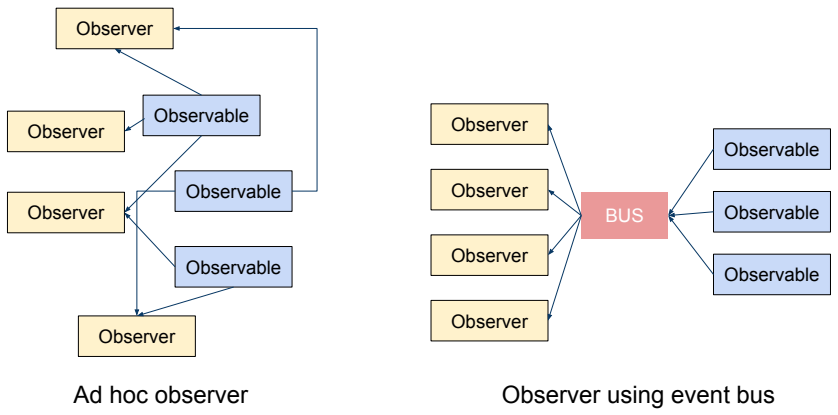
- Same problems as mutual dependencies between classes
- Must avoid, see tools … (upcoming)

MVC Design Review



There are many opinions about MVC.

Implementing Observer



Implementation of observer better use an event based model with an event bus

- Bus globally accessible (Singleton)
- Observables publishes events
- Observers register as event handlers
- All event pass through the bus, possible to inspect/log events!

MP: Will use a simple event bus

Keep Model Clean

```
public class Dices {

    private int first;
    private int second;
    ...
    private void setFirst(int first) {
        this.first = first;
        EventBus.BUS.
            publish(new Event(Event.Tag.DICE_FST, first));
    }

    private void setSecond(int second) {
        this.second = second;
        EventBus.BUS.
            publish(new Event(Event.Tag.DICE_SEC, second));
    }
}
```

Don't want to clutter model classes with event publishing all over

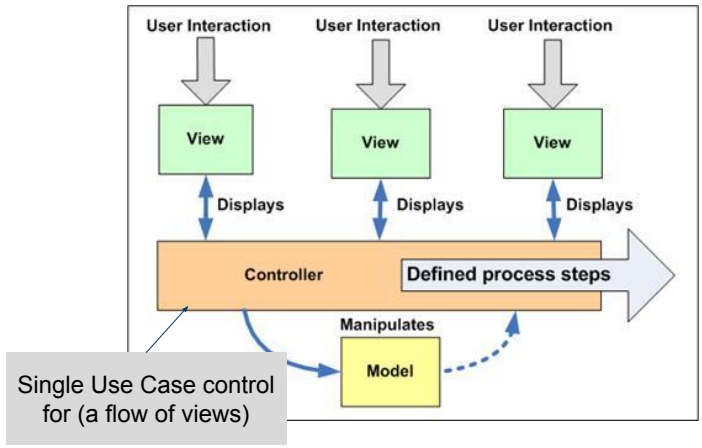
- Event publishing ONLY in setters (possibly private)
    - Class must use setters, no bare assignments!
- Should make it easy to locate observables behaviour

The Need for a Control Layer

How should GUI and model interact in MVC?

- Should model be updated after each tile?

Use case Controllers



Control layer could also be comprised of “use case controllers” (classes)

- Each UC (possibly part of) handled by a specific controller class.
  - Easy to locate use cases
  - Class runs UC parts not present in model or mediated UC between view and model …
  - … or between model and services.
- Slide shows a use case with many views (must not be the case)

MP: Not used so far

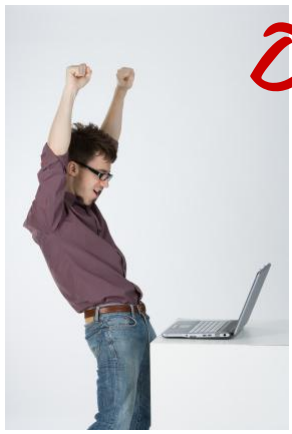
Choosing GUI Technology

Many choices …

- .. search web!
- MP: Will use Swing (Java2D)
- Maven or Gradle should handle dependencies.

NOTE: Swing can be really annoying in between …?

