CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Object-oriented Programming Project

Implementation

Dr. Alex Gerdes

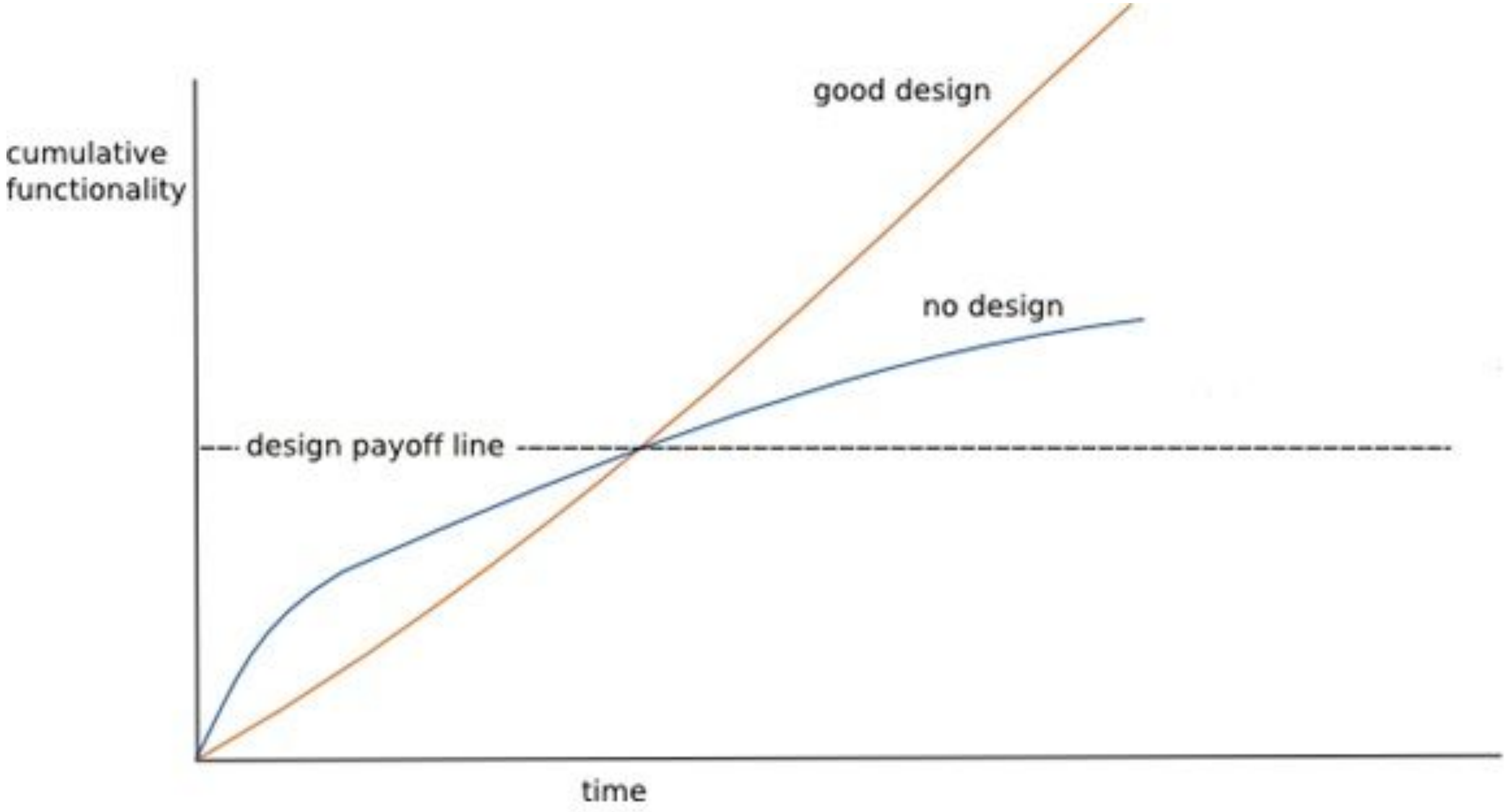
TDA367/DIT212 - HT 2018

Summary previous lecture

- Seminar
- Domain model -> design model -> implementation
- Test-driven development
- Sequence diagram
- Thinking high and low
- Monopoly: unit tests and Travis

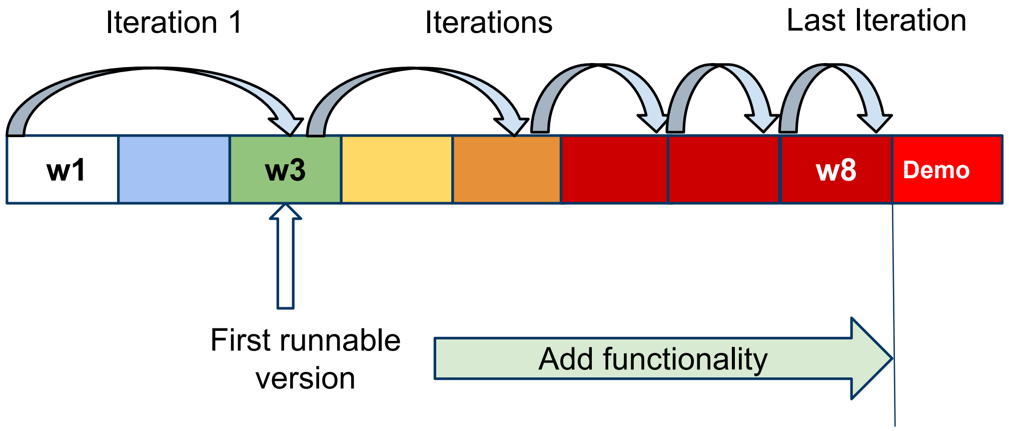
Technical debt

- The design is really important!
  - We design the model
  - We design the full application



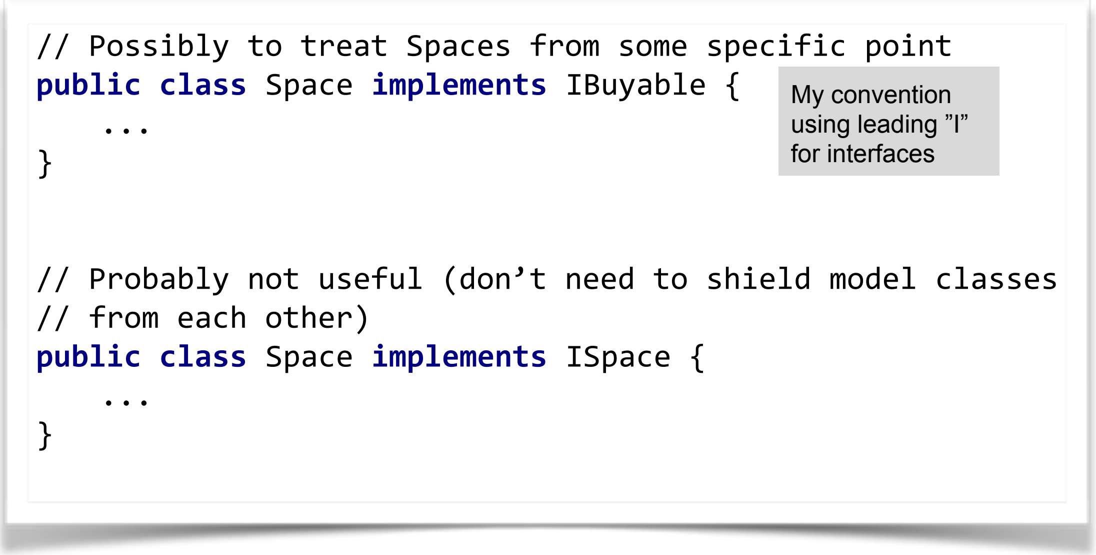
Next iteration (sprint)

- You should be (nearly) finished with the first iteration:
  - First version of RAD
    - Description of application
    - User stories
    - Domain model
  - Design model (will end up in SDD)
  - Implementation: able to run application and tests!
- Next iteration:
  - Refactor before you begin!
  - Revise User Stories (content, estimation, priority)
  - Choose new set of User Stories
  - Update models
  - etc.



