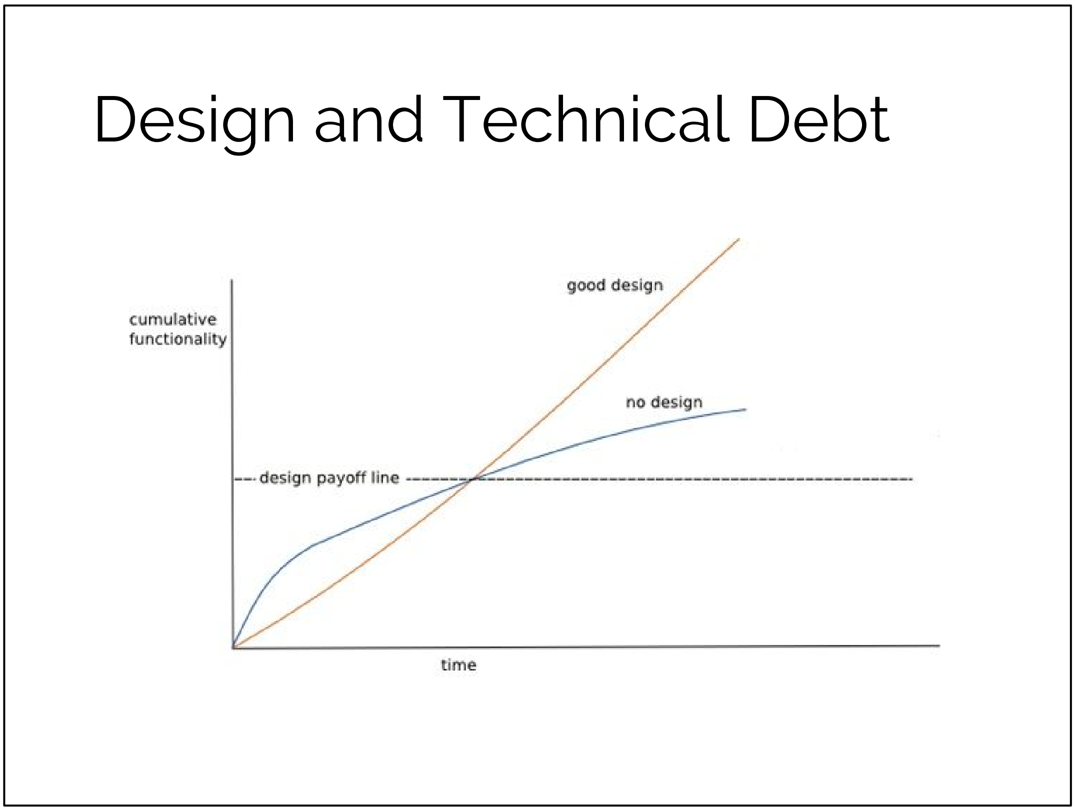
Design

Slide Series #5

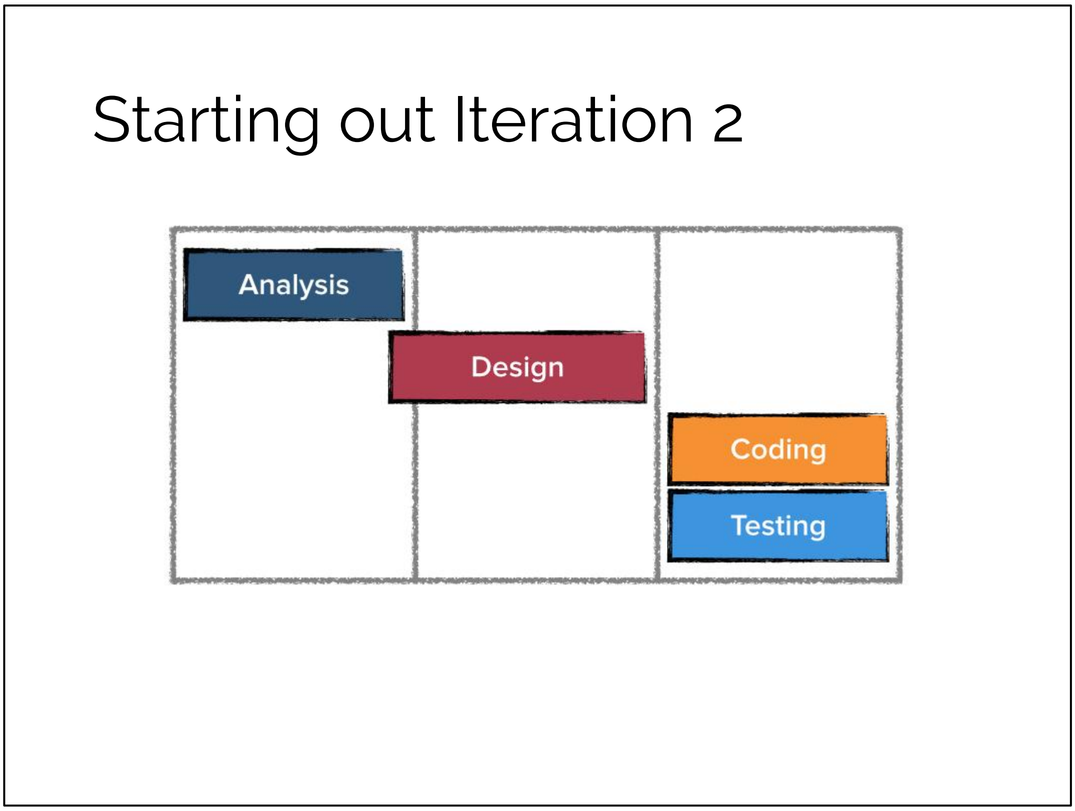


Design and Technical Debt

Technical debt is "a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution".

So from here the design is a matter of great concern to us!

- We design the model
- We design the full application



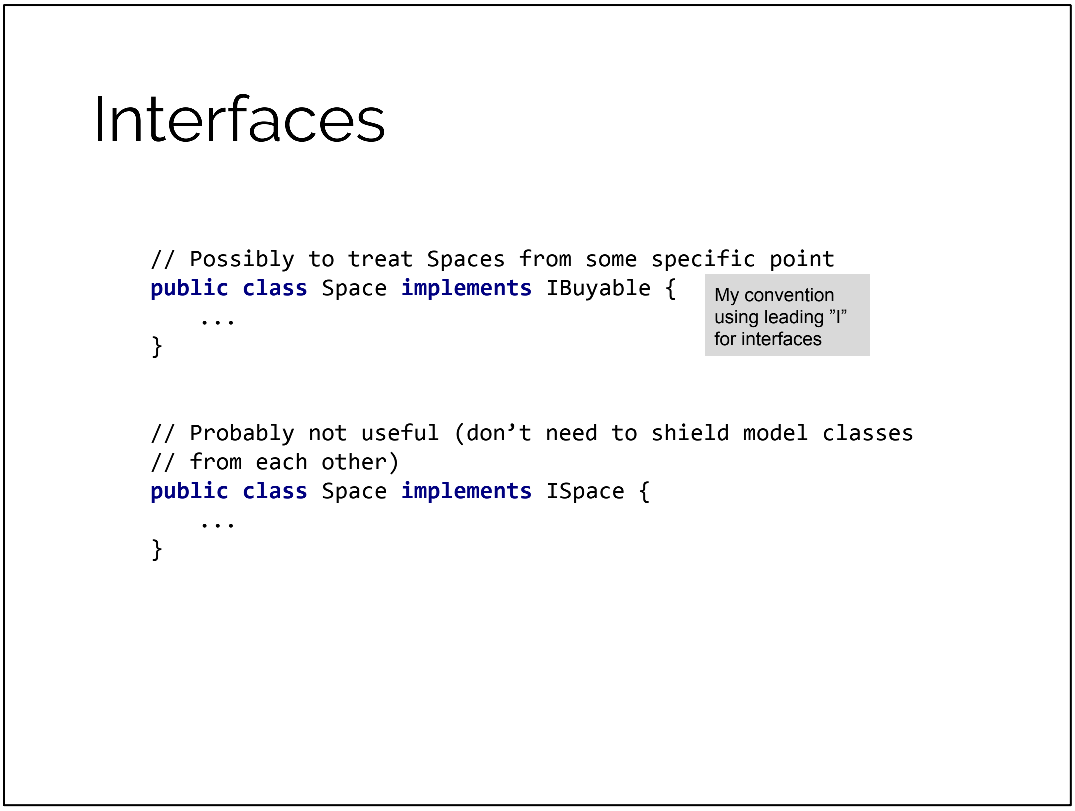
Starting out Iteration 2

We'll run a complete cycle again (no new requirement for now)!

- The goal is to get a stable model! (before starting with full application design)

Add 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)

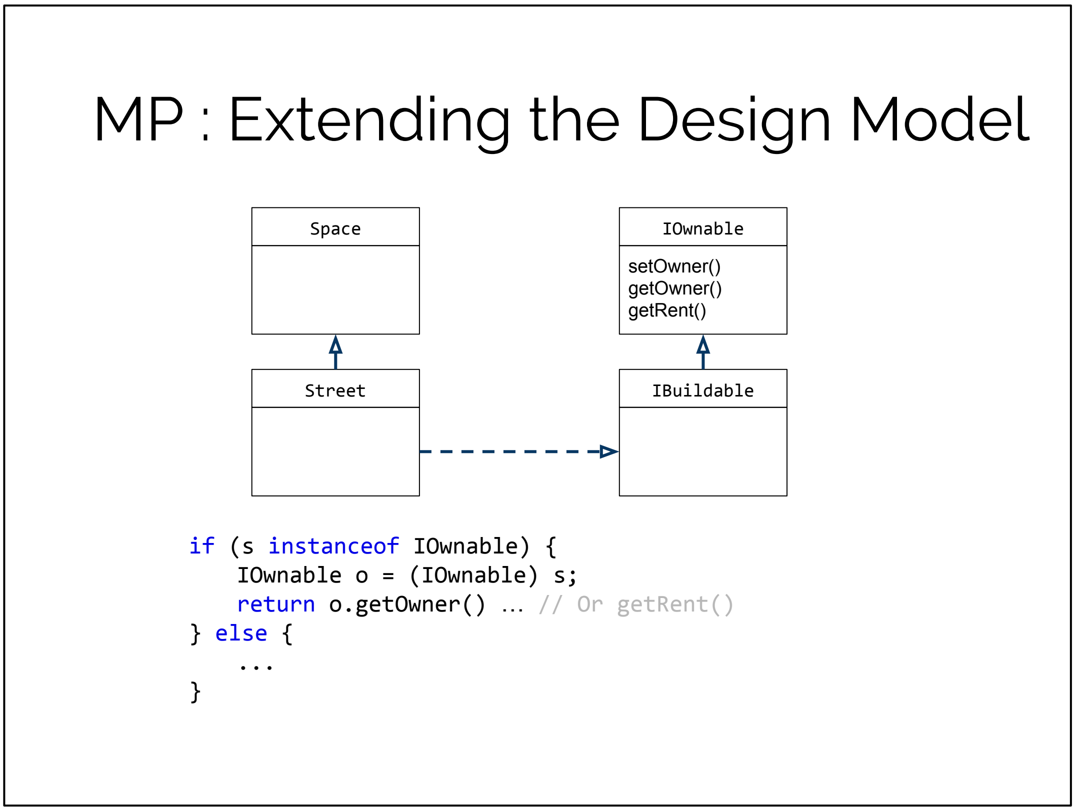


Interfaces

```
// Possibly to treat Spaces from some specific point
public class Space implements IBuyable {
    ...
}


// Probably not useful (don’t need to shield model classes
// from each other)
public class Space implements ISpace {
    ...
}
```

My convention using leading ”I” for interfaces

Use of interfaces
- Make objects “the same type”.
  - Guarantee some general operations is present
  - Possible to store heterogenous objects in Collections
  - This use is possible in Model
- The “seams” in the system, shielding different parts of application.
  - Not in model (model is a single part)