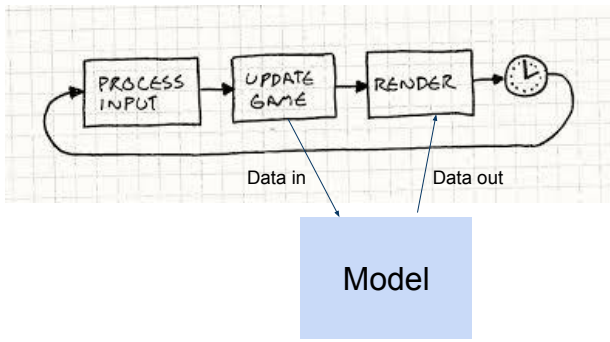
MP : Monopoly-0.3 (MVC)



Demo time

Download from course page, inspect and run!

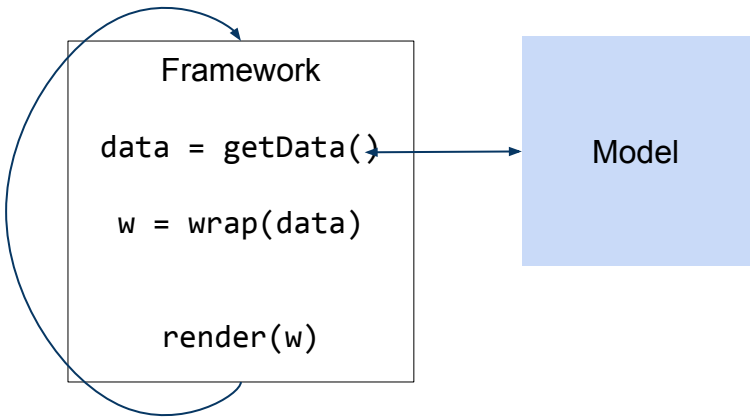
Framework MV Design



If using a graphics framework normally no full MVC design

- Mostly using a pull design (render ask model for data)
  - Observer is a push design!
- Control replaced by update game (method periodically called by framework)

Framework Render Model



If rendering handled by framework

- NO rendering data in model!
  - Let framwork (if needed wrap and) consume the data
  - Framework render consumed data
- Keep model clean!!!

Design Review

- Every class has well defined responsibility (represents one concept)?
- Split or collapse classes? Introduce generalization?
- Missing or unnecessary classes (convert to attribute)?
- Directions of associations
- No cyclic traversion of associations or dependencies (no mutual)
- Model in one package (possibly organisational subpackages)?
- Interface(s) to model (model package) to use by others?
- Building the model (factories)?
- Aggregates and call chains?
- Parameterization of model (user options)?
- Absent values (avoiding null)
- Minimize state
- Canonical form
- Is everything located in one single place
- Is flow consequent (same flow for all of events (of same type))
- Testability

Regularly review design until stable.

- Refactoring!

Quality Tools

Use tools to increase design and code quality!

- See web!
- Possible to incorporate into pom.xml (if Maven project)

SDD

System design document for NNN

1 Introduction
1.1 Design goals
1.2 Definitions, acronyms and abbreviations
2 System design
2.1 Overview
2.2.1 General
2.2.2 Decomposition into subsystems
2.2.3 Layering
2.2.4 Dependency analysis
2.3 Concurrency issues
2.4 Persistent data management
2.5 Access control and security
2.6 Boundary conditions
…

The system design documents (SDD) overall goal is to make the application possible to understand (as quick as possible)

- The system design is recorded in the System Design Document (SDD). This document completely describes the system at the architecture [high] level, including subsystems and their services, hardware mapping, data management, access control, global software control structure, and boundary conditions [start/stop]. A foundational guide for further implementation details all the way to an executable solution.
- Audience: The audience for the SDD includes the software architect and lead members (liaisons) from each subsystem development team (i.e. programmers).
- The SDD is a "live" document that should be incrementally expanded and refined during/after iterations.
- This is about communication, no absolute rules how to write
- We prefer this top down explanation approach
    - Start out high level (big picture)…
        - Hardware setup, communication applications involved (if more)
    - .. then refine in each step …
        - Structure of (each) application
        - Packages
        - Possibly classes/Interfaces
        - Design model
    - … until close to code (when reaching this level the code and the tests are the documentation)
- Another template

- SDD Sample
- And yet other sample

Summary

Iteration 2 and 3

- We got more UC's up and running!
- We got a full MVC version of the first UC's

Next: Next iteration, a service, exceptions, ...

Code sample for iteration 1: monopoly-0.2/0.3 (course page)