Refactoring

Interfaces and abstraction

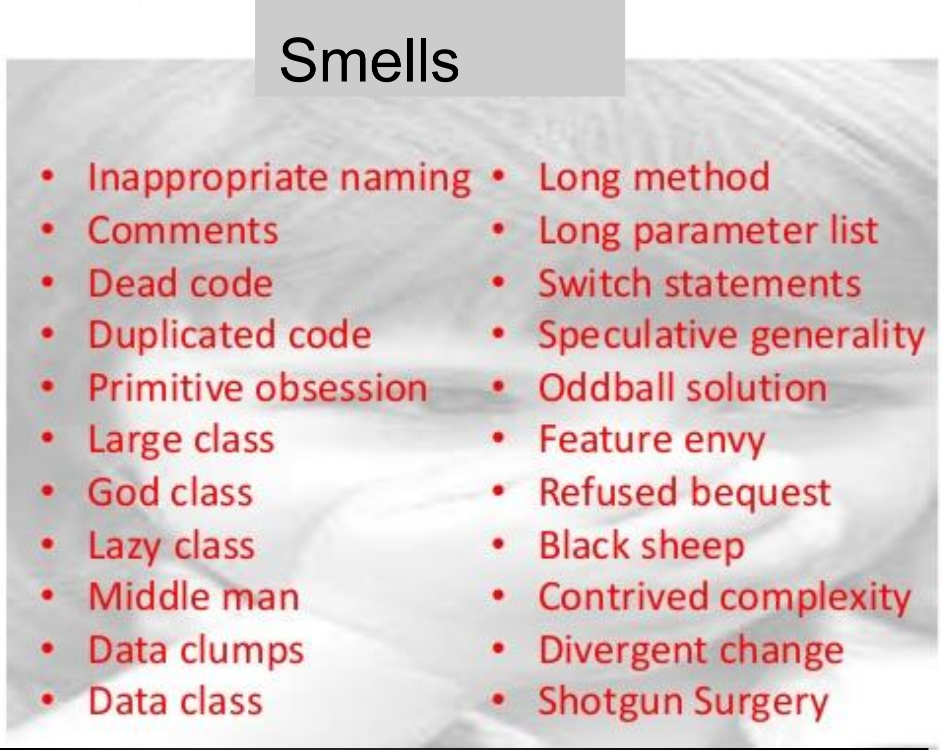


```
// Possibly to treat Spaces from some specific point
public class Space implements IBuyable {
    ...
}


// 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

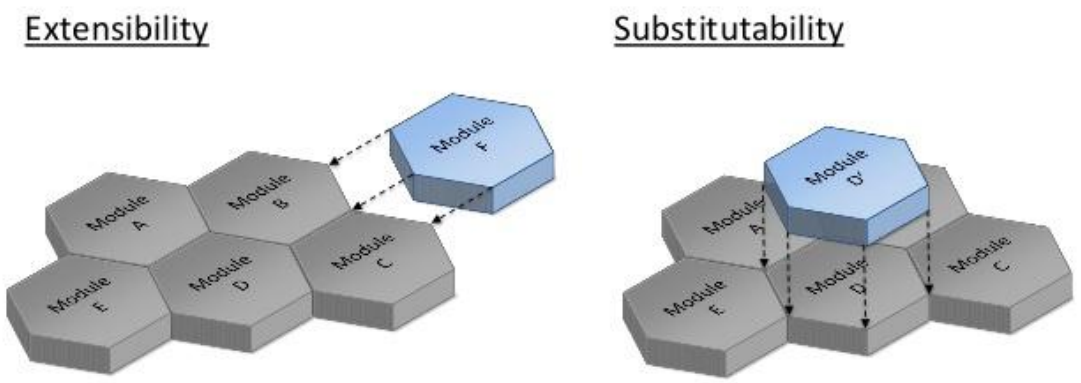
- Make objects “of the same type”
  - Guarantee certain operations are available
  - Possible to store heterogenous objects in Collections
- Isolate the model
  - Shield different parts of application
- Try to abstract, but don’t overdo! Abstract away from:
  - file formats
  - Storage systems

Implementation principles

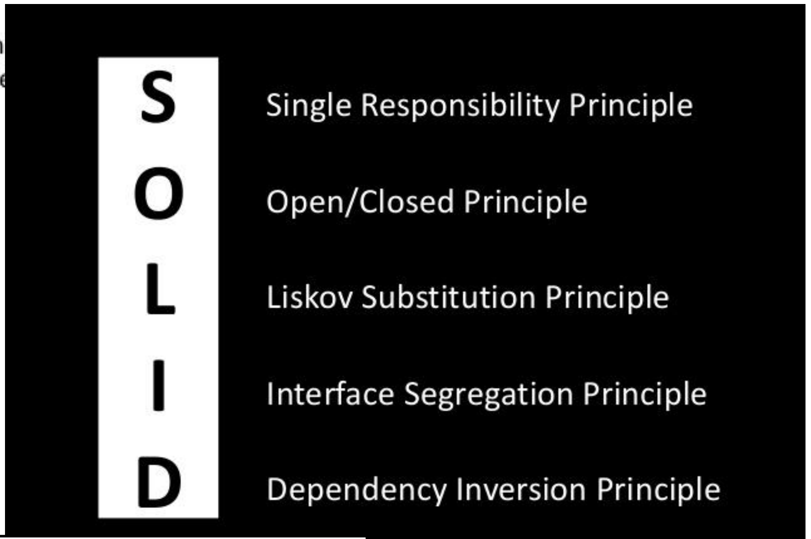


Smells

- 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



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



S Single Responsibility Principle

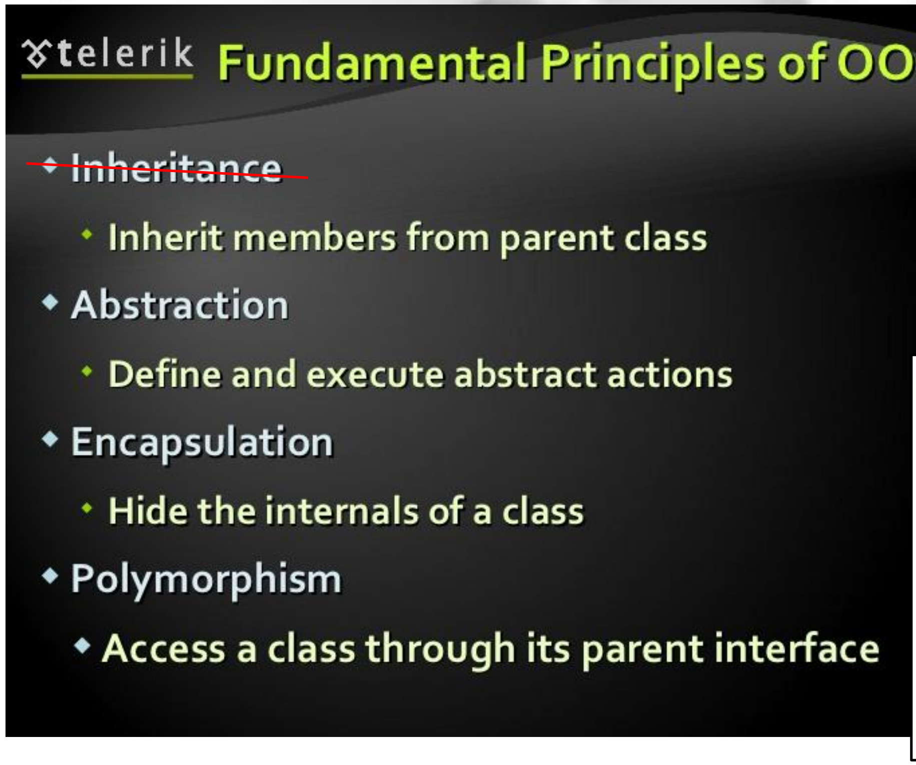
O Open/Closed Principle

L Liskov Substitution Principle

I Interface Segregation Principle

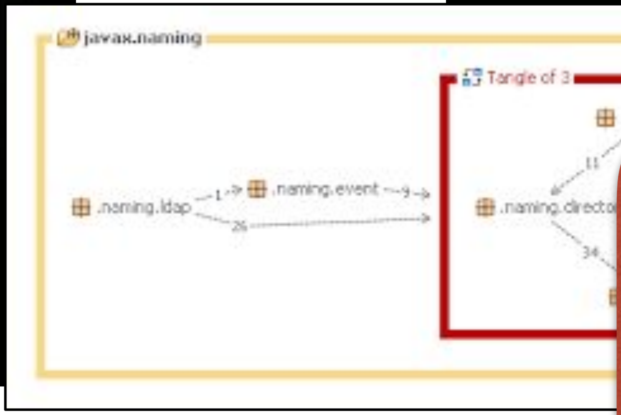
D Dependency Inversion Principle

:lebob.com/ArticleS.UncleBob.PrinciplesOfOod



telerik **Fundamental Principles of OOP**

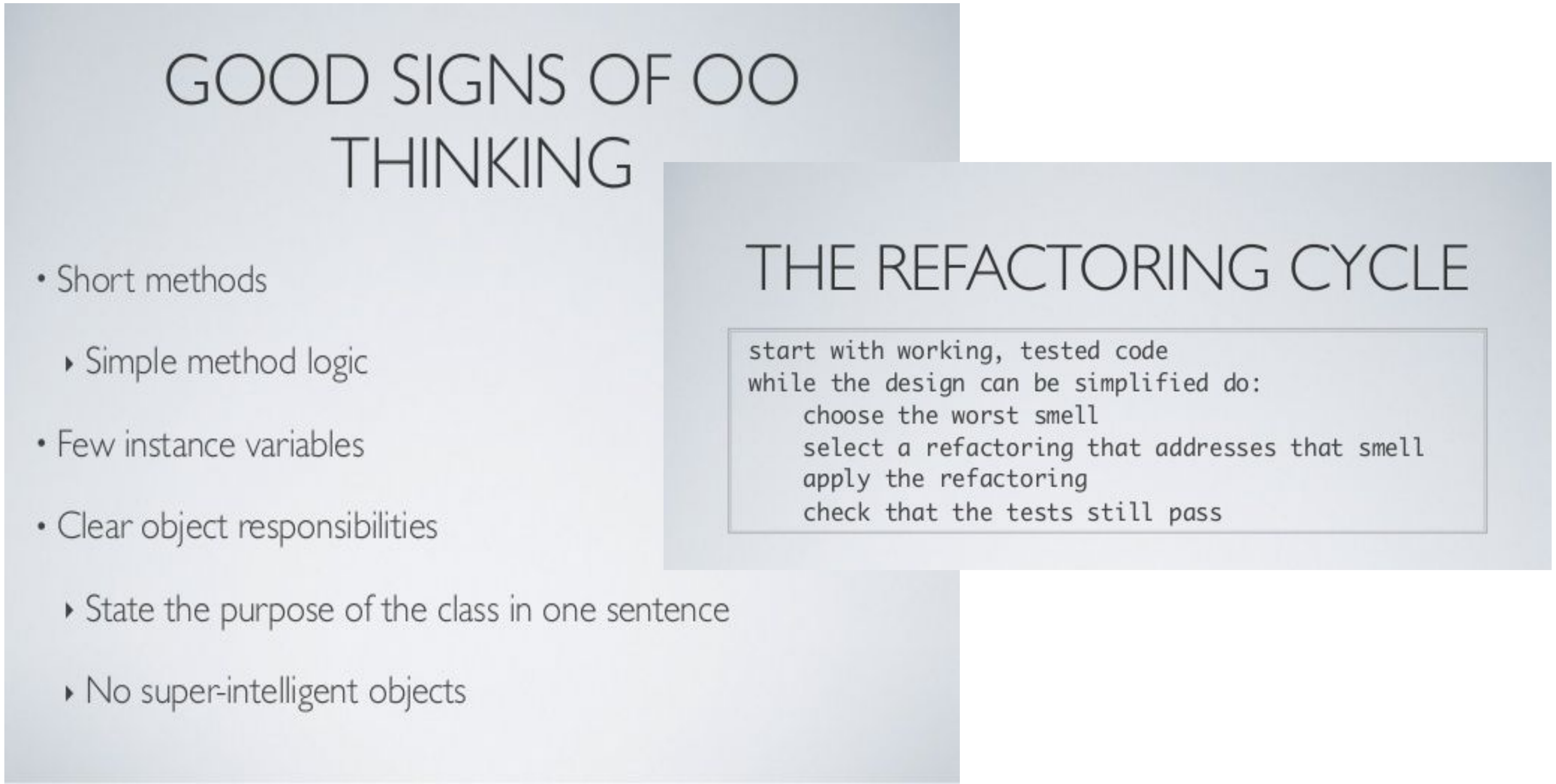
- ~~Inheritance~~
  - Inherit members from parent class
- Abstraction
  - Define and execute abstract actions
- Encapsulation
  - Hide the internals of a class
- Polymorphism
  - Access a class through its parent interface



Check this during code review / reflection

Refactoring

- Refactor after each iteration
  - Check against implementation principles
  - Keep functionality the shame
  - Run tests! (regression testing)



GOOD SIGNS OF OO THINKING

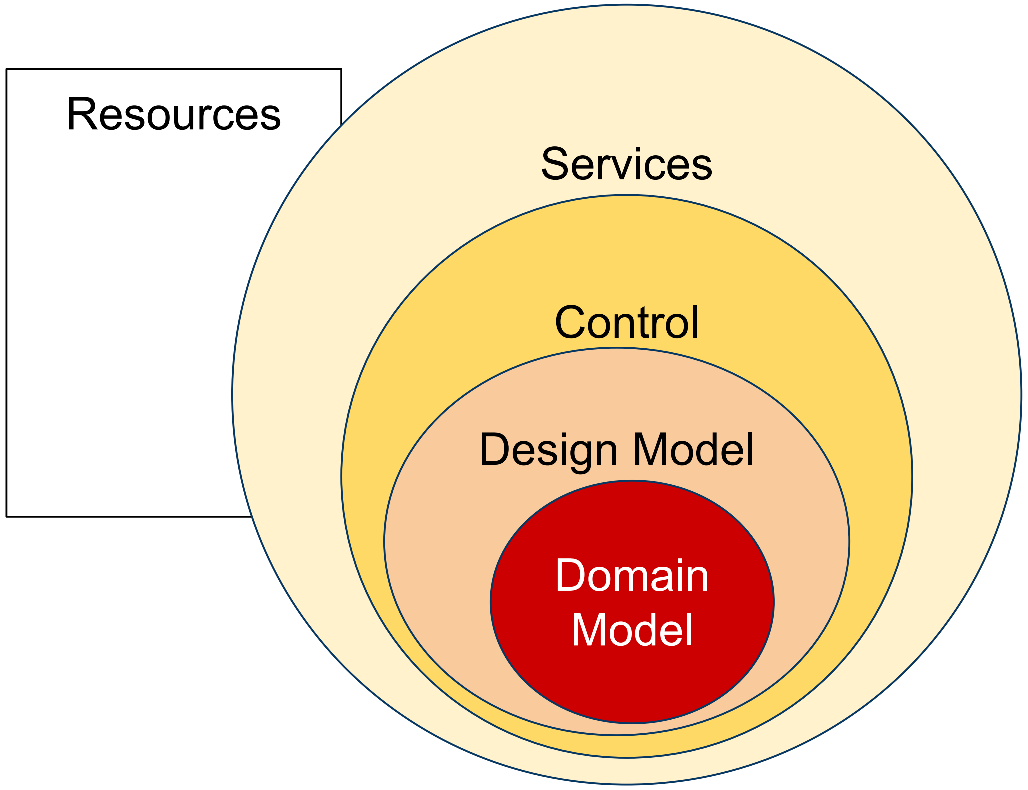