MP : Extending the Design Model

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

Use of Space class

- Common data for Street, Chance, Community Chest factored out to base class Space
- Easy to send List<Space> to GUI for rendering .. but need use instanceof …. (not so good) .. to find runtime type.
- Spaces with specific characteristics (operations) implements interfaces

Update domain: (Street) and design model (Street, IOwnable, IBuildable)

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

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

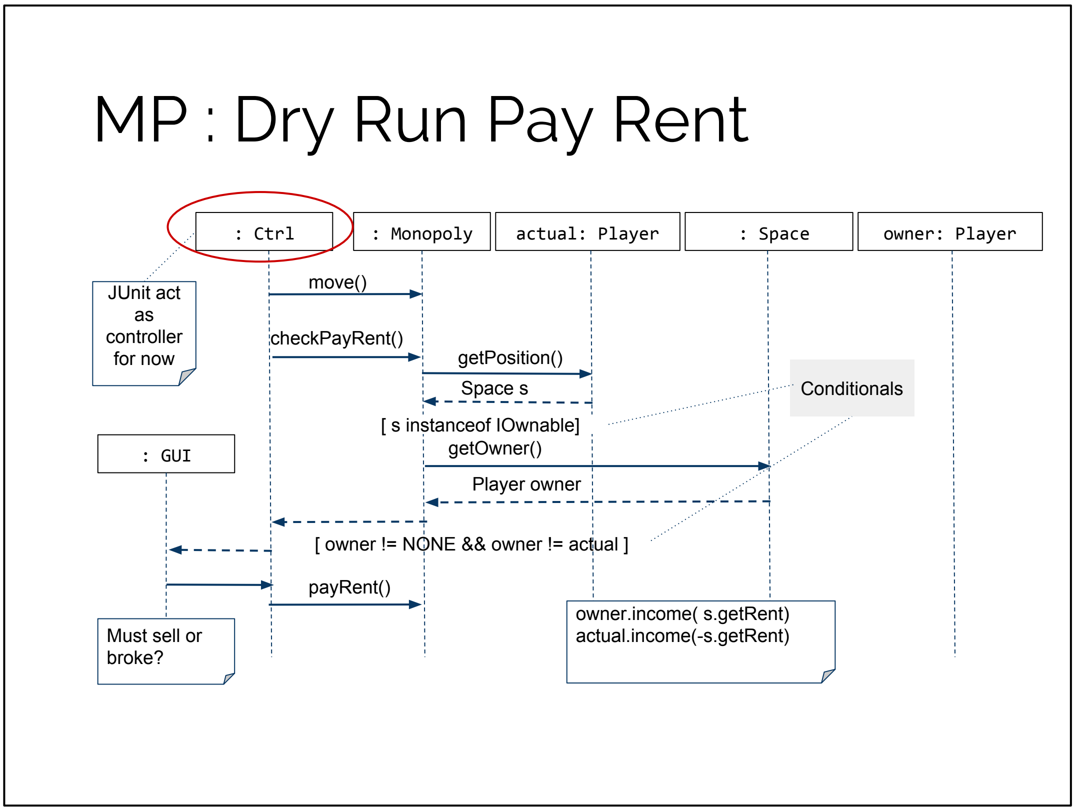
MP: Update Use Case Move

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

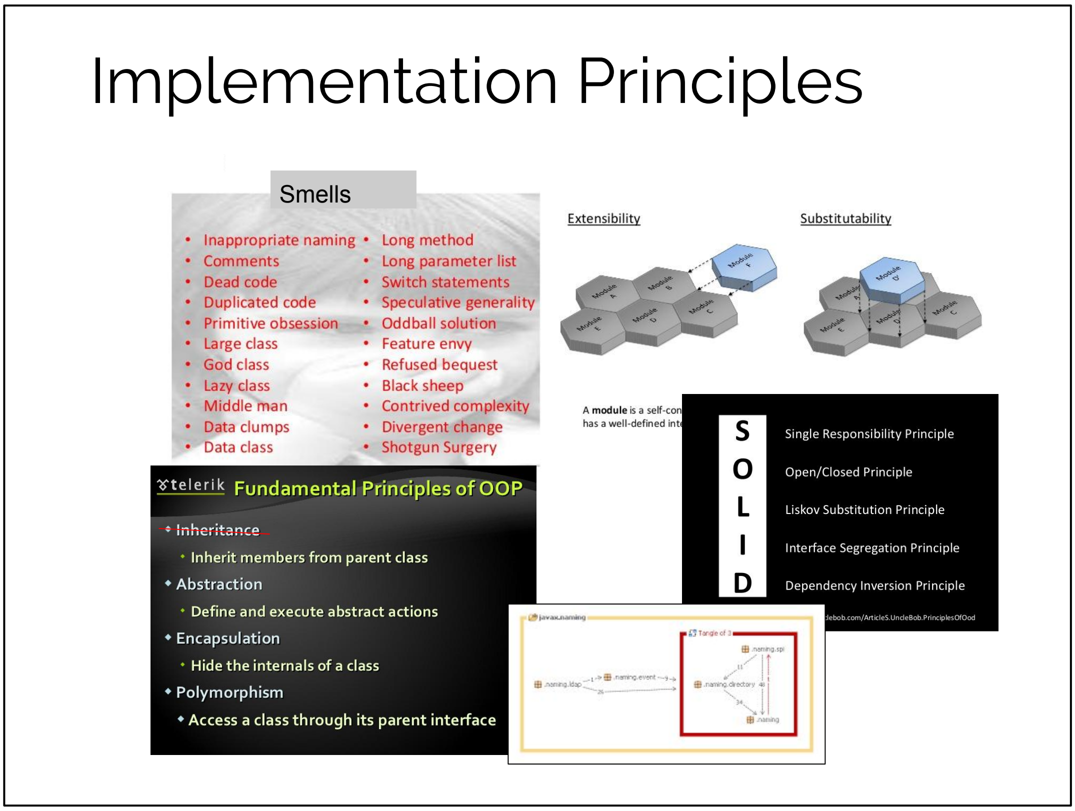
- This UC need user interaction
    - We know we will have a control layer because we use MVC (not all applications do it like this)
    - Interaction handled by control layer
- Control for now is the JUnit test.

If getting complex … !

- Simplify: Use notes, pseudo code, …

MP : Monopoly-0.2

Download from course page, inspect and run!



Implementation Principles

Smells

- Inappropriate naming
- Comments
- Dead code
- Duplicated code
- Primitive obsession
- Large class
- God class
- Lazy class
- Middle man
- Data clumps
- Data class
- Long method
- Long parameter list
- Switch statements
- Speculative generality
- Oddball solution
- Feature envy
- Refused bequest
- Black sheep
- Contrived complexity
- Divergent change
- Shotgun Surgery

telerik **Fundamental Principles of OOP**

- ~~Inheritance~~
  - Inherit members from parent class
- Abstraction
  - Define and execute abstract actions
- Encapsulation
  - Hide the internals of a class
- Polymorphism
  - Access a class through its parent interface

Extensibility

Substitutability

A **module** is a self-con
has a well-defined inte

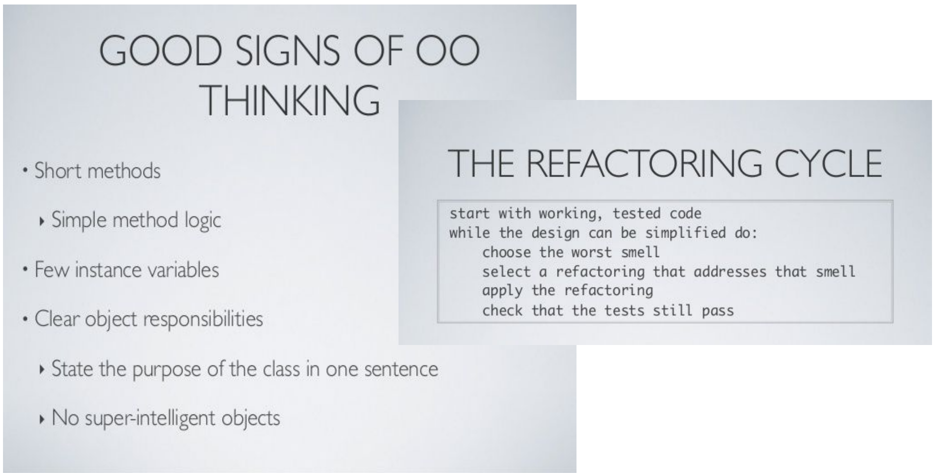
S Single Responsibility Principle
O Open/Closed Principle
L Liskov Substitution Principle
I Interface Segregation Principle
D Dependency Inversion Principle

lebob.com/ArticleS.UncleBob.PrinciplesOfOod

There are quite a few design principles and best practices

- Can't remember them all, … but during code reviews, check the list

Refactoring



GOOD SIGNS OF OO THINKING

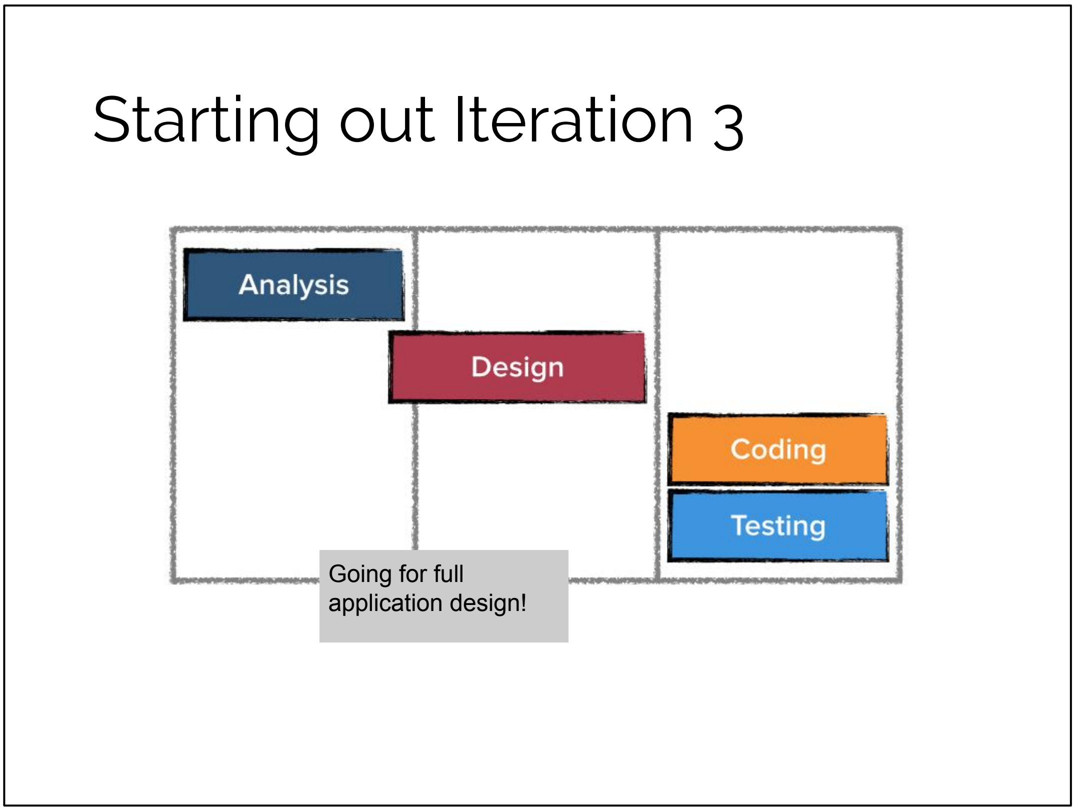
- Short methods
  - Simple method logic
- Few instance variables
- Clear object responsibilities
  - State the purpose of the class in one sentence
  - No super-intelligent objects

THE REFACTORING CYCLE

```
start with working, tested code
while the design can be simplified do:
    choose the worst smell
    select a refactoring that addresses that smell
    apply the refactoring
    check that the tests still pass
```

Should refactor aggressively after each iteration!