- Short methods
  - Simple method logic
- Few instance variables
- Clear object responsibilities
  - State the purpose of the class in one sentence
  - No super-intelligent objects

THE REFACTORING CYCLE

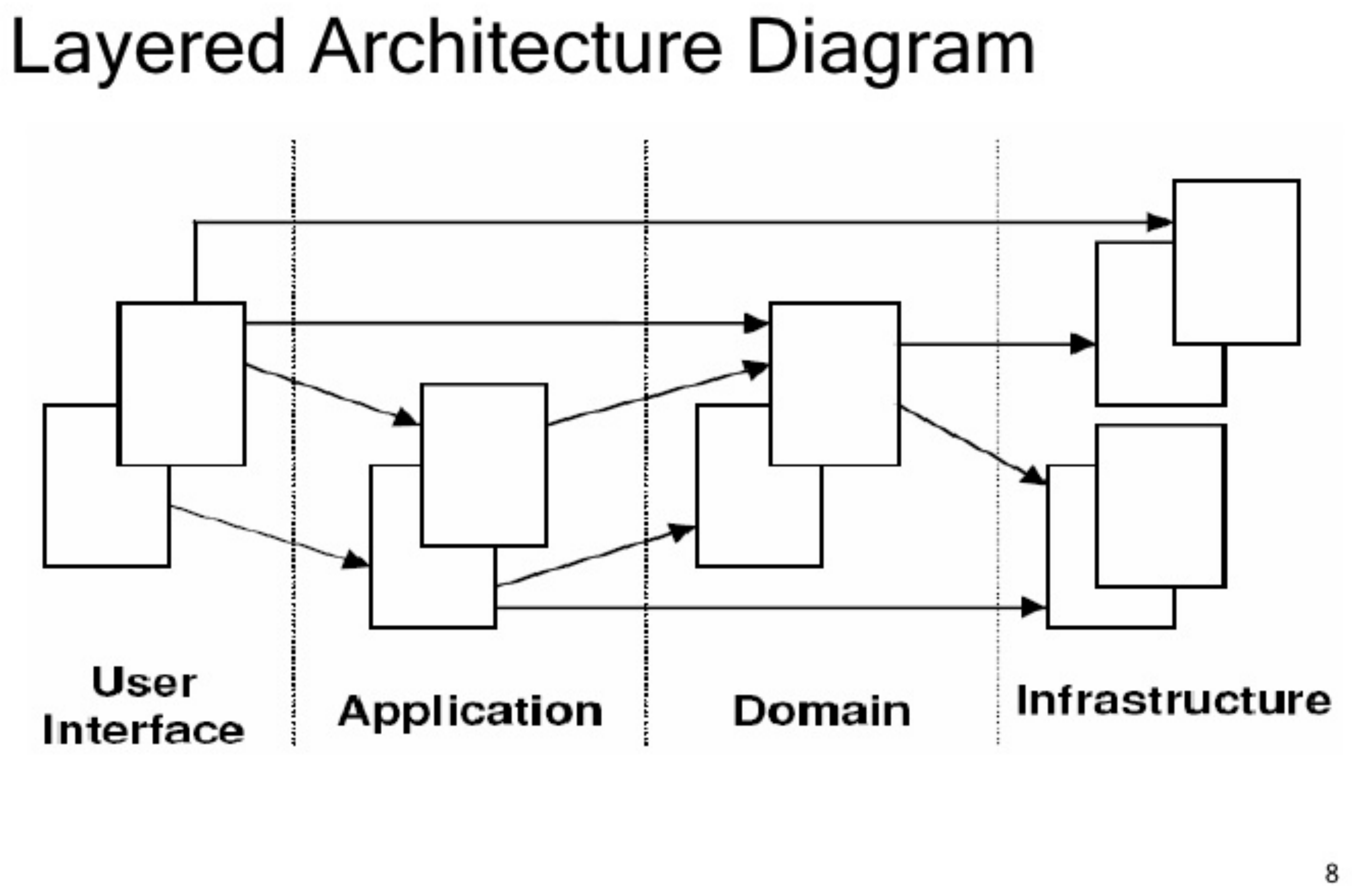
```
start with working, tested code
while the design can be simplified do:
    choose the worst smell
    select a refactoring that addresses that smell
    apply the refactoring
    check that the tests still pass
```

Application design

- Domain model contains the core classes from the analysis
- Design model is the domain model adapted for implementation
  - Extended with “technical”-support classes
- Control is a layer coordinating the flow between the model and services
- Services are everything supporting the model
  - GUI
  - Handling of resources
  - Persistence (save to file, database)
  - Communication (network, ...)
- Resources
  - Data for configuration, initialization, ...
  - Images, sounds, ...
  - Internationalisation data



Application design

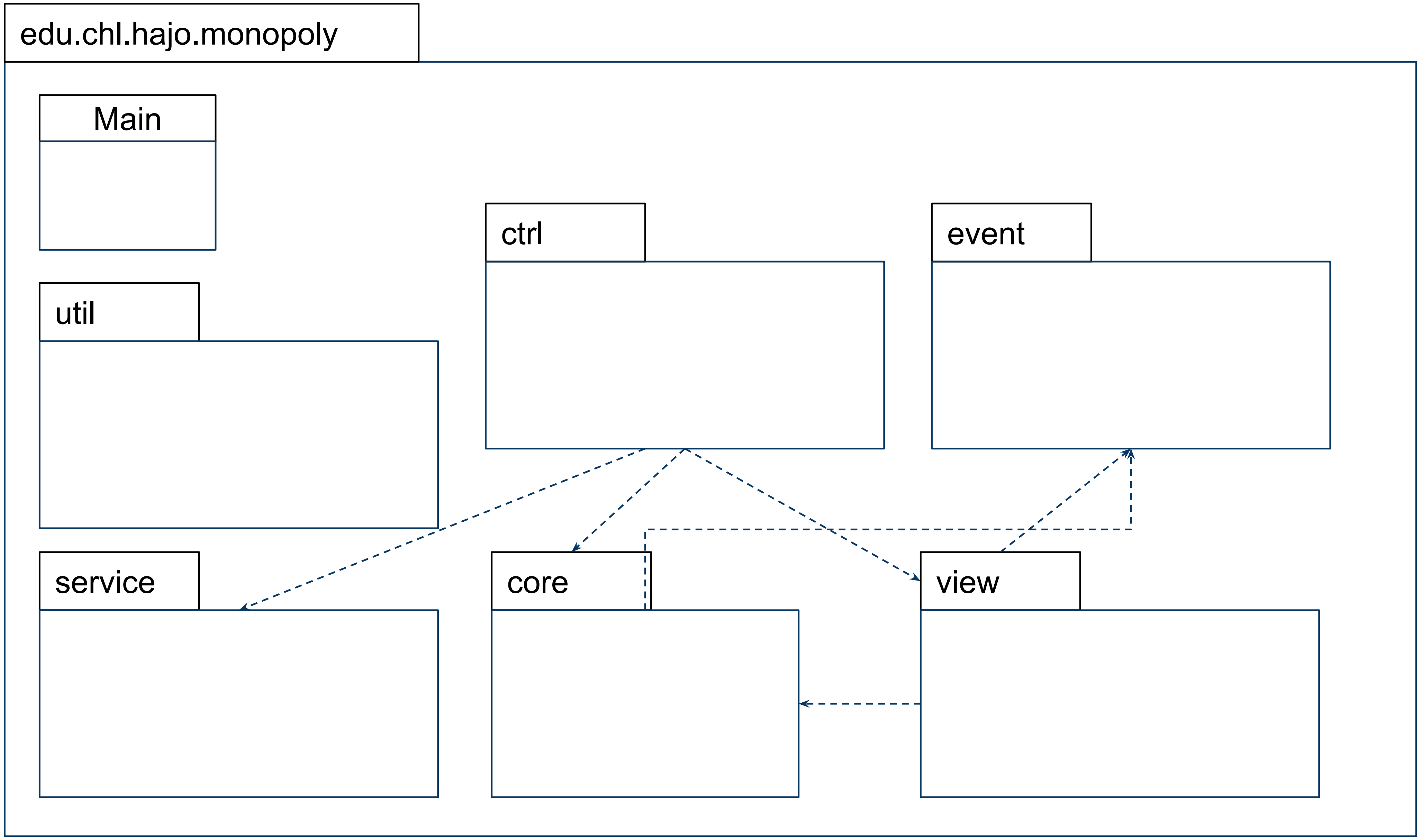


Layered Architecture Diagram

8

src: https://www.slideshare.net/srinip/domain-driven-design-development-spring-portfolio

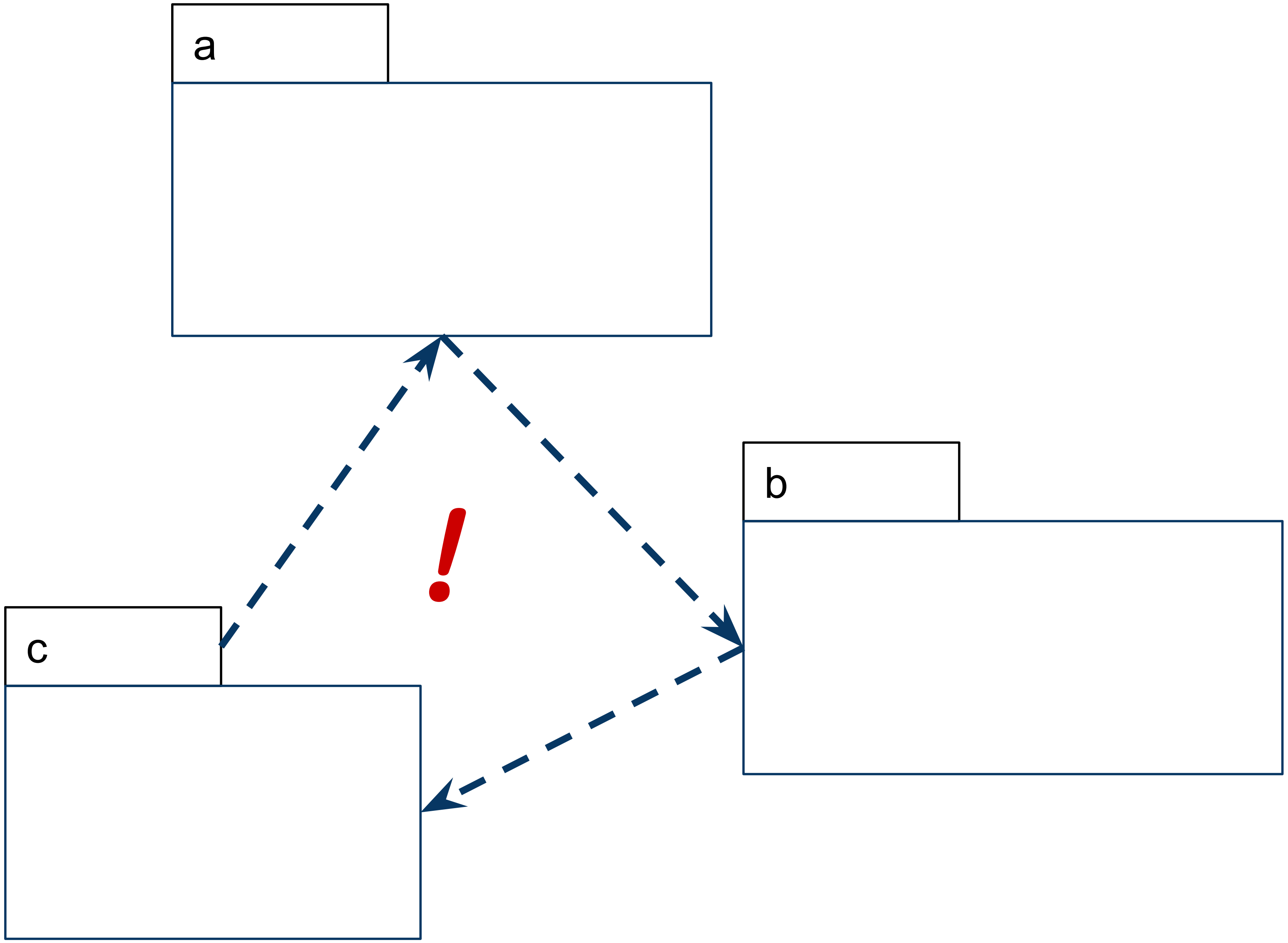
Package structure



Package structure

- Application should be partitioned into packages
  - Organises the overall structure of application
  - Each package should have a well defined purpose (same as classes, methods)
- Arrows show *dependencies*
  - `util` and `config` used by many but uses NONE (only incoming arrows)
  - Arrows for `util` and `config` not shown, would clutter up
- Model not dependent on services (used via `ctrl`)