- Check against implementation principles
- Refactoring patterns

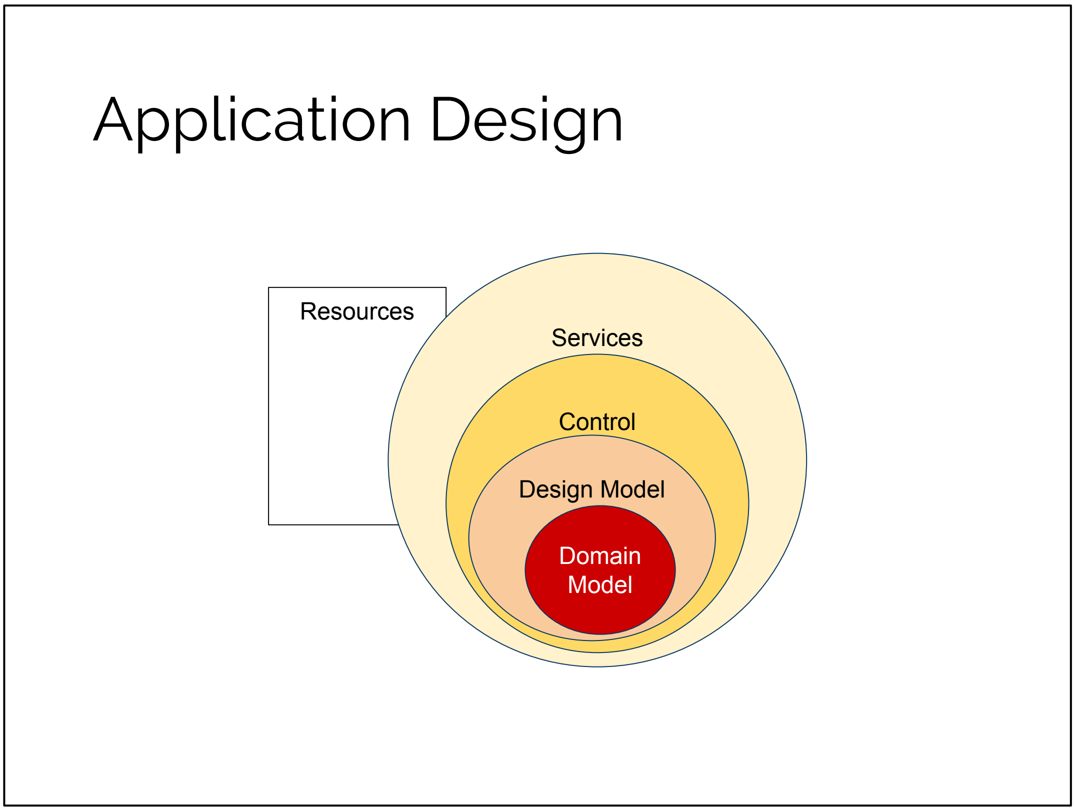


Starting out Iteration 3

Going for full application design!

This iteration will produce a complete application with GUI and a MVC model

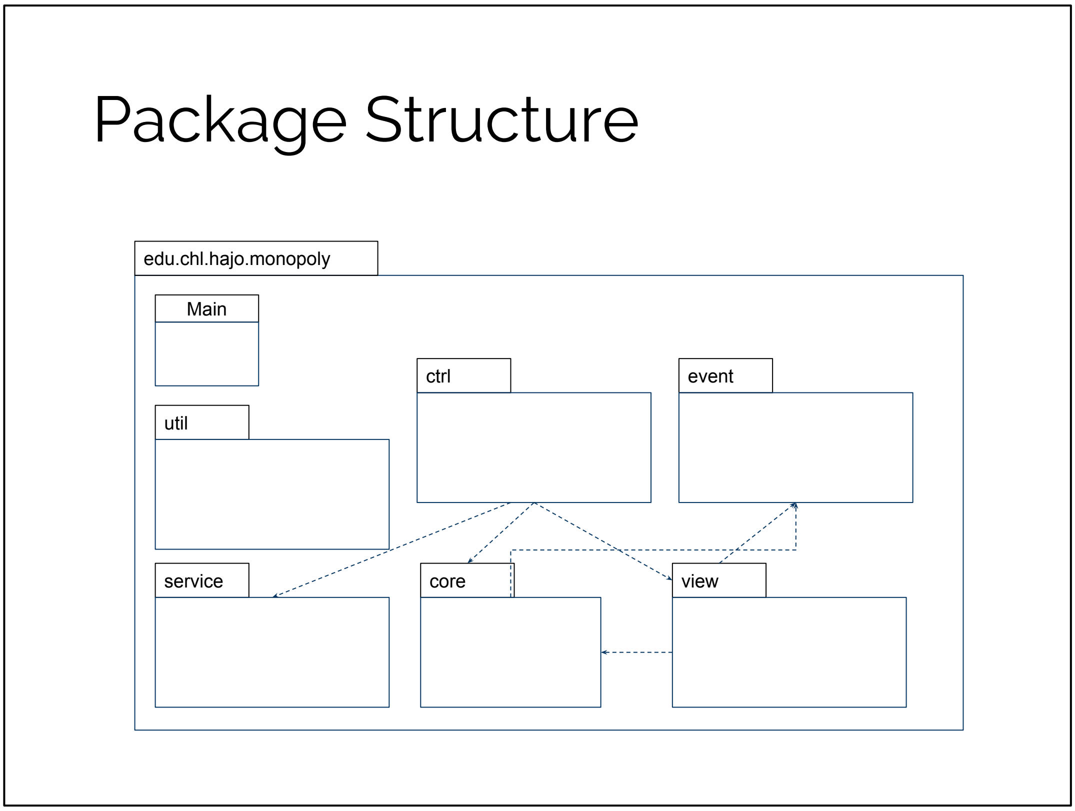
- So for now will focus on system design (i.e. vs model design)



Application Design

This is an abstract view how an OO-application (system) “should” look (not universally true, but ok for this course)

- 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



Package Structure

Application should be partitioned into packages.

- Will organize the overall structure of application.
    - Each package should have a well defined purpose (same as classes, methods)
- 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 more later ...)
- Package structure should guarantee unique qualified class names
- Use UML package diagram

Packages

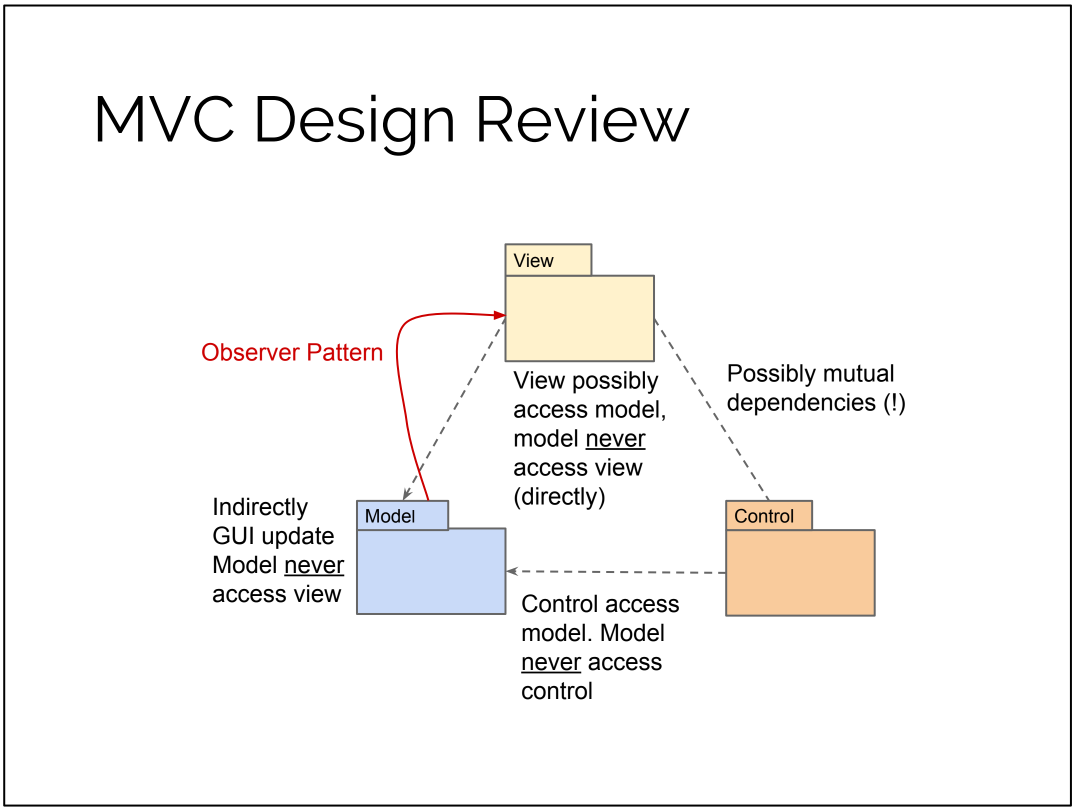
- edu.chl.hajo.monopoly: (nested) package(s) for full application. Using approx. reversed internet domain
    - Only class (for now) Main. Application start class (main method)
- util: non-application specific classes (possibly reusable)
- service: classes for file handling, etc.
- ctrl: control classes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

Choosing GUI Technology

JAVA 2D API Graphics
THE SUN MICROSYSTEMS PRESS JAVA SERIES
VINCENT J. HARDY
SWING!
SWIX
SwiXML
jMonkeyEngine
libGDX
JavaFX
Qt Jambi

Many choices …

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



MVC Design Review

Observer Pattern

View possibly access model, model never access view (directly)

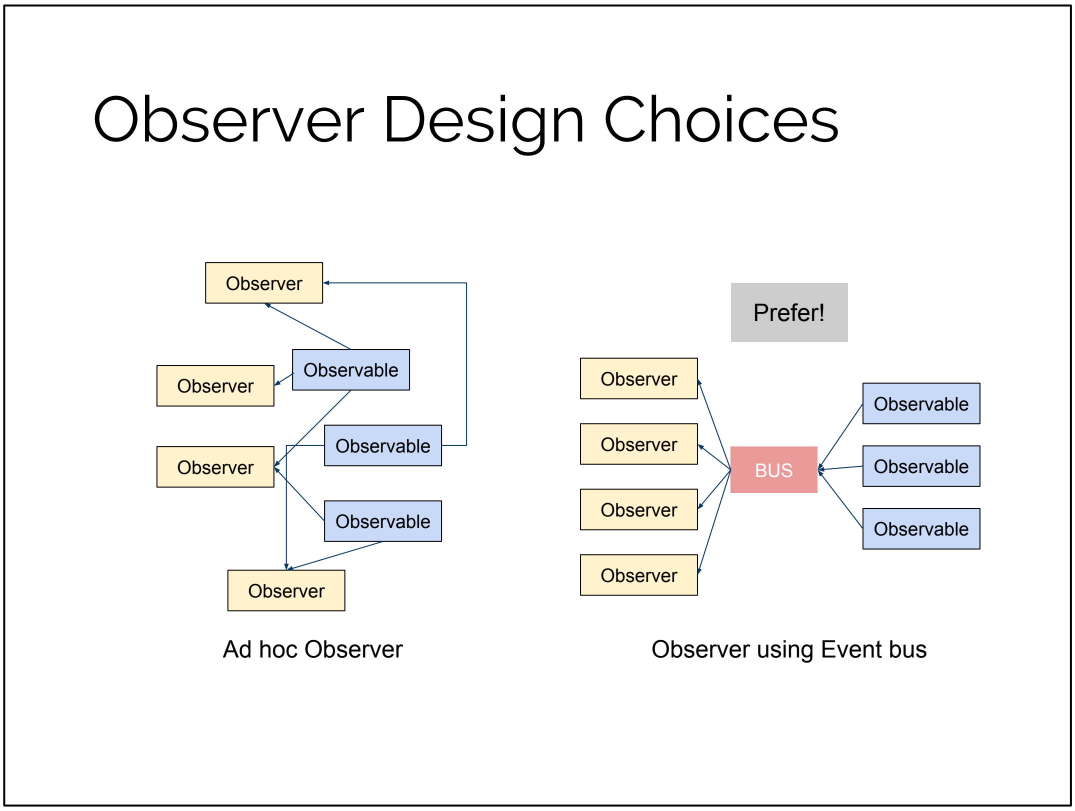
Possibly mutual dependencies (!)

Indirectly GUI update Model never access view

Control access model. Model never access control

There are many opinions about MVC.

- When using Observer this is a push design (vs. pull design)



Observer Design Choices

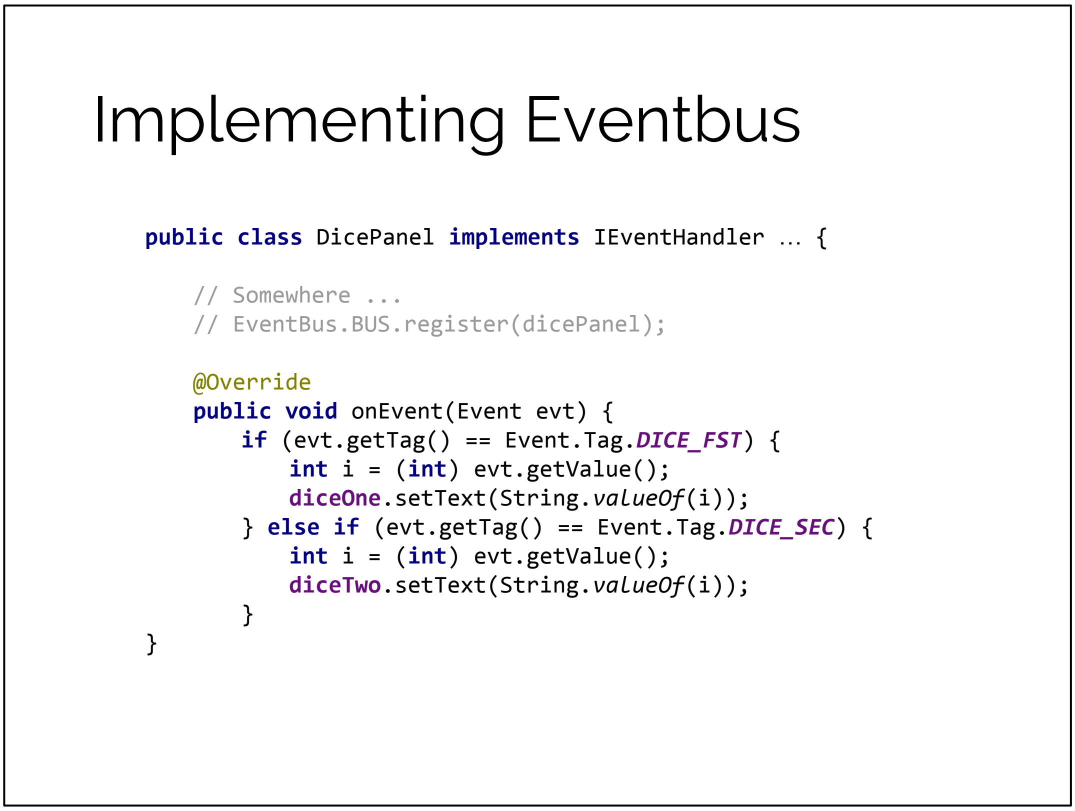
Ad hoc Observer

Observer using Event bus

Implementation of observer better use an event based design 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!

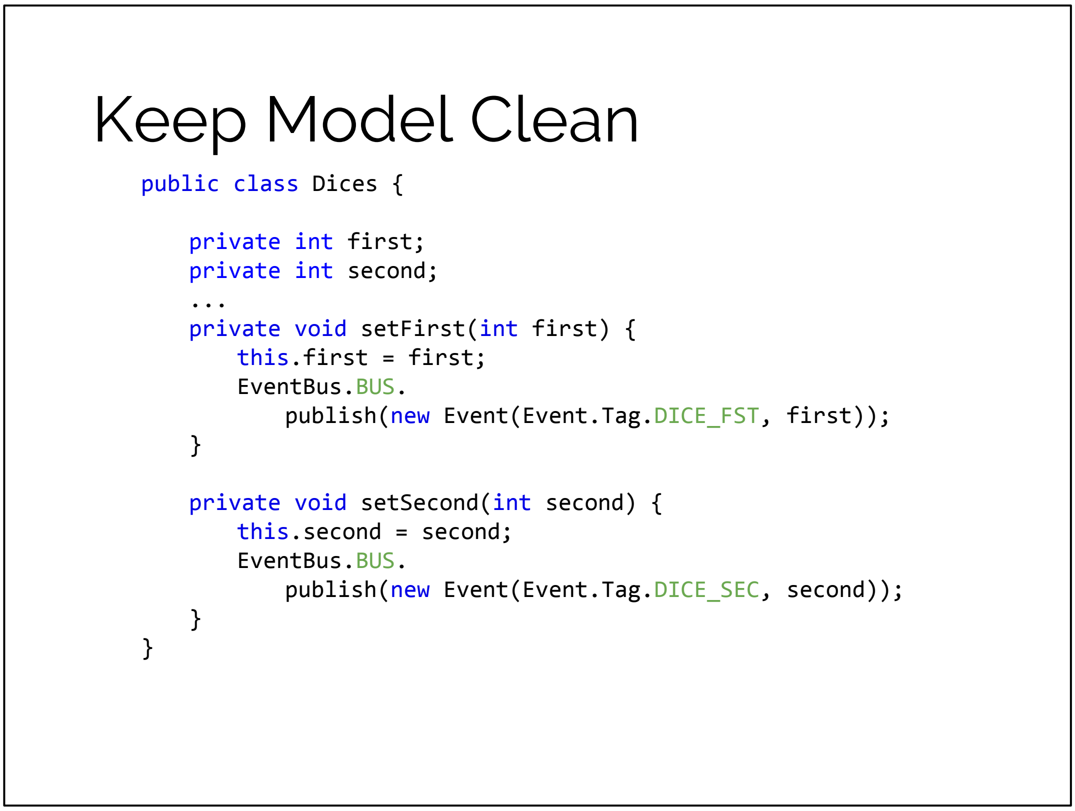
MP: Will use a simple, in house, event bus



Implementing Eventbus

```
public class DicePanel implements IEventHandler … {

    // Somewhere ...
    // EventBus.BUS.register(dicePanel);

    @Override
    public void onEvent(Event evt) {
        if (evt.getTag() == Event.Tag.DICE_FST) {
            int i = (int) evt.getValue();
            diceOne.setText(String.valueOf(i));
        } else if (evt.getTag() == Event.Tag.DICE_SEC) {
            int i = (int) evt.getValue();
            diceTwo.setText(String.valueOf(i));
        }
}
```

EventBus is a singleton class with methods register/unregister/publish.
IEventhandler is interface with method onEvent



Keep Model Clean

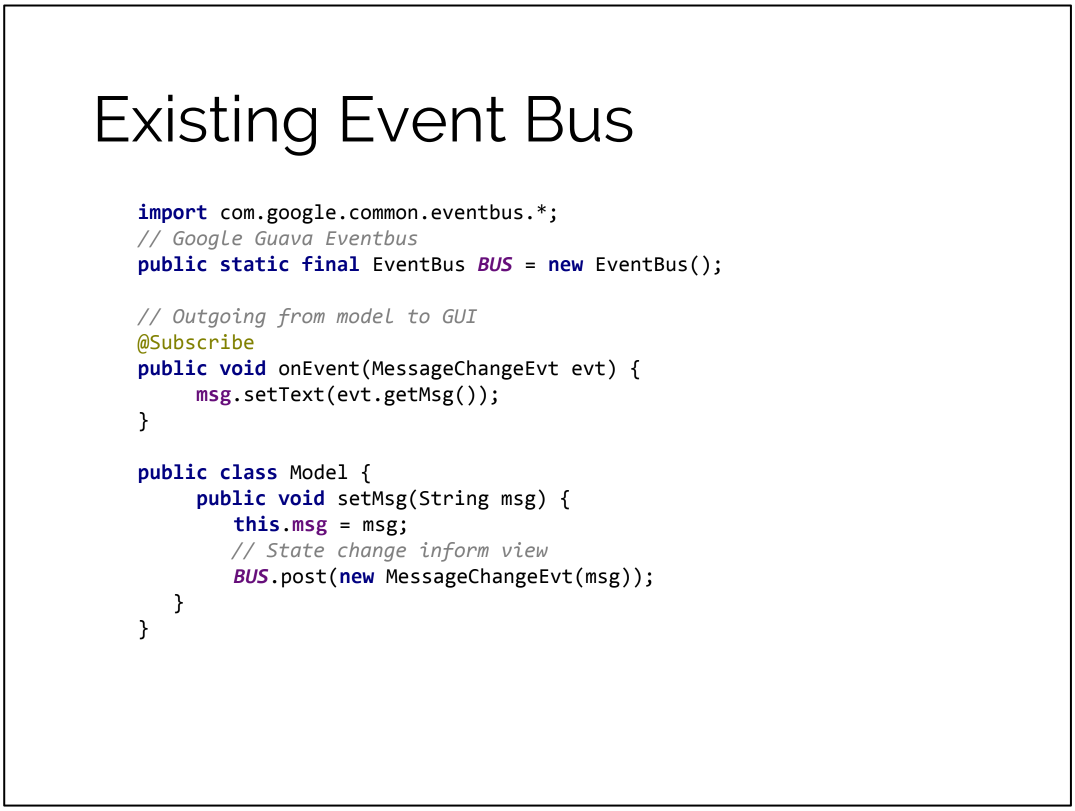
```
public class Dices {

    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

Or…

- Wrap model class in Observable (forward calls to real model class)
- Extend Model class, add publishing in sub class

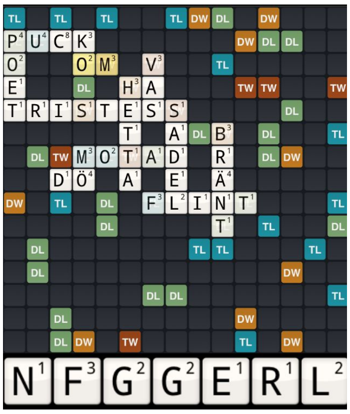


Existing Event Bus

```
import com.google.common.eventbus.*;
// Google Guava Eventbus
public static final EventBus BUS = new EventBus();