- Package structure should guarantee unique qualified class names
- Use UML package diagram

Circular dependencies



Analysis tools

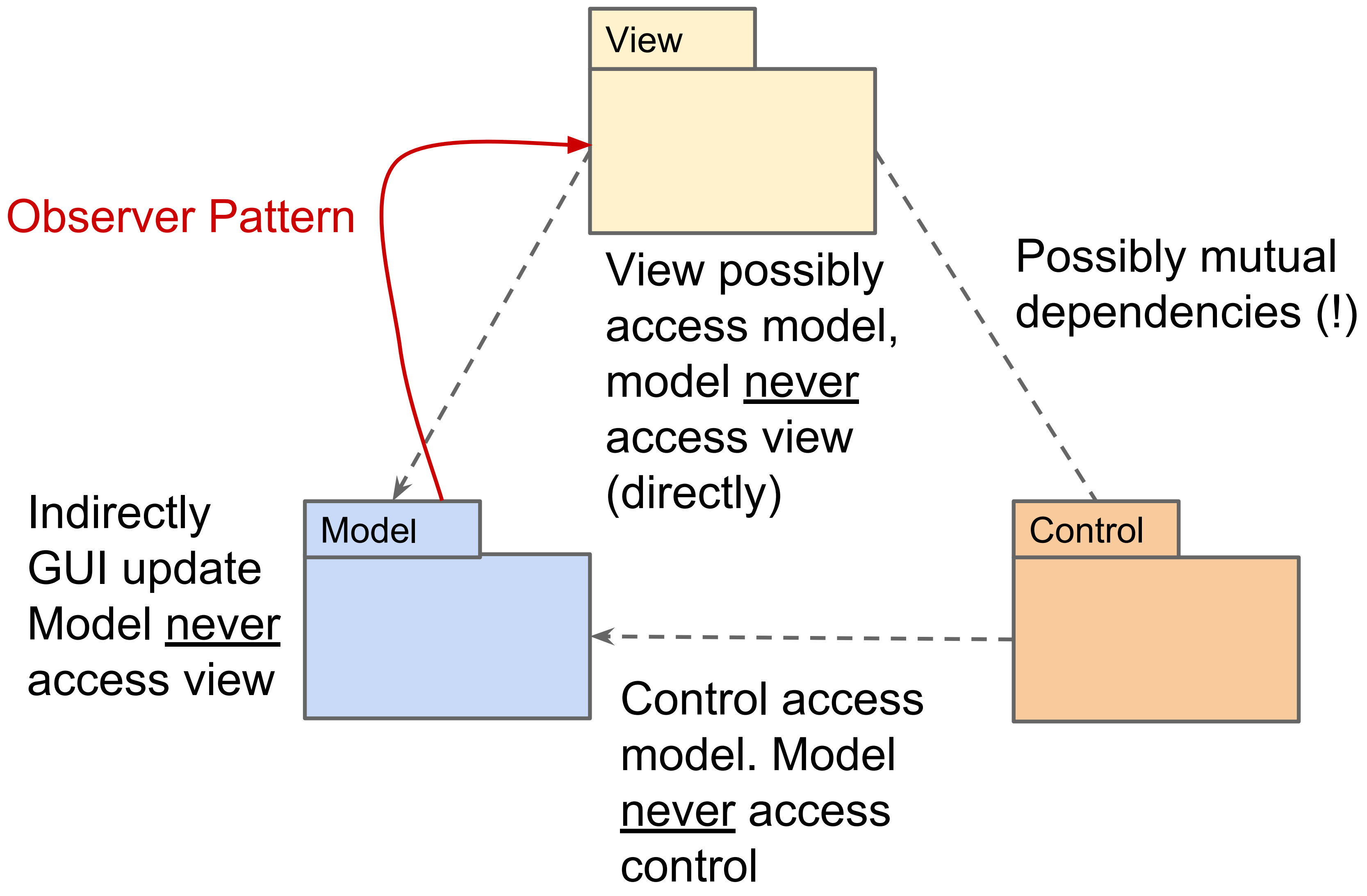
- Use tools to increase design and code quality!
  - Some built in to IDE’s
  - See web!
  - Possible to incorporate into pom.xml (Maven project)

Design review

- Every class has well defined responsibility (represents one concept)?
- Redundancy? Split or collapse classes? Introduce generalisation?
- Missing or unnecessary classes (convert to attribute)?
- Directions of associations
- No cyclic traversal 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)
- Are unit tests in place for the entire model?
- Is everything located in one single place?

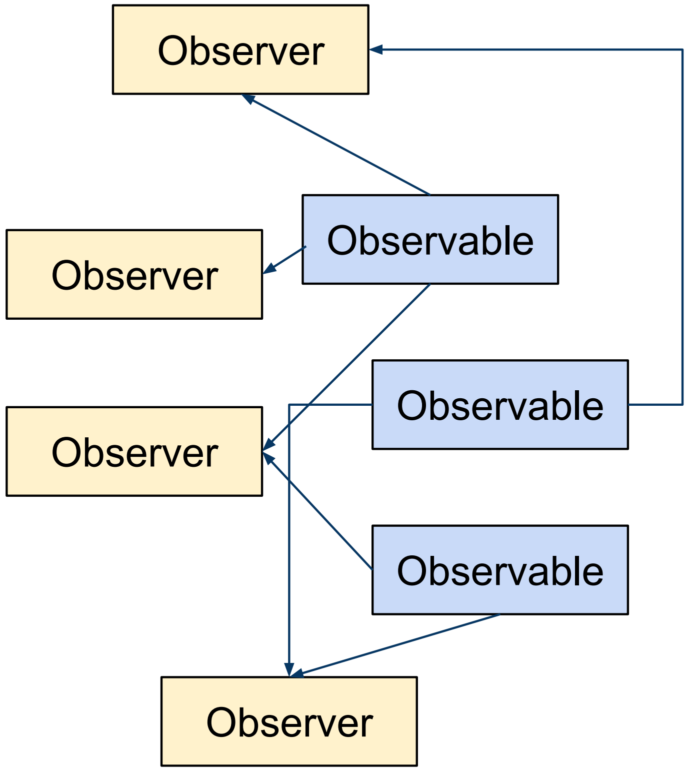
MVC implementation

MVC design review

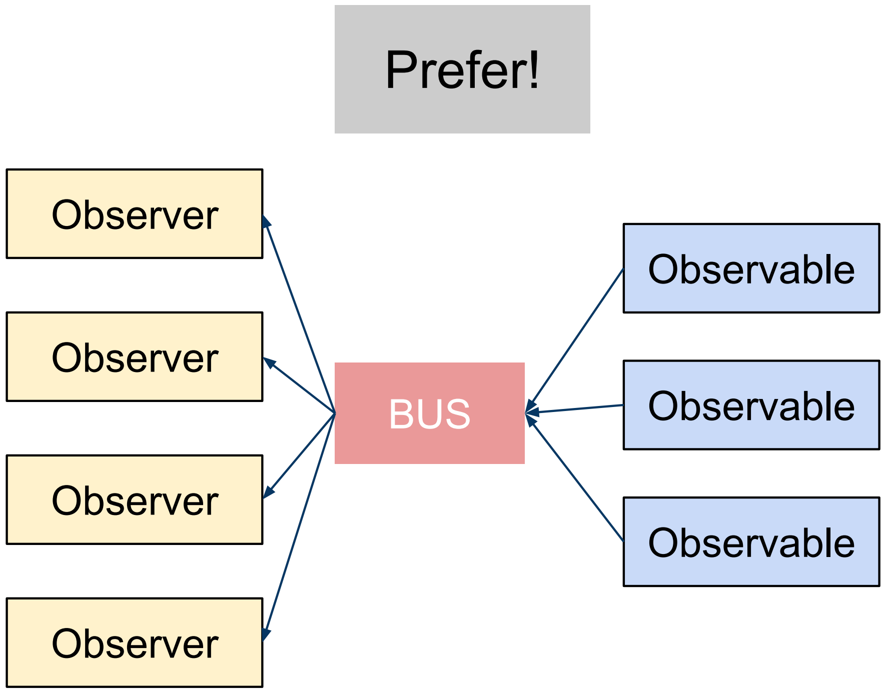

View
Observer Pattern
View possibly access model, model never access view (directly)
Possibly mutual dependencies (!)
Indirectly GUI update Model never access view
Model
Control
Control access model. Model never access control

- Different opinions about MVC structure
- This is a *push* design (vs. pull design) when using an *Observer* pattern

Observer design choices



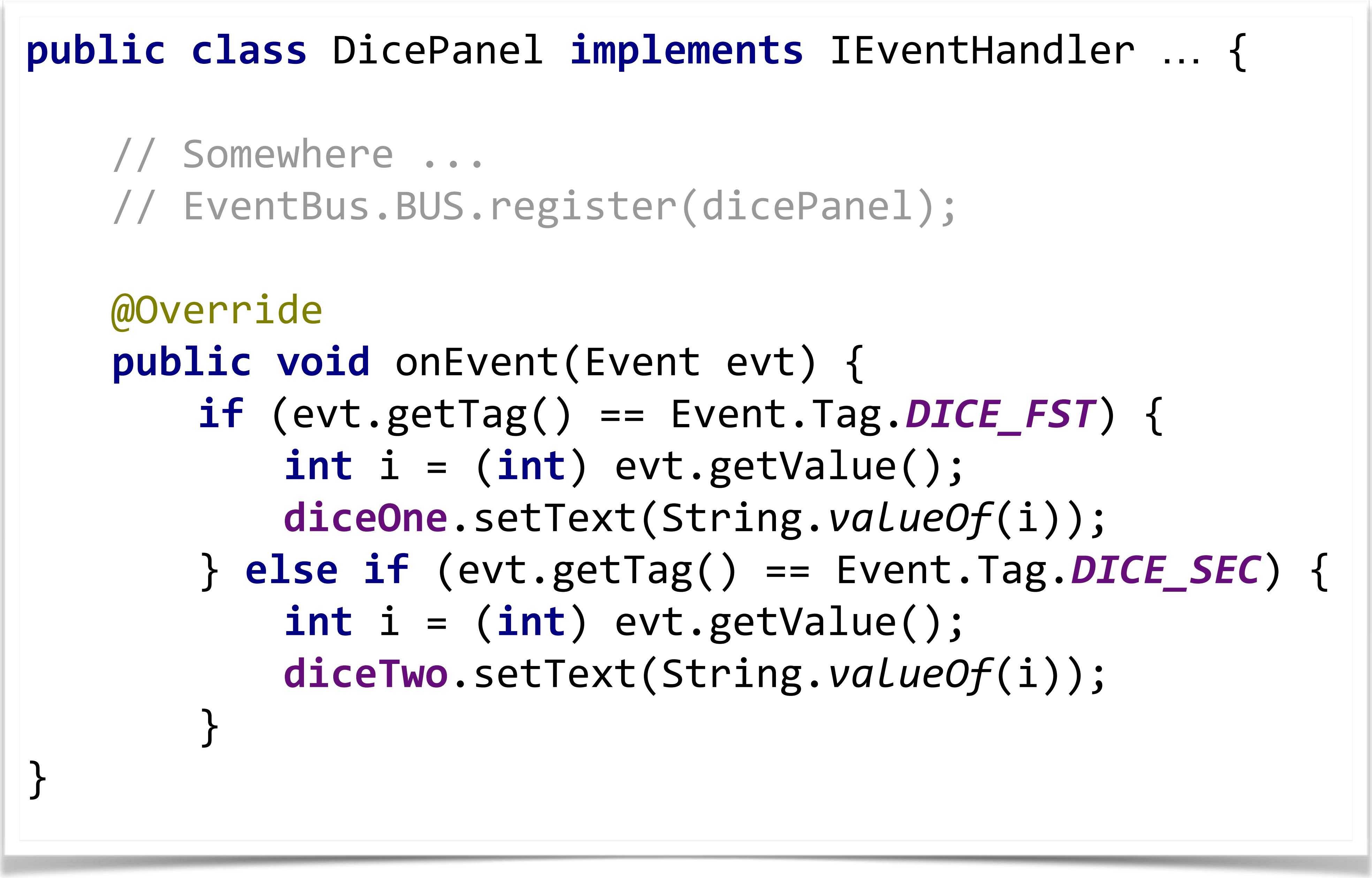
Ad hoc Observer



Observer using Event bus

- An alternative implementation of the observer pattern is an *event bus*
  - The bus is interface to model (along with types of messages)
  - Observables publish events
  - Observers register as event handlers
  - All events pass through the bus, easy to inspect/log events

Implementing `EventBus`

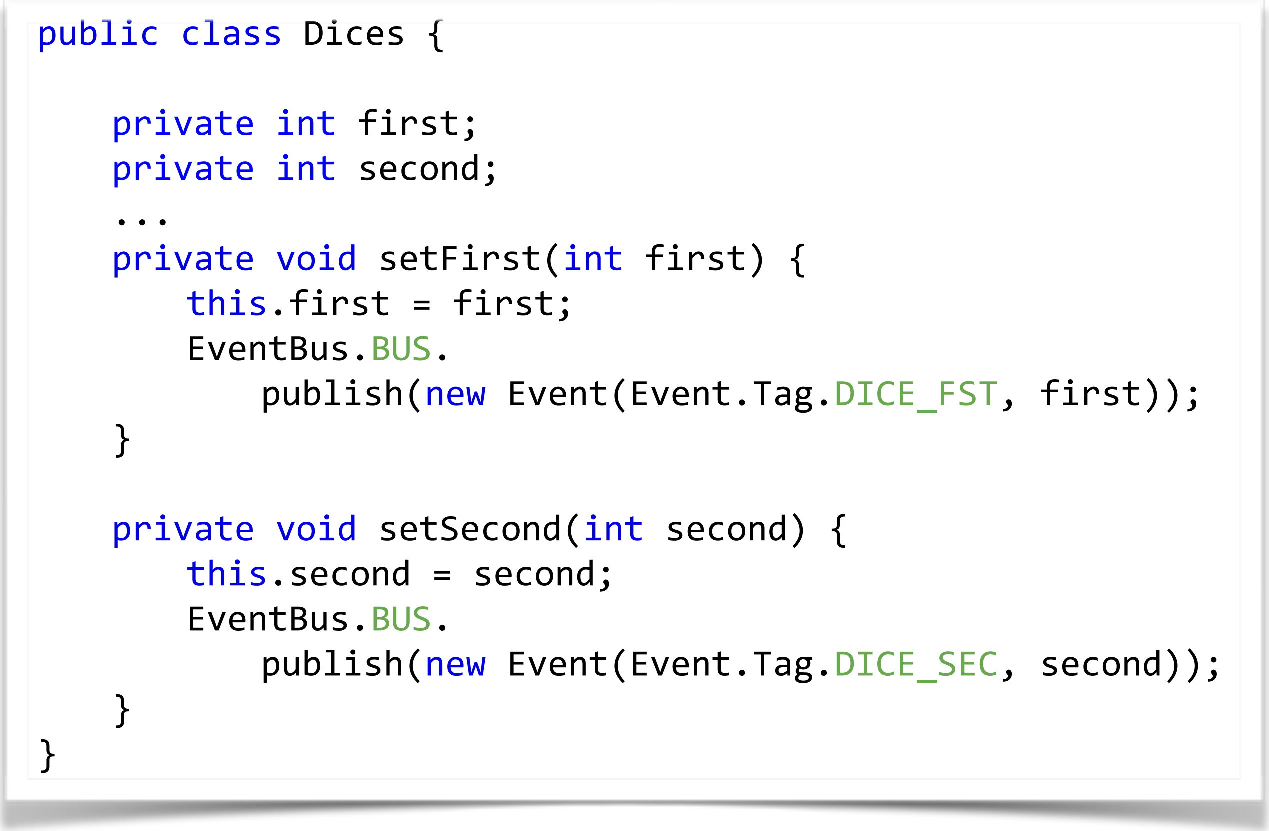


```java
public class DicePanel implements IEventHandler … {

    // 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

- We don’t want to clutter model classes with event publishing
    - Do event publishing in setters (possibly private). Class must use setters, not direct assignments!
- Alternatives:
    - Wrap a class in an ’Observable’ class and forward calls
    - Extend a class and add publishing in sub-class

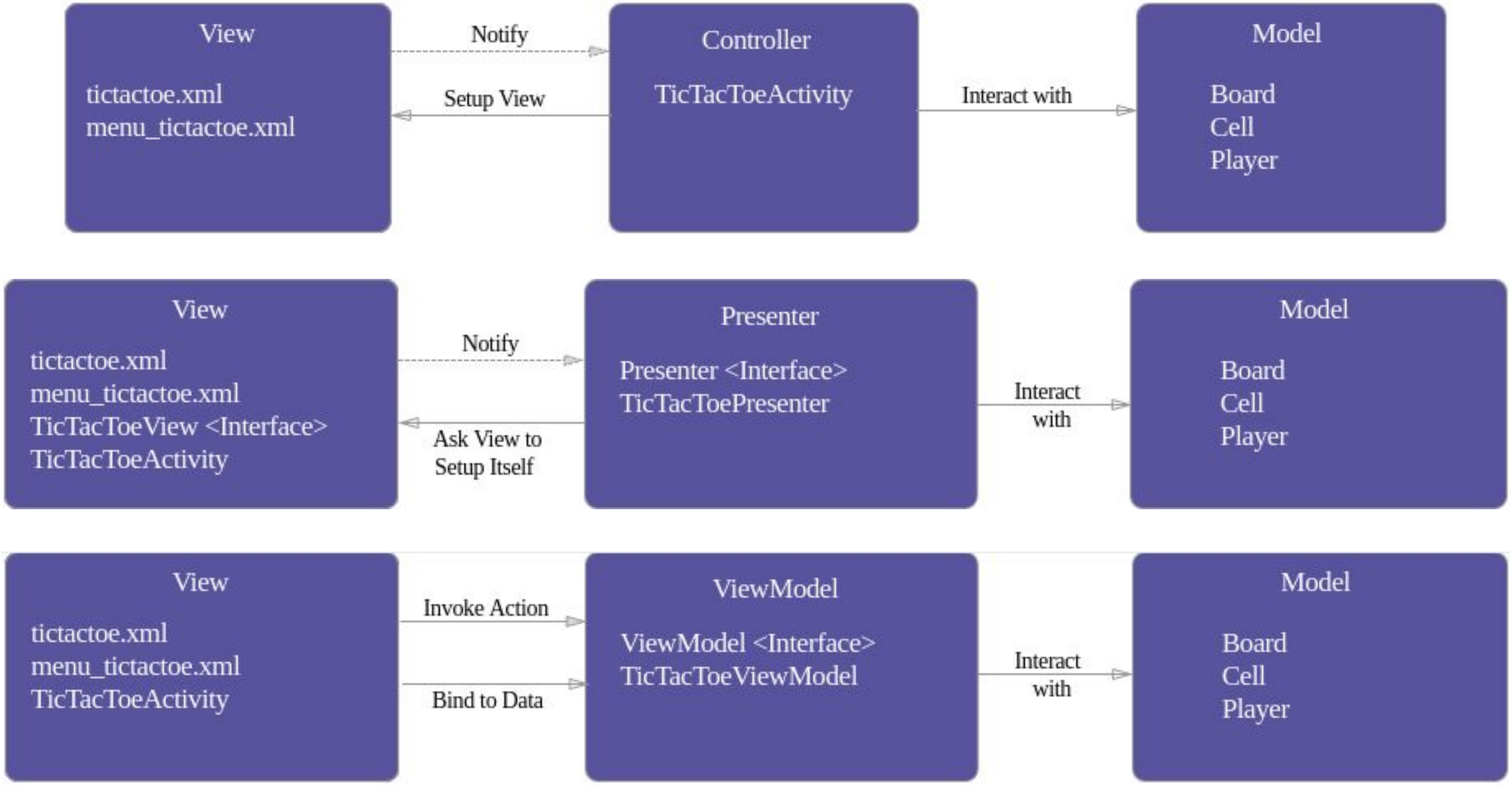


```
public class Dices {

    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));
    }
}
```