// Outgoing from model to GUI
@Subscribe
public void onEvent(MessageChangeEvt evt) {
     msg.setText(evt.getMsg());
}

public class Model {
     public void setMsg(String msg) {
        this.msg = msg;
        // State change inform view
        BUS.post(new MessageChangeEvt(msg));
   }
}
```

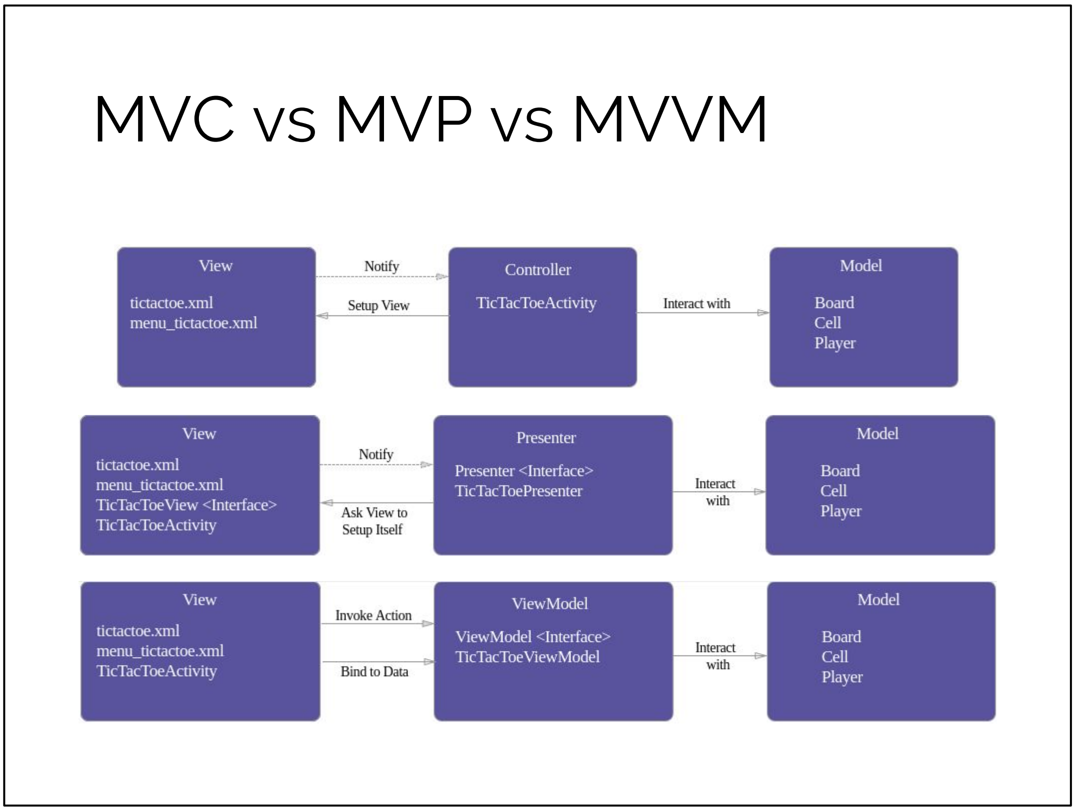
Google guava eventbus

The Need for a Control Layer



How should GUI and model interact in MVC?

- Should model be updated after each tile?
- No, … if so possible have to remove single chars, better let control compose a full word, then add it (when to check word is a real word?)..



MVC vs MVP vs MVVM

View
tictactoe.xml
menu_tictactoe.xml

Notify

Setup View

Controller
TicTacToeActivity

Interact with

Model
Board
Cell
Player

View
tictactoe.xml
menu_tictactoe.xml
TicTacToeView <Interface>
TicTacToeActivity

Notify

Ask View to Setup Itself

Presenter
Presenter <Interface>
TicTacToePresenter

Interact with

Model
Board
Cell
Player

View
tictactoe.xml
menu_tictactoe.xml
TicTacToeActivity

Invoke Action

Bind to Data

ViewModel
ViewModel <Interface>
TicTacToeViewModel

Interact with

Model
Board
Cell
Player

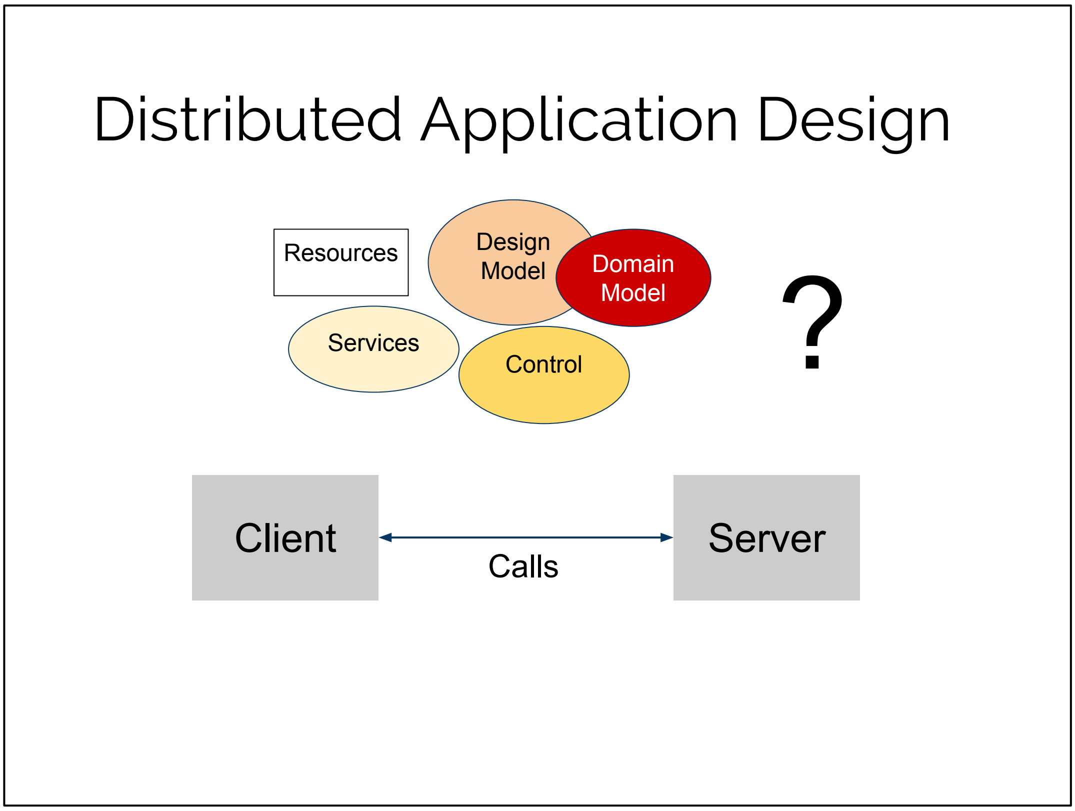
This seems to be an issue for Android developers.

- Can’t see very principally different ideas, ...
- ....will possibly be beneficial for technical reasons?
    - If so use!

MVC vs MVP vs MVVM

MP : Monopoly-0.3 (MVC)

Download from course page, inspect and run!



Distributed Application Design

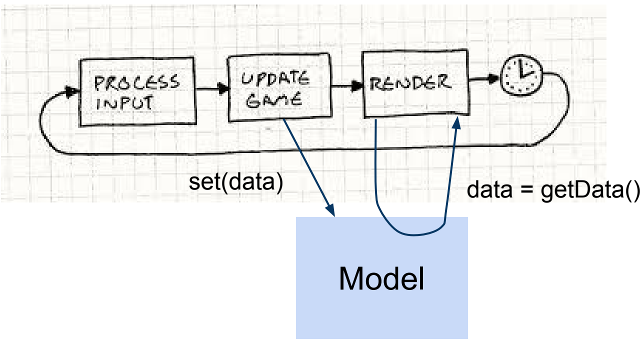
Resources
Design Model
Domain Model
Services
Control
?

Client
Calls
Server

Where to put the parts if application distributed?

- The Remote Proxy design pattern!

Using a Graphics Framework



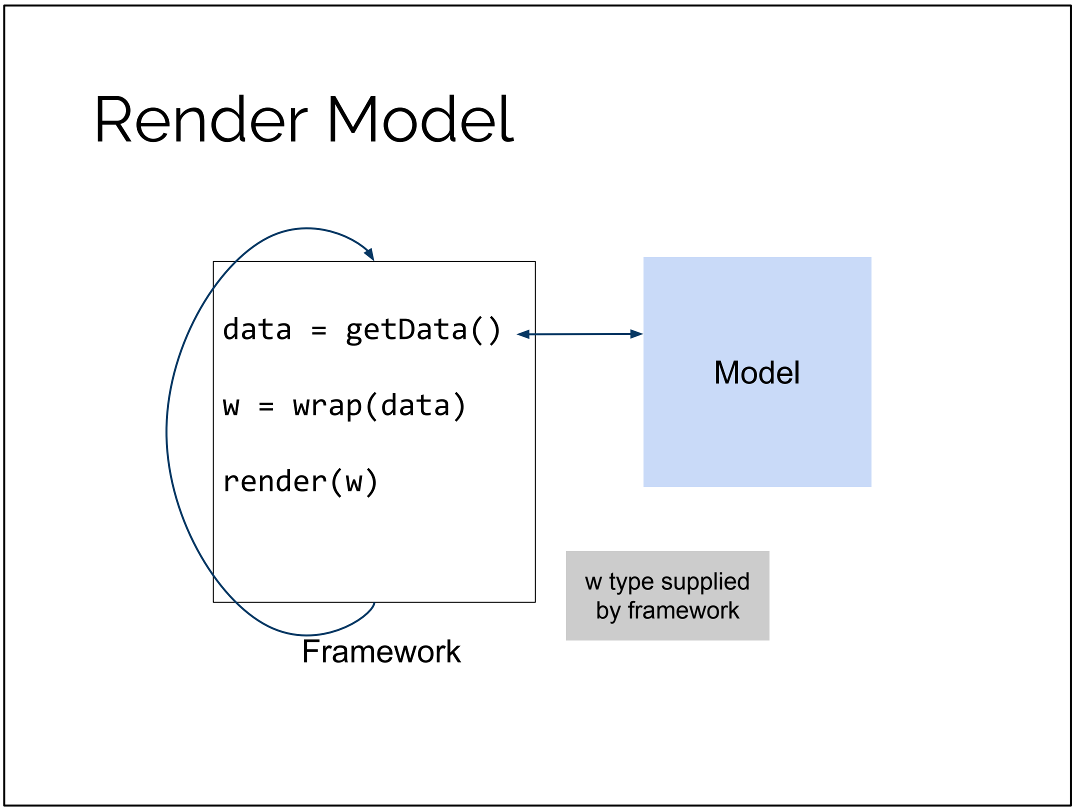
set(data)

data = getData()

Model

If using a graphics framework normally no full MVC design

- Mostly using a <u>pull</u> design (render ask model for data)
    - Vs. Observer, a push design!
- Control replaced by update game (method periodically called by framework)



Render Model

```
data = getData()

w = wrap(data)

render(w)
```

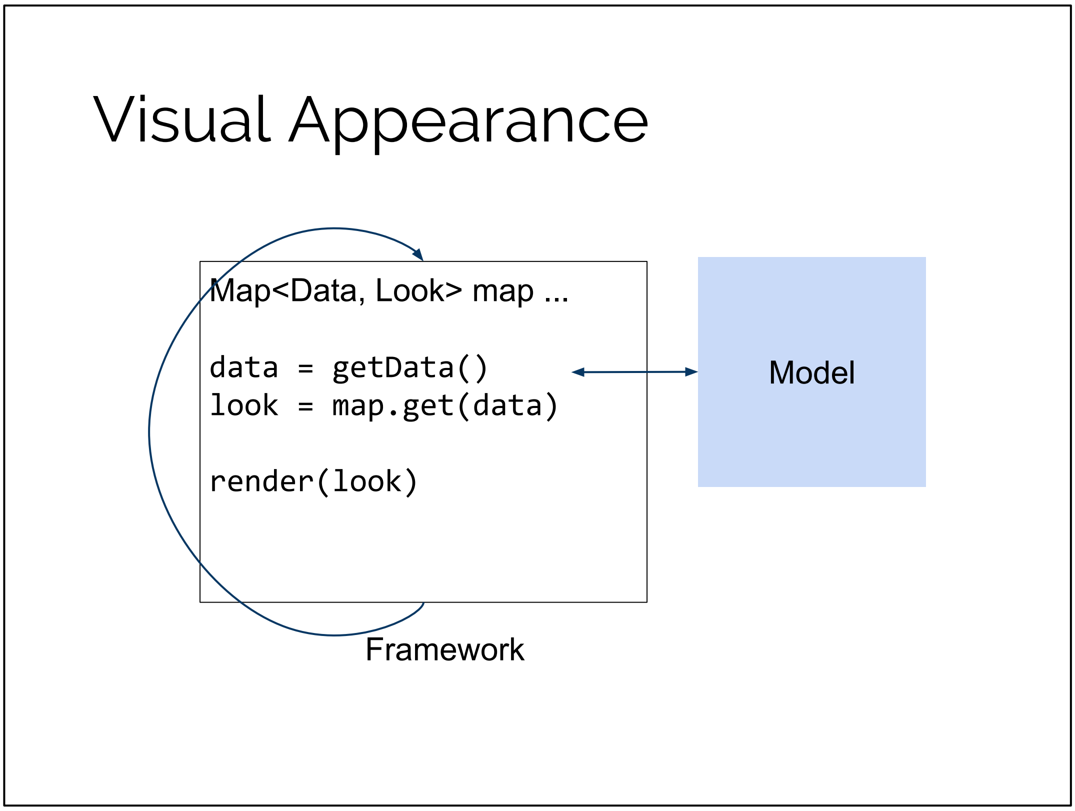
Model

w type supplied by framework

Framework

NO rendering data in model!
- I.e. no import of framework classes in model

If rendering (physics) handled by framework
- Wrap model data in framework classes
- Keep model clean!!!



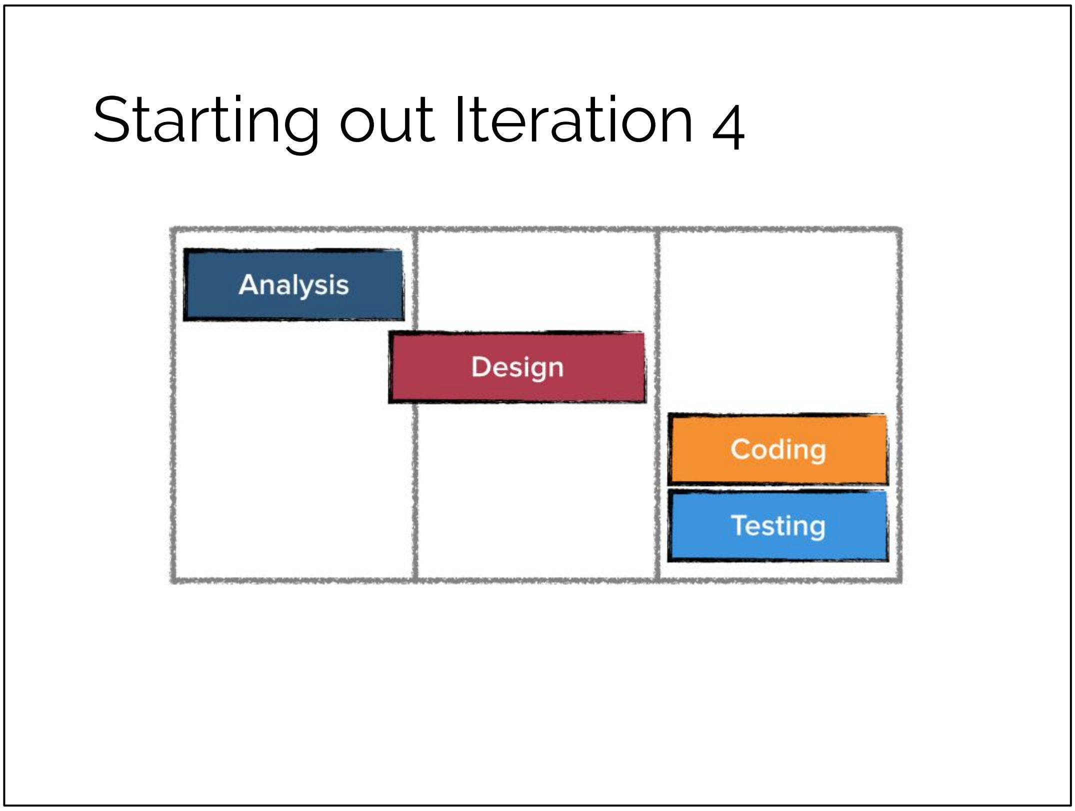
Visual Appearance

Map<Data, Look> map ...

```
data = getData()
look = map.get(data)

render(look)
```

NO visual attributes (icons, sprites, names of files) in model!

- Let framework, given the data, find the look!

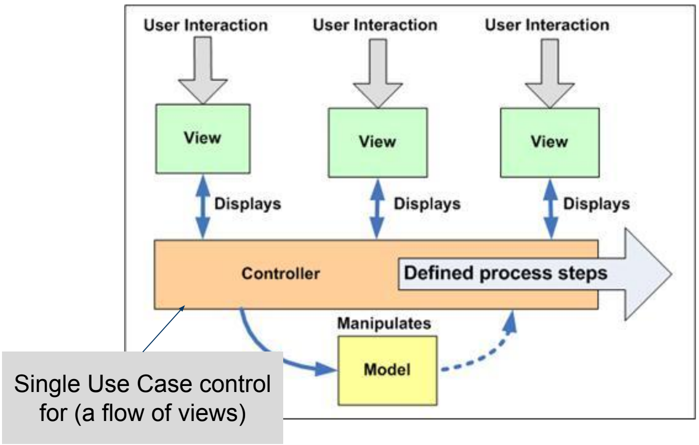


Starting out Iteration 4

Some final pieces!

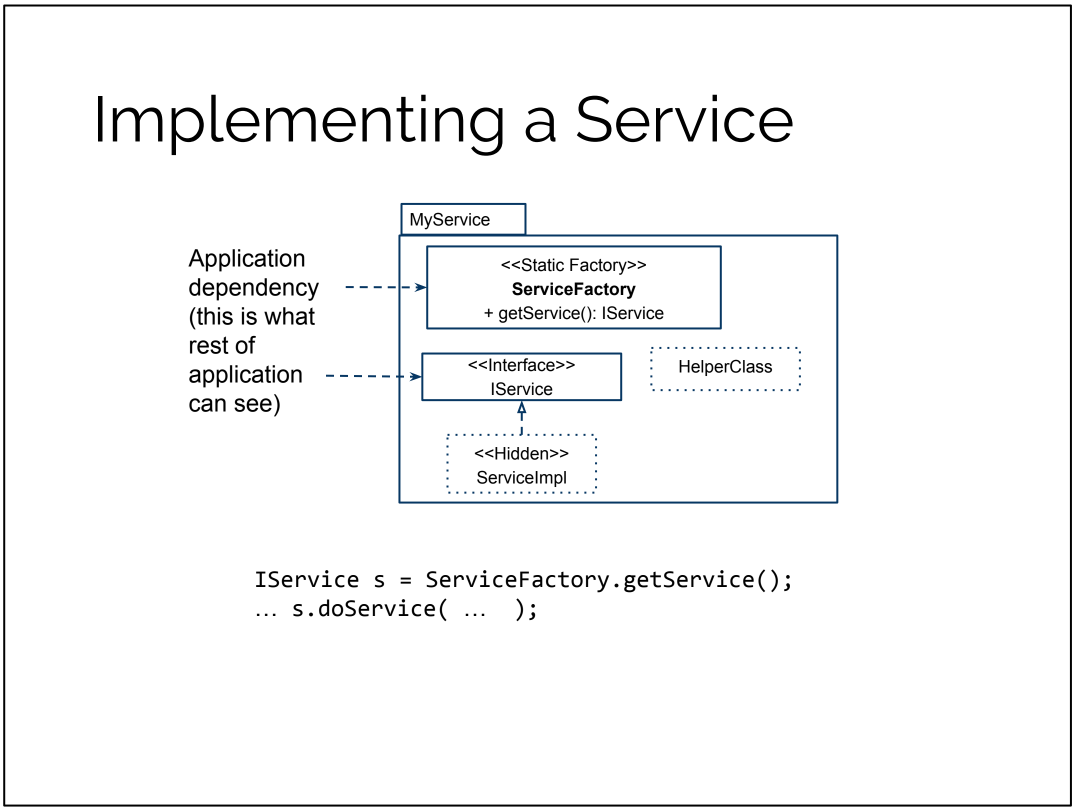
- Add a service
- Exception handling
- Making life easier, code injection and generation , ...

Use case Controllers



Single Use Case control for (a flow of views)

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)



Implementing a Service

Application dependency (this is what rest of application can see)

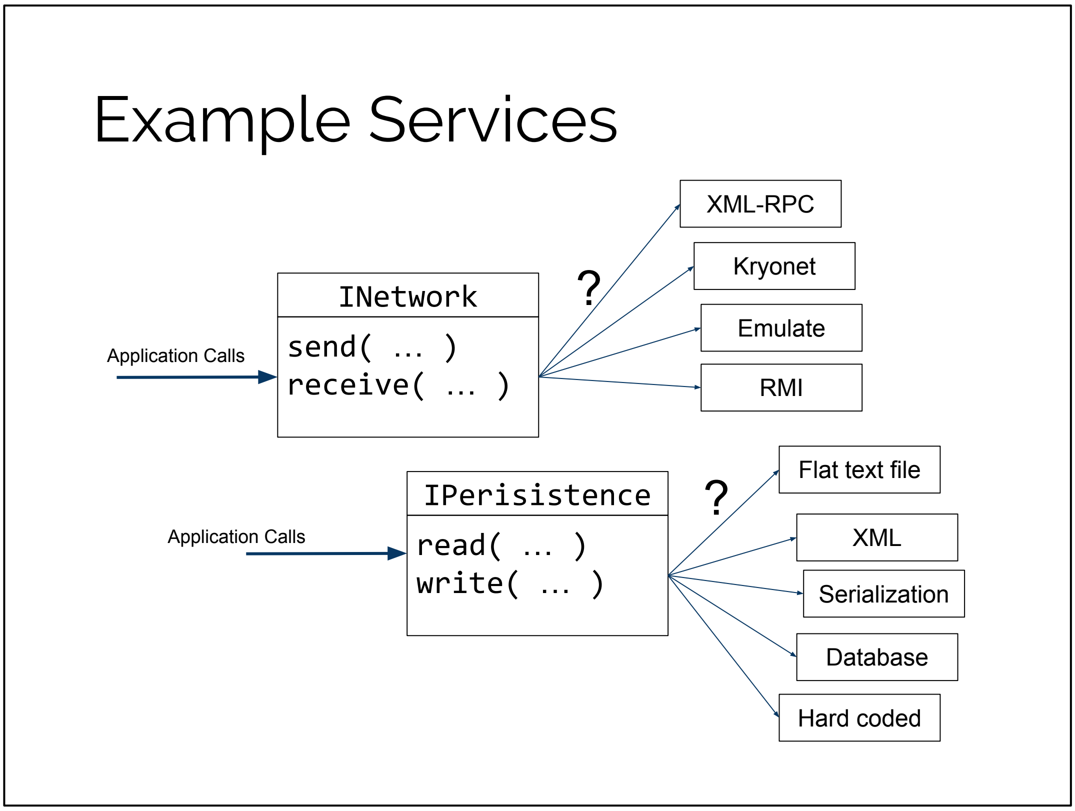
```
IService s = ServiceFactory.getService();
… s.doService( …  );
```

Services implemented using Facade pattern

- I.e. an interface used by control and a Factory to get an implementing object
- All other classes are package private (i.e. no public)
    - Possibly pure data classes implemented as immutable value objects
- For application (control layer) to find a service possibly use the Service locator design pattern
- If problems with dependencies use layering inside service
- Use of generics may remove dependencies

ALSO: Often need to decide on format for data

- Try to shield application from changes in data formats!



Example Services

Application Calls

INetwork

send(…)
receive(…)

?

XML-RPC

Kryonet

Emulate

RMI

Application Calls

IPerisistence

read(…)
write(…)

?

Flat text file

XML

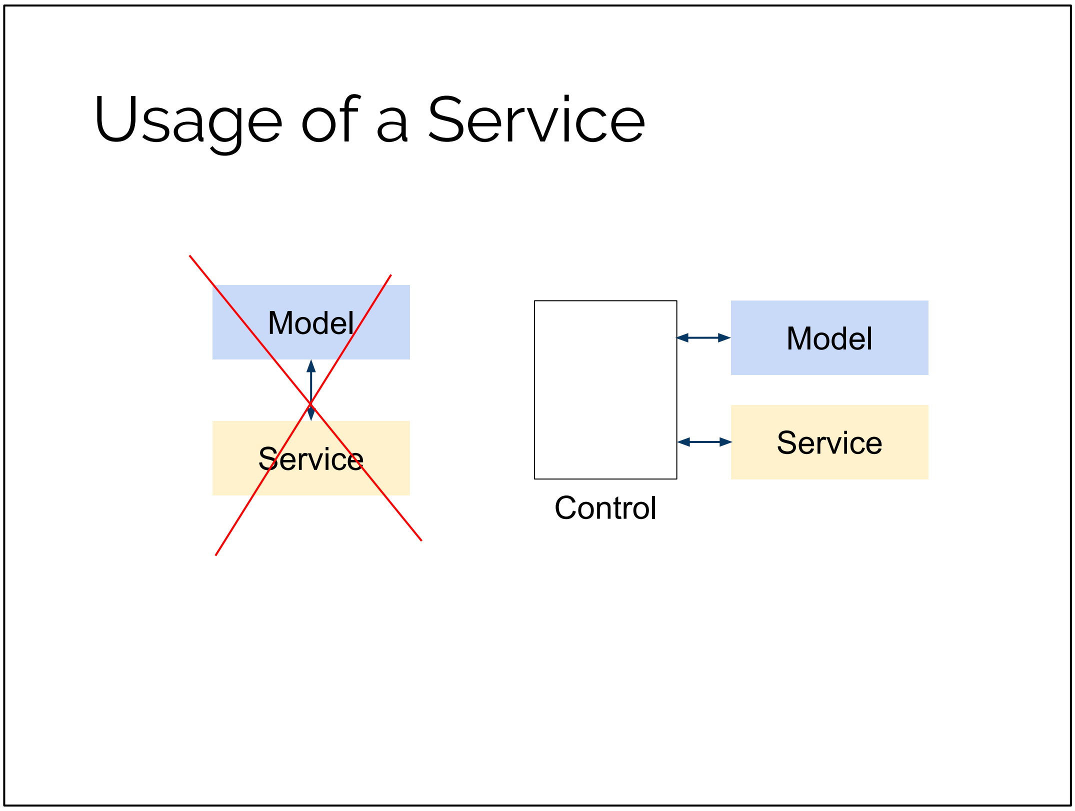
Serialization

Database

Hard coded

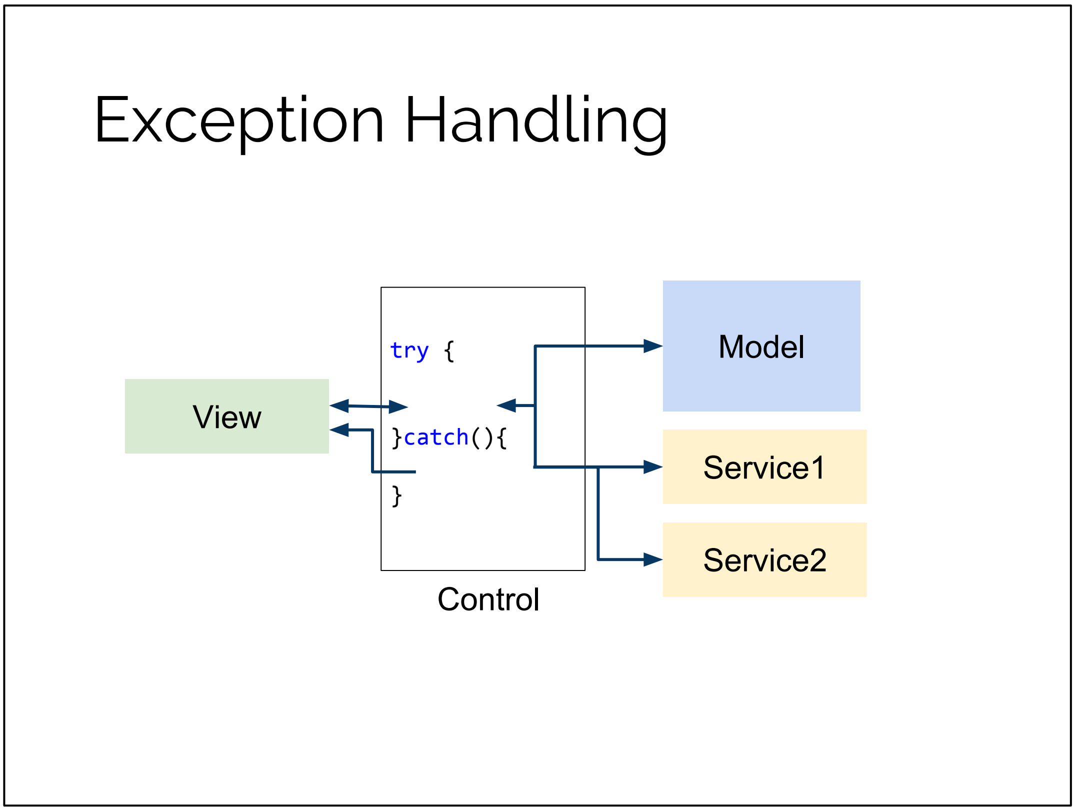
Any service is accessed via an interface (INetwork or IPersistence)!

- Exact implementation technique never exposed to application
  - Also hide data formats
- Exact implementation technique, is a technical detail, not overly interesting for us
- Interfaces is a crucial part of application design!



Usage of a Service

Again: We don't want to clutter the model!
- No service code in model
- Use a controller
    - Get data from model and shuffle to service or ...
    - Get data from service set in model



Exception Handling