Existing EventBus

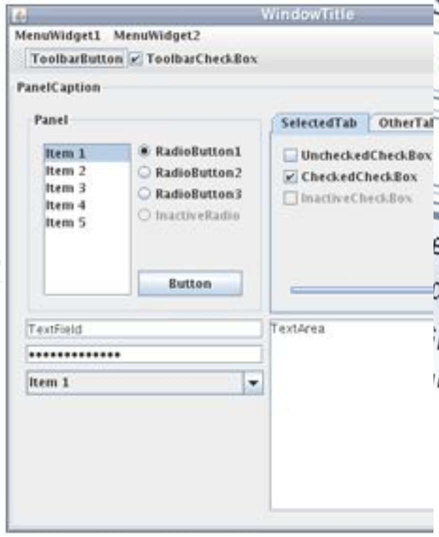
```java
import com.google.common.eventbus.*;
// Google Guava Eventbus
public static final EventBus BUS = new EventBus();

// 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));
   }
}
```

MVC vs MVP vs MVVM



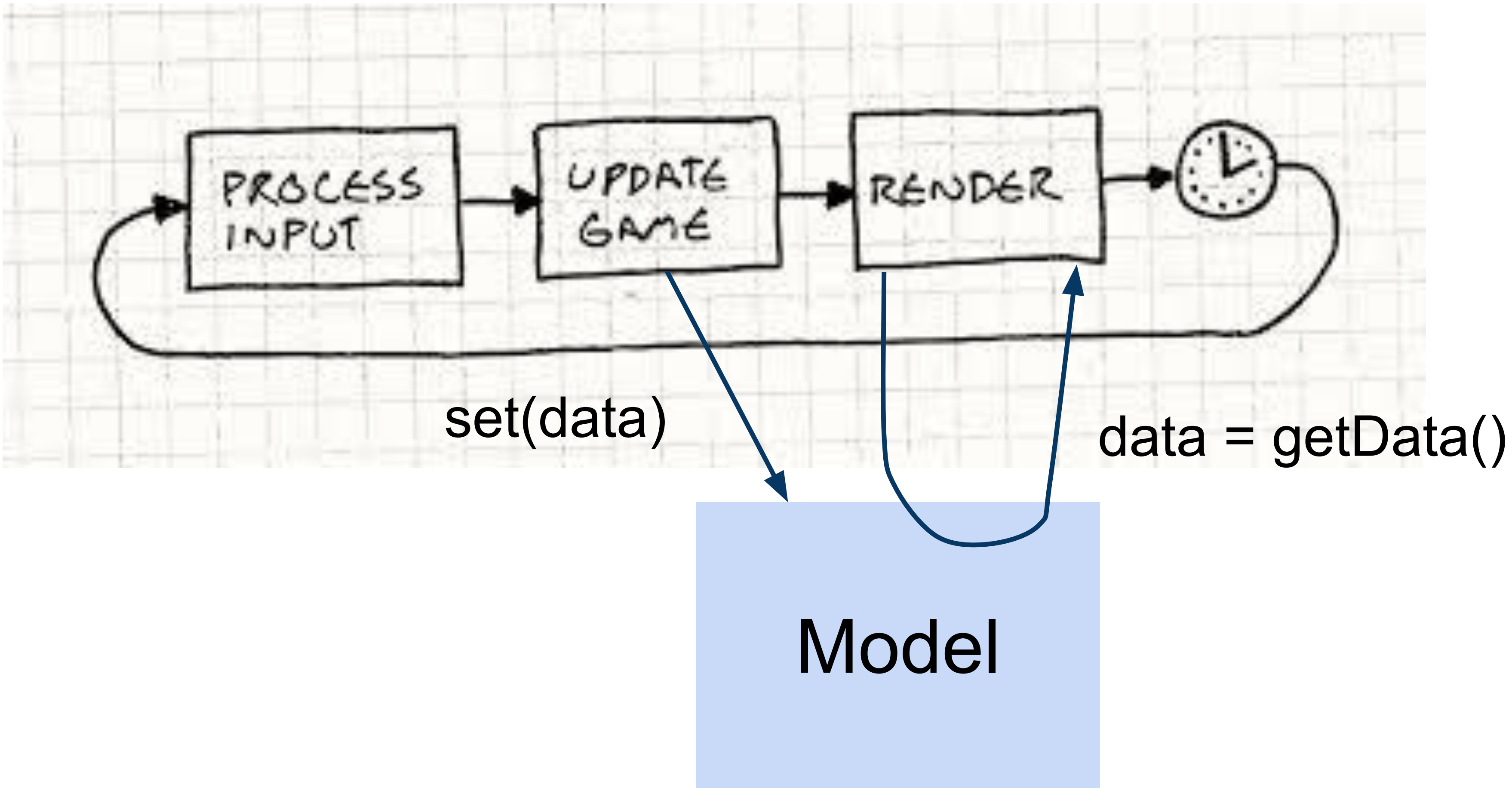
src: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

Choosing GUI technology



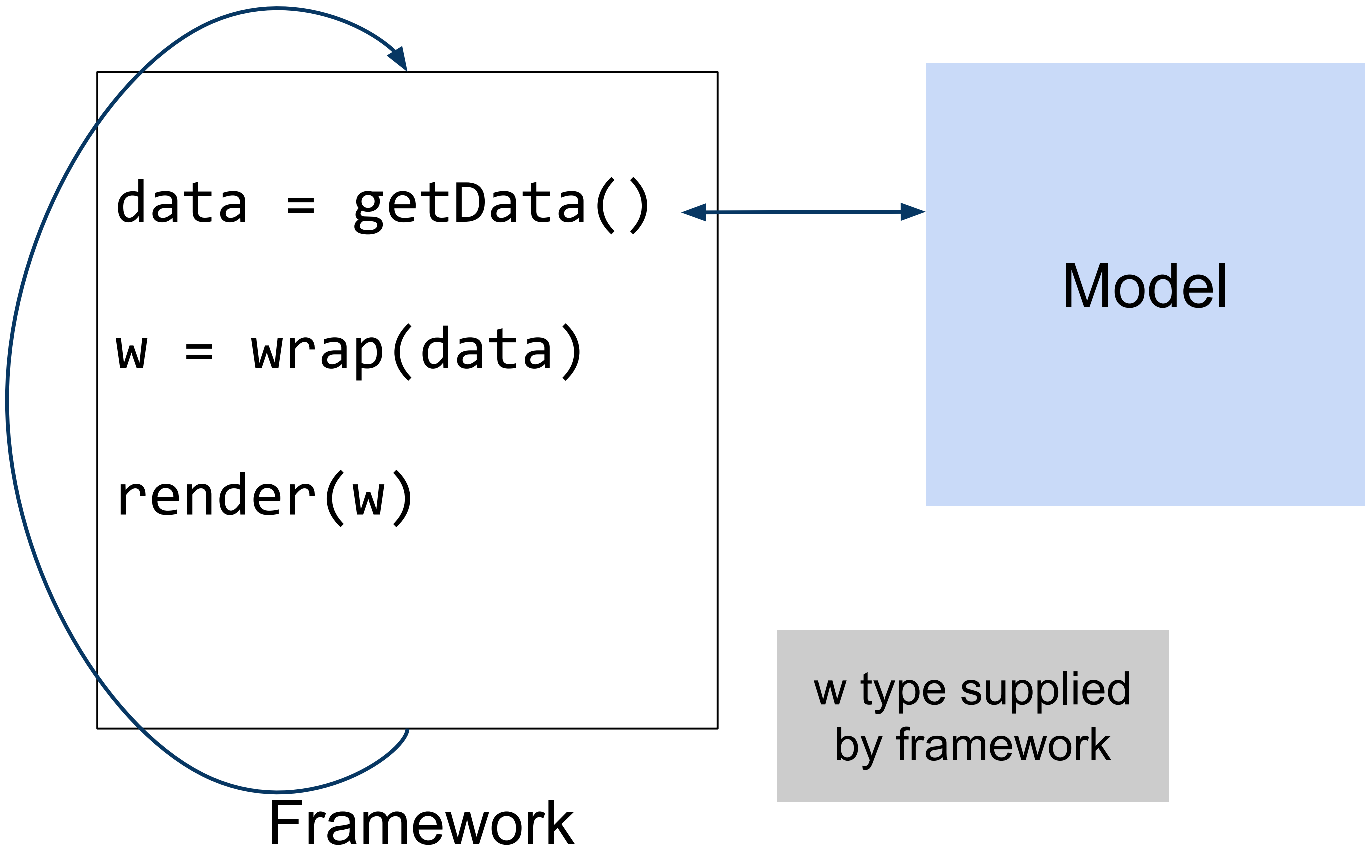
SWIX
User
Help
SwiXML
<button name="btn_panel" ToolTipText="JPanel
<button name="btn_button" ToolTipText="JButto
ActionCommand="widget_button" />
<button name="btn_label" ToolTipText="JLabel"
</toolbar>
<panel layout="border
ndable="true" dividerLoc
<tree name="tree" />
</scrollPane>
<panel layout="borderlayout">

Using a graphics framework



- Probably no ’full’ MVC design when using a graphics framework
  - No problem, but the model should be isolated!
  - Mostly using a *pull* design (render ask model for data)
  - Control replaced by update game (method periodically called by framework)

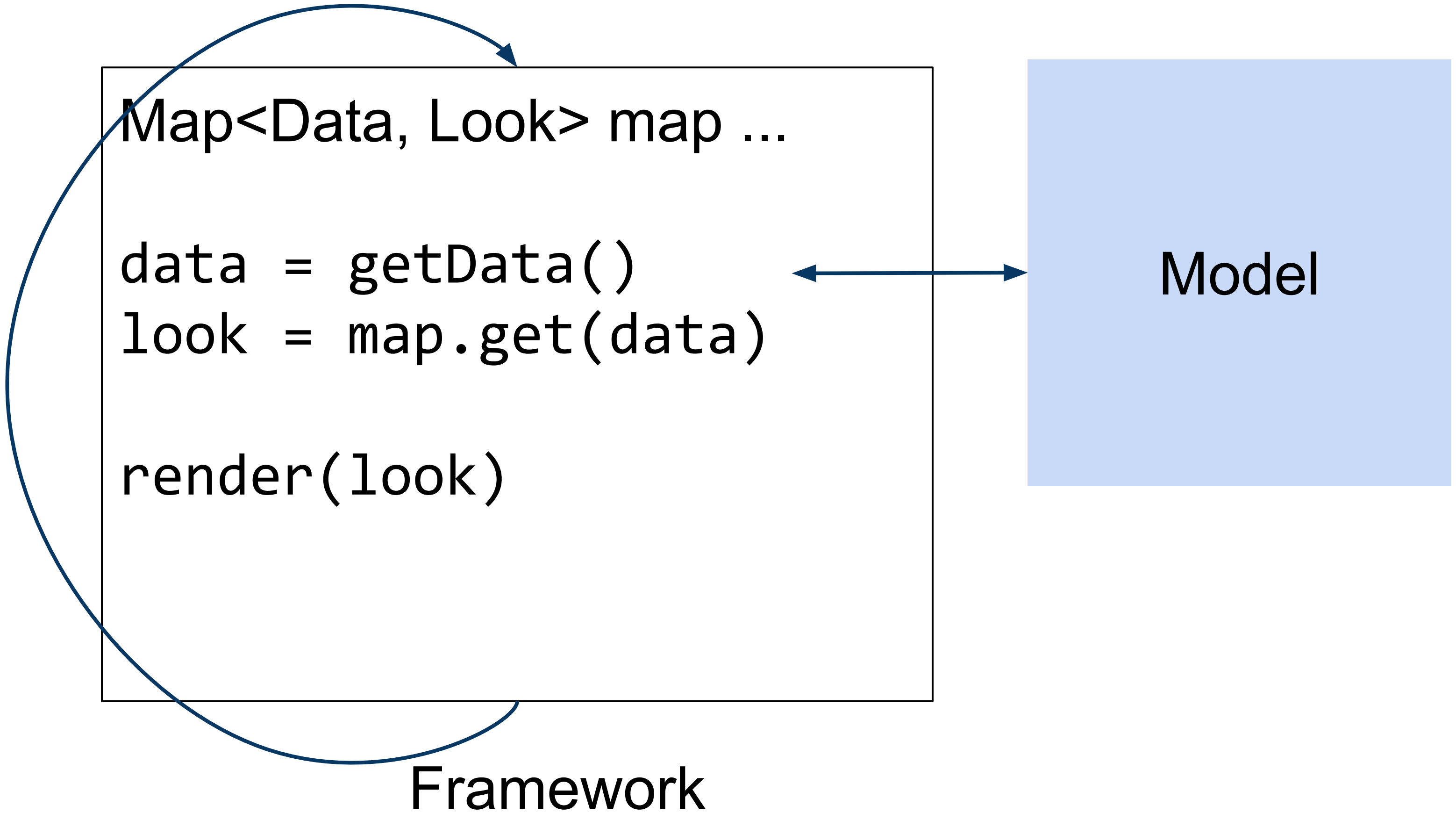
Render model



```
data = getData()

w = wrap(data)

render(w)
```

- No rendering data in model
- No imports of framework classes in model!
- If the rendering is handled by framework:
  - Wrap model data in framework classes
  - Keep model clean

Visual appearance

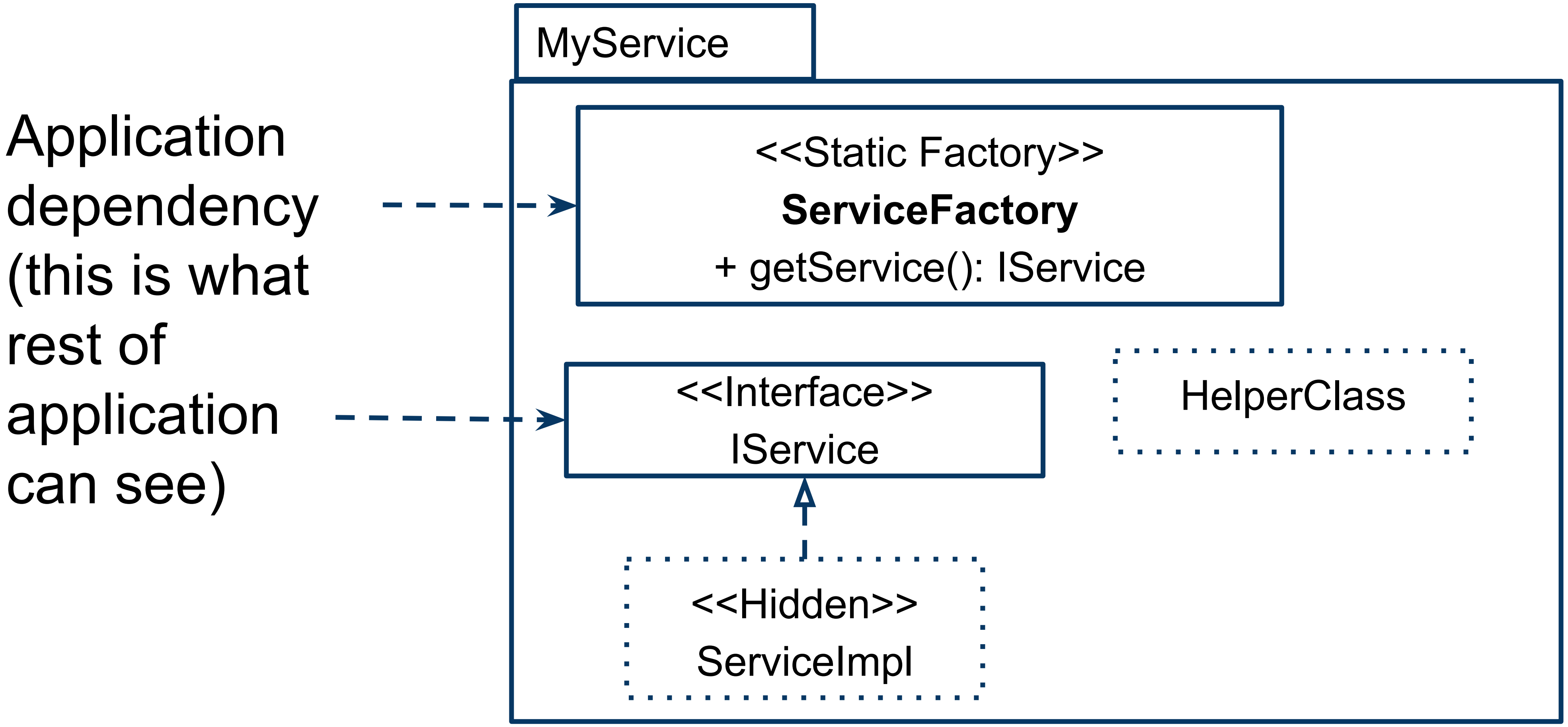


```
Map<Data, Look> map ...

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!

Services

Implementing a Service

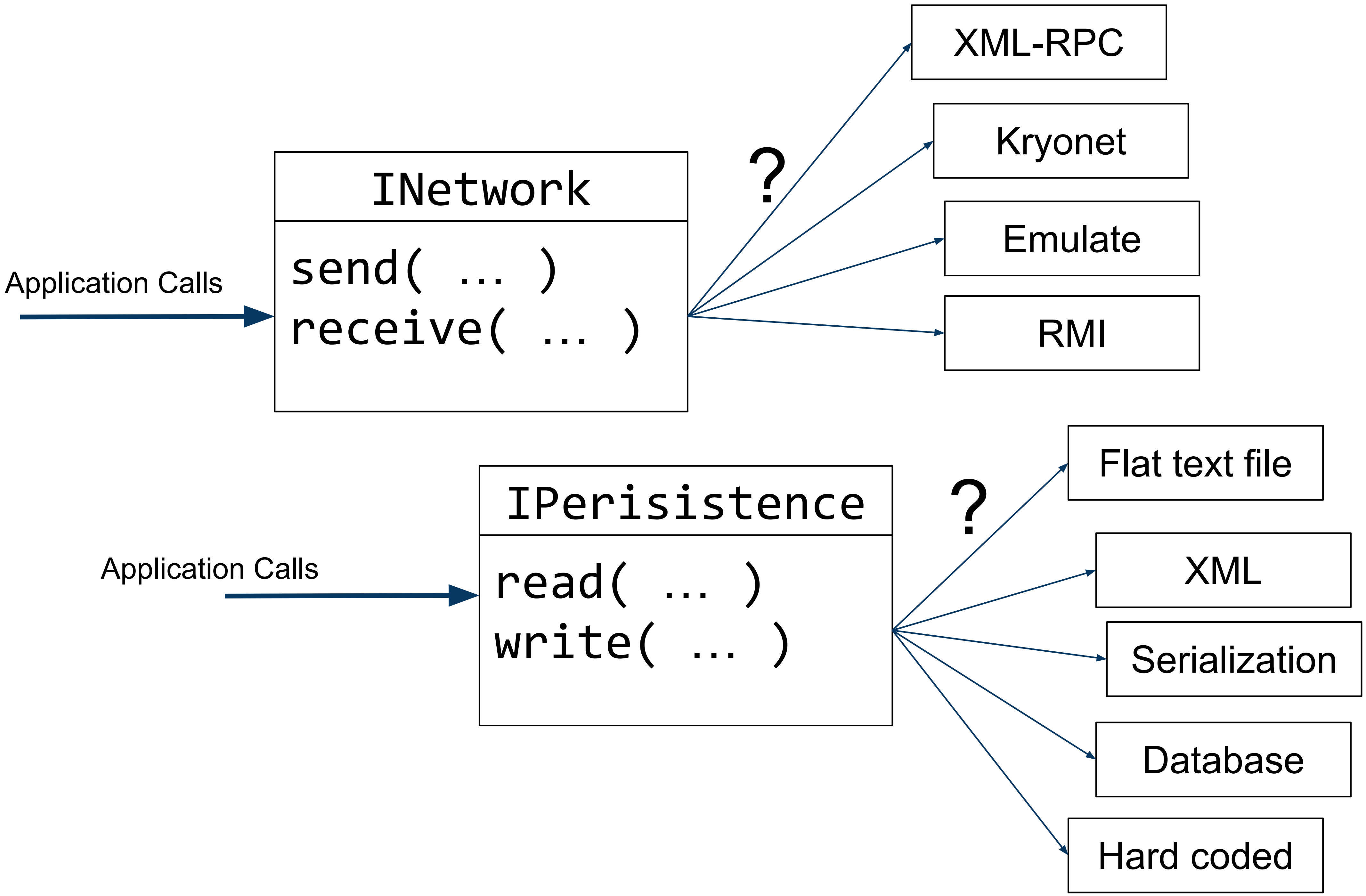