```
try {
}catch(){
}
```

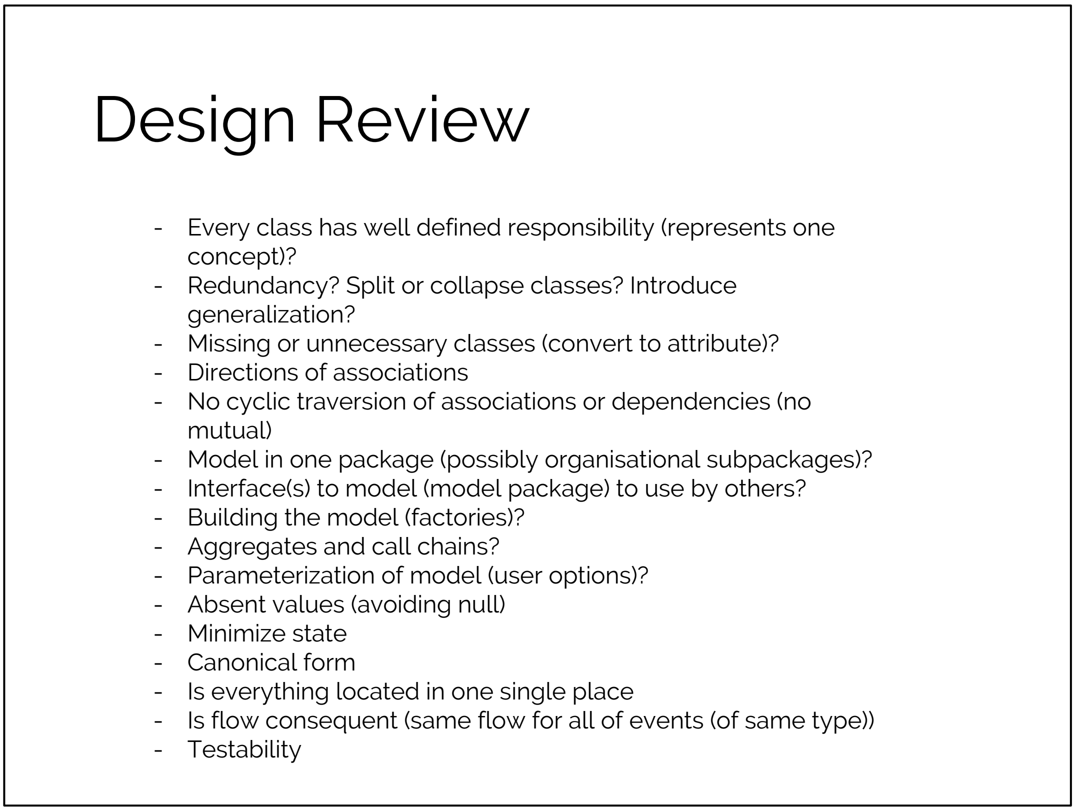
Exceptionhandling not well understood subject
- This is an advice

Exceptions may come from Model or Services
- Model or Services called from control …
  - Model never call service directly (except eventbus)
- Handle exceptions in control …
- … propagate message to view to inform user

Possibly create high level exception classes if many layers in application
- Also exception tunneling (Java specific)

MP : Monopoly-0.4

Download from course page, inspect and run!



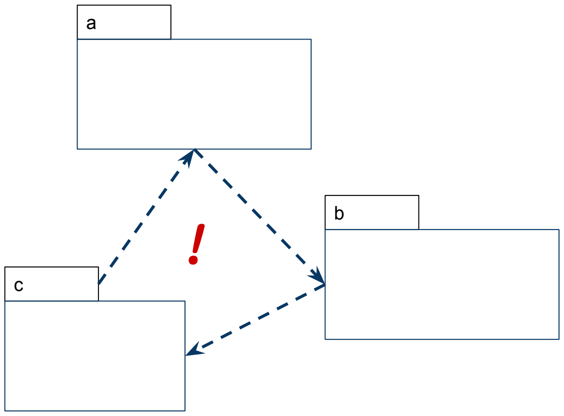
Design Review

- Every class has well defined responsibility (represents one concept)?
- Redundancy? 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!
- Use tools!

Circular Dependencies



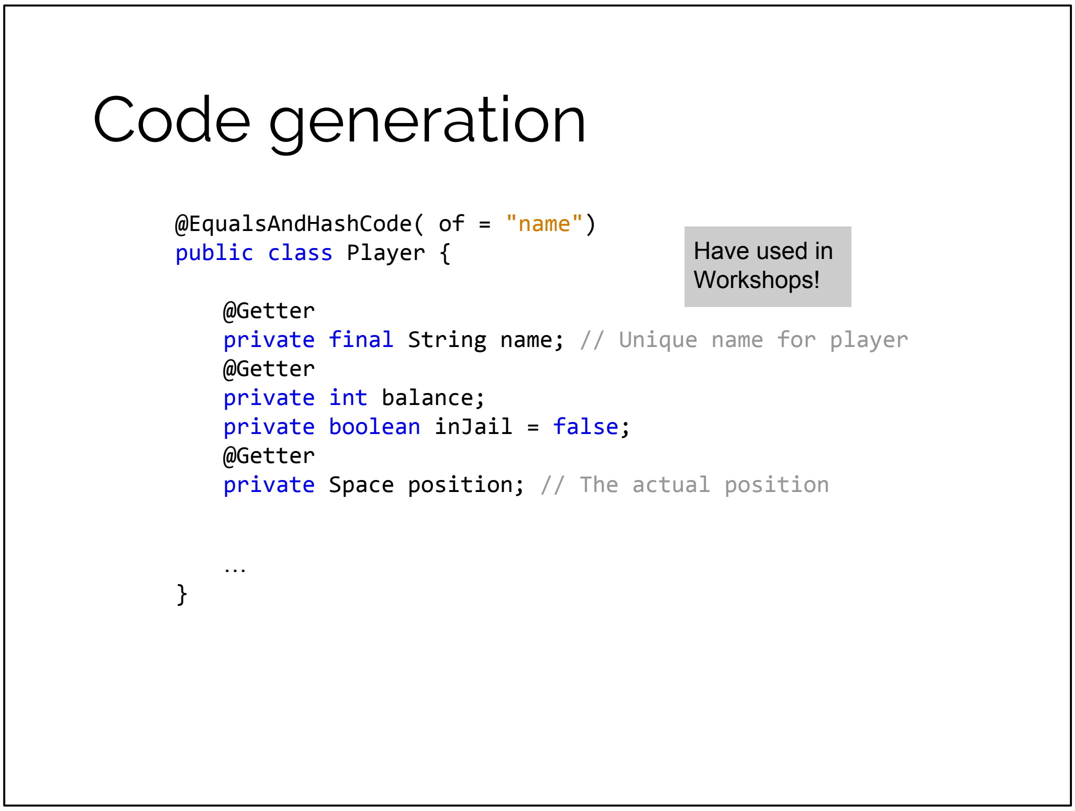
Circular dependencies between packages

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

Quality Tools

JDepend
stan4j.com
STAN
Structure Analysis for Java
pmd
DON'T SHOOT THE MESSENGER
Powered by
JACOCO
Java Code Coverage
TM
FindBugs

Use tools to increase design and code quality!
- Some built in to IDE's
- See web!
- Possible to incorporate into pom.xml (Maven project)



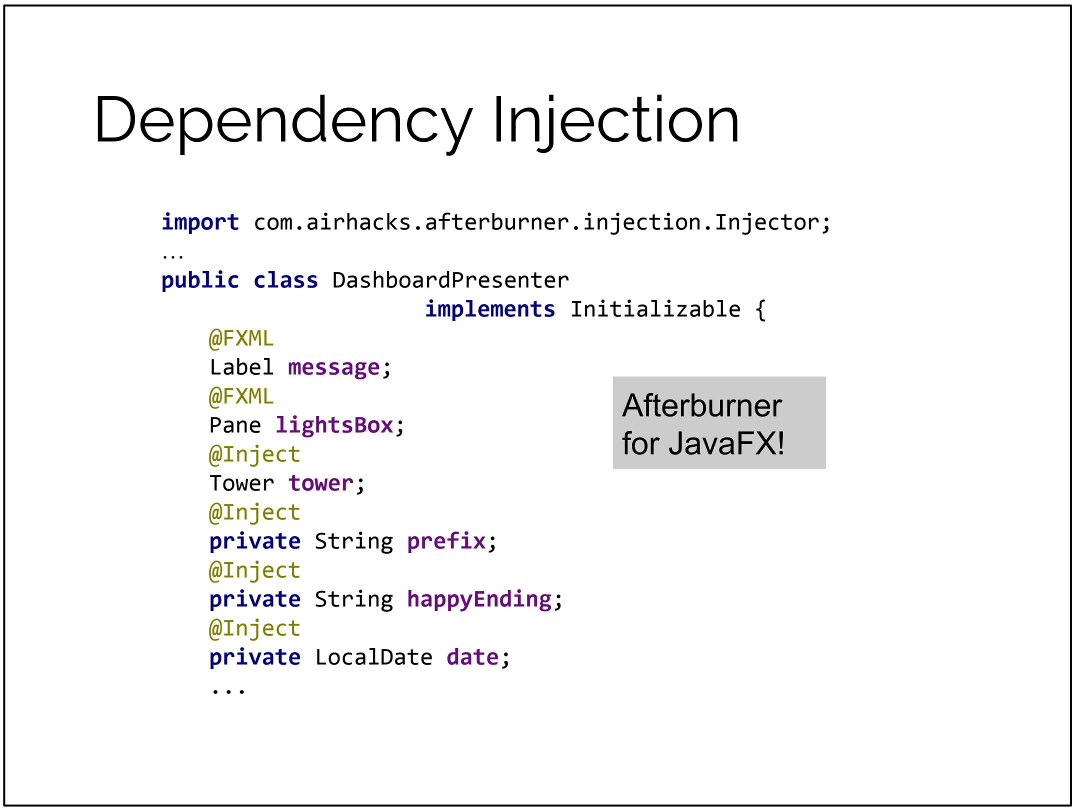
Code generation

```
@EqualsAndHashCode( of = "name")
public class Player {

    @Getter
    private final String name; // Unique name for player
    @Getter
    private int balance;
    private boolean inJail = false;
    @Getter
    private Space position; // The actual position


    …
}
```

Have used in Workshops!

Tiresome and boring to write “boilerplate” code

- We use Lombok for now.
    - Add Maven dependency
- Add “Annotations” to generate setter/getter/ and more ….
- Easy to add to to project thanks to Maven



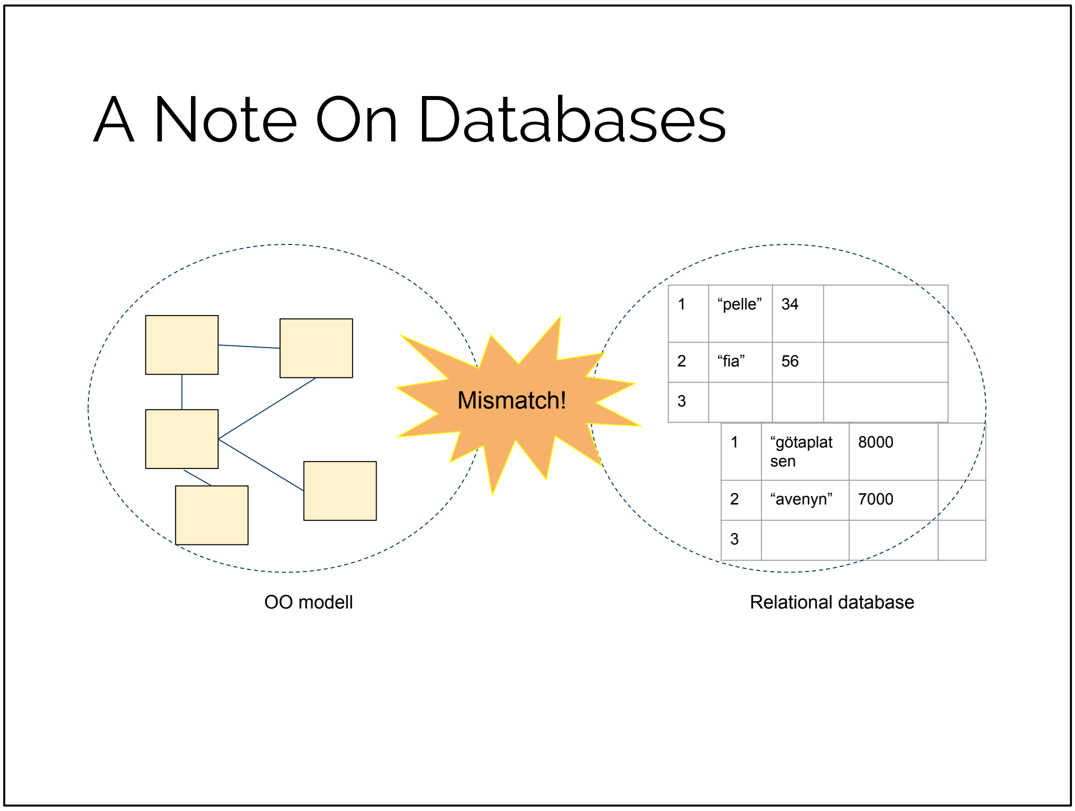
Dependency Injection

```
import com.airhacks.afterburner.injection.Injector;
...
public class DashboardPresenter
                    implements Initializable {
    @FXML
    Label message;
    @FXML
    Pane lightsBox;
    @Inject
    Tower tower;
    @Inject
    private String prefix;
    @Inject
    private String happyEnding;
    @Inject
    private LocalDate date;
    ...
```

Afterburner for JavaFX!

Injection makes it possible to let the application supply instances as needed.
- We don't need to explicitly create (and connect) objects.
- Not used for model: Model constructed in single location.

[Structuring Complex JavaFX 8 Applications for Productivity]()



A Note On Databases

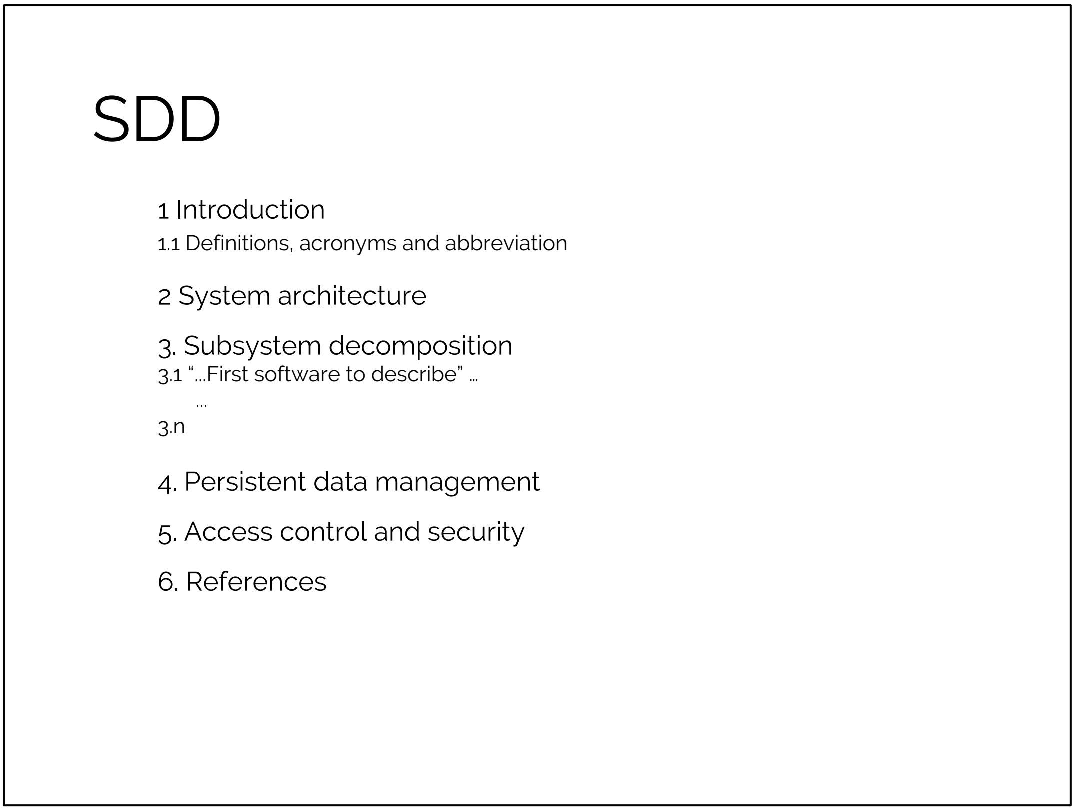
1	"pelle"	34	
2	"fia"	56	
3			

1	"götaplatsen	8000	
2	"avenyn"	7000	
3			

OO-models and relational databases hard (unsolved) problem

- OO model is a web of objects
- Database is primitive data in tables
- Object relational impedance mismatch
- Possibly : Use some ORM framework

Avoid using databases, emulate (use an interface)!



SDD

1 Introduction
1.1 Definitions, acronyms and abbreviation

2 System architecture

3. Subsystem decomposition
3.1 "...First software to describe" ...
...
3.n

4. Persistent data management

5. Access control and security

6. References

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.
- 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.
- <u>This is about communication, no absolute rules how to write</u>
- We prefer this top down explanation approach
    - Start out high level (big picture)...
        - Hardware setup, communication, applications involved (if applicable)
    - .. then refine in each step ...
        - Structure of (each) application
        - Packages
        - Possibly classes/Interfaces
        - <u>Design model</u>
    - ... until close to code (when reaching this level the code and the tests are the documentation)

Summary

Iteration 2, 3 and 4

- We got more UC's up and running!
- We got a full MVC version of the first UC's
- Implemented a Service
- Some exception handling

During the phases we continuously check design, and quality (using quality tools).

Next: Continue until finished ….