```
IService s = ServiceFactory.getService();
… s.doService( …  );
```

Implementing a Service

- Services are implemented using a *Facade* pattern
  - An interface used by control layer and a Factory to instantiate a service
  - All other classes are package private (i.e. no public)
  - Implement pure data classes as immutable value objects whenever possible
  - Use of generics may remove dependencies
- Often need to decide on data format
  - 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

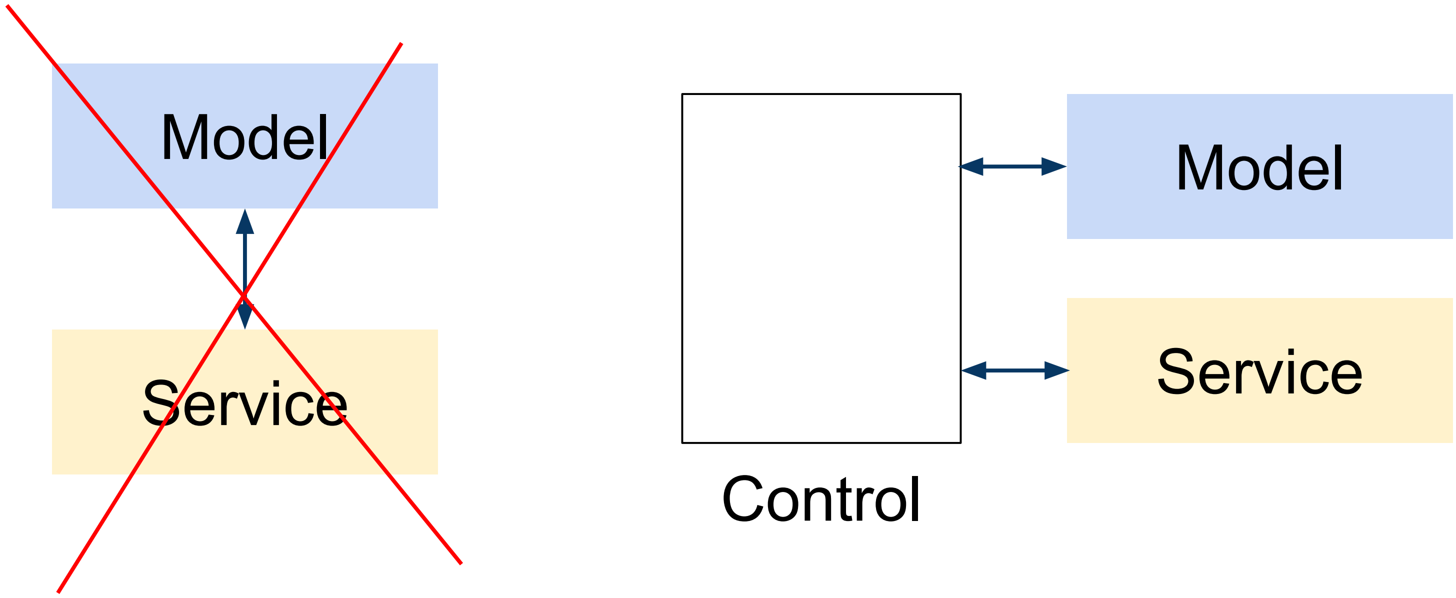
XML

Serialization

Database

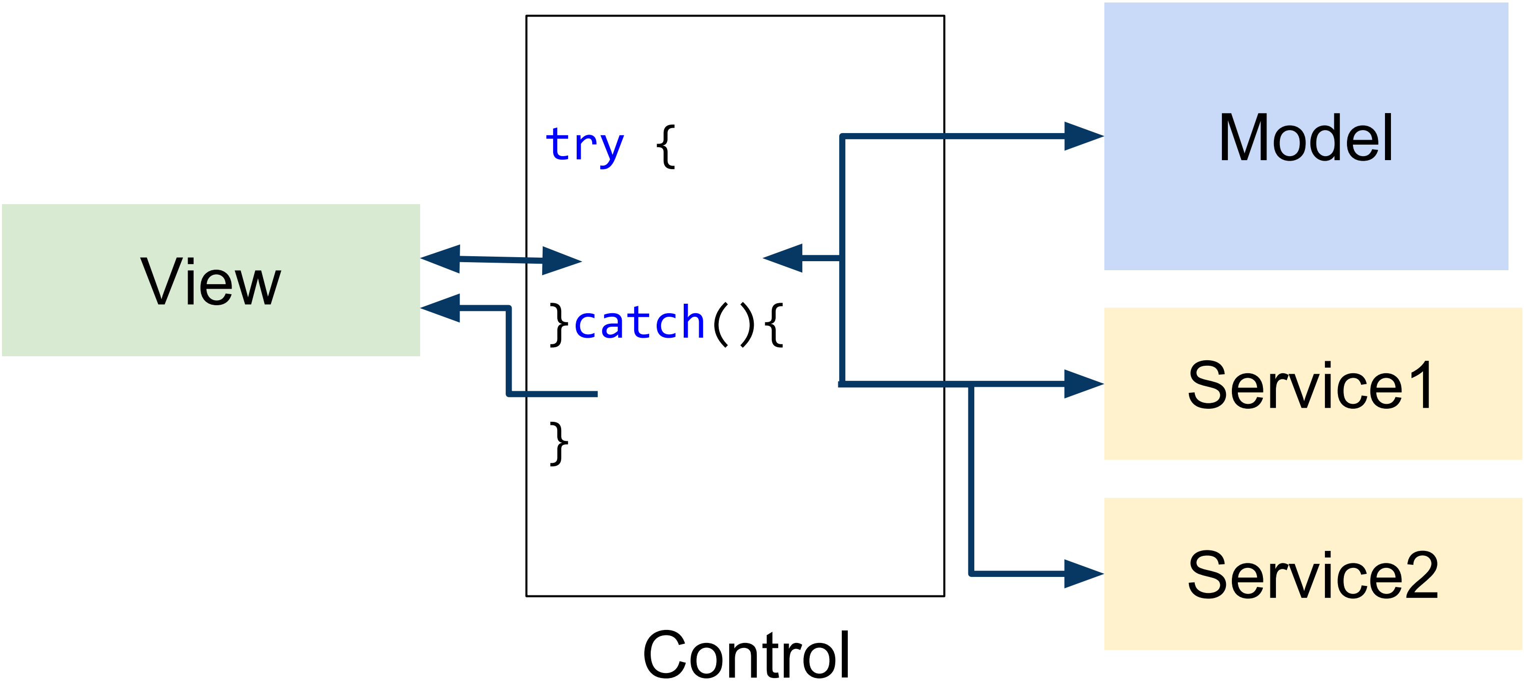
Hard coded

Usage of a Service



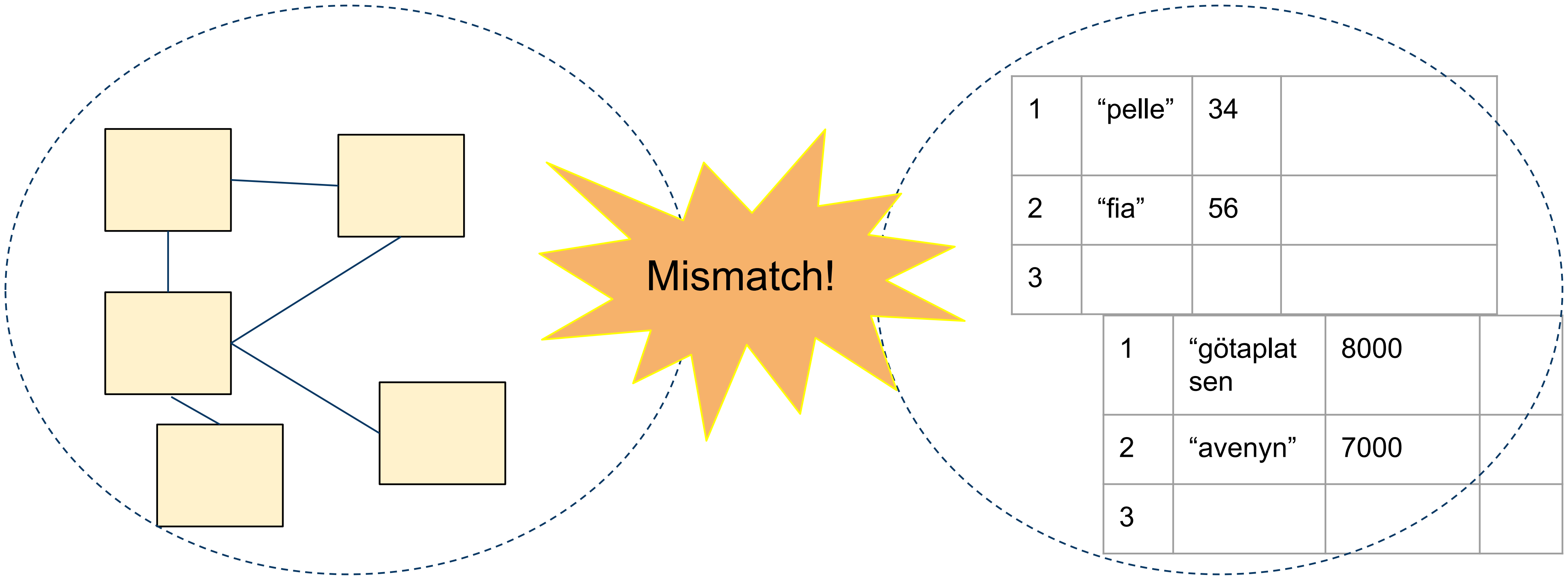
- Again: don't clutter the model!
  - No service code in model
  - Use a controller:
    - Get data from model and shuffle to service or
    - Get data from service set in model

Exception handling



- Exceptions may come from Model or Services
  - Model or Services called from control
    - Model never calls service directly
  - Handle exceptions in control
  - Propagate message to view to inform user

A note on databases



1	“pelle”	34	
2	“fia”	56	
3			

1	“götaplat sen	8000	
2	“avenyn”	7000	
3			

OO modell

Relational database

- OO-models and relational databases don’t match
  - OO model is a web of objects
  - Database is primitive data in tables
  - Object relational impedance mismatch
  - Possibly : Use some ORM framework (Hibernate)
- Avoid using databases, emulate (use an interface)!

System Design Document (SDD)

SDD

- The system design document’s (SDD) goal is to make the implementation of the application understandable

- The SDD 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: software architects and programmers

- The SDD is a **"live"** document that should be expanded and refined during/ after iterations

- *The SDD is about communication, no strict rules on how to write it*

SDD

- We prefer a top down explanation:
  - Start out with the high level (big picture):
    - Hardware setup, communication, applications involved (if applicable)
  - then refine in each step:
    - Structure of (each) application
    - Packages
    - Design model
    - Possibly classes/interfaces
  - until close to code:
    - when reaching this level: the code and the tests are the documentation

Summary

- We have discussed many implementation issues:
  - Refine and refactor both design and implementation
  - MVC issues
  - Services
  - System Design Document
- Next: continue until finished 😀