Iteration 2 [All phases]

Slide Series 6

Iteration 2 Input

From iteration 1 we have

- Requirements and analysis model (RAD)
- Some prototype experiences (possibly technical)
- The design model v 0.1
- A few running use cases
- No real GUI, possibly a very simple one (or a command line)

Iteration 2 Goals

Iteration 2 should add more use cases …

- … but we also need to review the model …
- … and check new/changed requirements?
- During iteration 2 we also start with a more well-reasoned design (which we begin to document in the SDD)

- More serious testing and code coverage
- Adding (more of) a GUI

Iteration 2: Requirement Elicitation

Hmmm... possibly have missed something?

- An important or misunderstood use case?
- Missing functionality?
- New or changed requirements for GUI?
- If so update RE sections of RAD

(I skip this for MP, ... you do for your project)

Iteration 2: Analysis

If changes to RE probably changes to analysis parts

- If so update analysis sections of RAD (in particular the analysis model)

(I skip this for MP, … you do for your project)

More UCs for MP

We start out with adding two more UCs

- Buy and Sell
- Have Property class, introduce some more spaces: Street, Tax
- Interesting (new) class: Debt
- Still testing out model using the command line
- **Demo:** Design model 0.2

There's a README file in all MP version. Do read it!

And now for some testing (see Test slides,
… then we return to here)

Design

Design is a multi dimensional activity. Many different aspects

- Modularity
- Efficiency
- Communication
- Testability
- Reuse

Design is a multi (abstraction) level activity

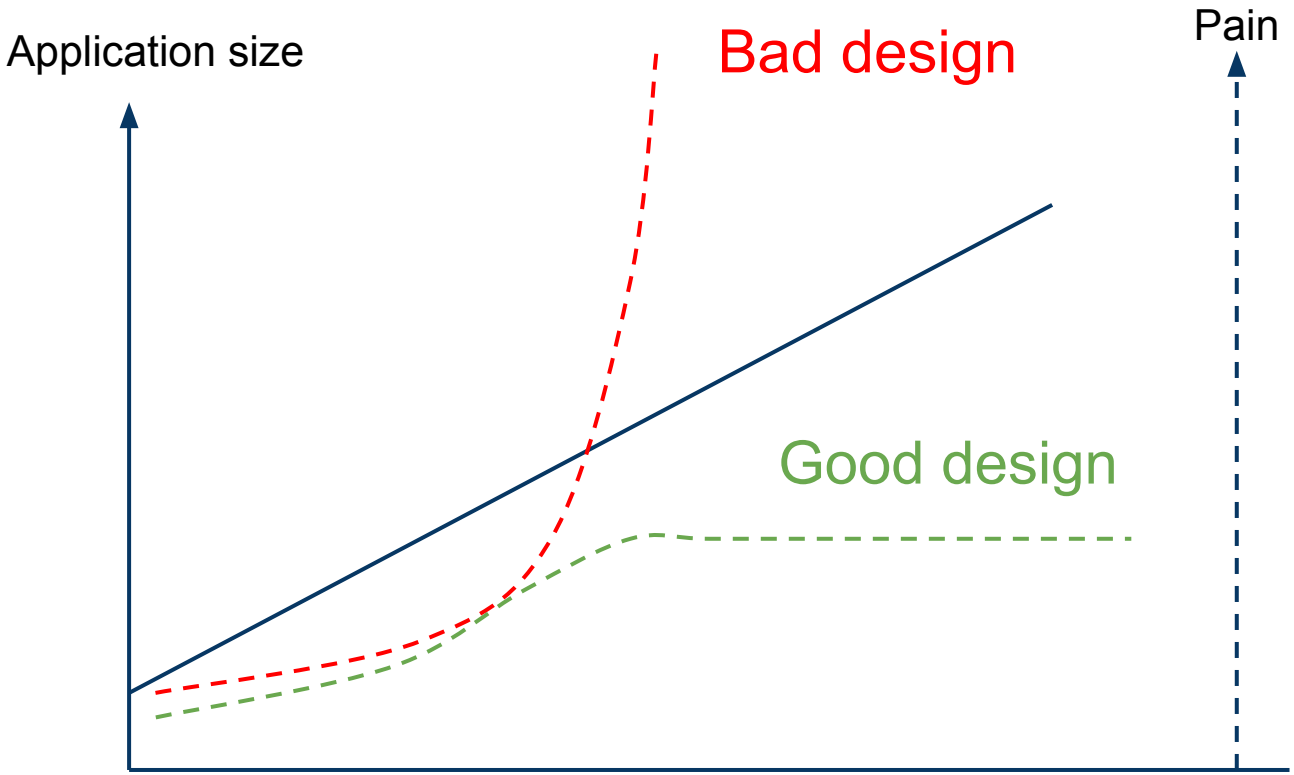
- Overall application (system) design
- Module design
- Class design
- Method design

Design Goals

Overall design goals
- Create an identifiable structure
- Enforce localization of responsibilities
- Minimize dependencies (i.e modularity)
- Control (minimize) state
- Clear and robust inter-application communication

Thereby making it possible to create a modifiable, extendable and testable program (with possibly reusable parts)

The Impact of Design



Design is important …!

Design Considerations

Some considerations

- Information expert
- Single responsibility principle
- Open closed principle
- Programming to interface
- Low coupling/high cohesion
- Information hiding
- Law of Demeter
- Invariants
- Mutability
- Minimize State
- Canonical form for objects (equals, clone...)
- Threading?
- MVC (must use, some kind of)

During our work we must keep an eye on this, if violating, refactor!

Design patterns ... if needed!

Mutual Associations

Mutual associations are bad

- Must keep two object in synch (reference each other)
- Domino effects (change one, affect other)
- Classes not understood in separation

Resolve by

- Ignore one Direction
- Lookup one direction
- Association class

Mutual: Ignore one Direction

Do the application need to traverse in both directions?

```
public class Order {
    ...
    private Customer customer;
    ...
}
```

```
public class Customer {
    ...
    private List<Order> orders;
    ...
}
```

Do Order need to call methods on Customer? Why?

```
// Alternative
public class Order {
}
```

```
// Alternative, but...
// ...given an order have to
search
// all customers
public class Customer {
    ...
    private List<Order> orders;
    ...
}
```

Mutual: Lookup One Direction

Lookup Customer given Order (Orders unique)

```
public class Order {
    ...
}
```

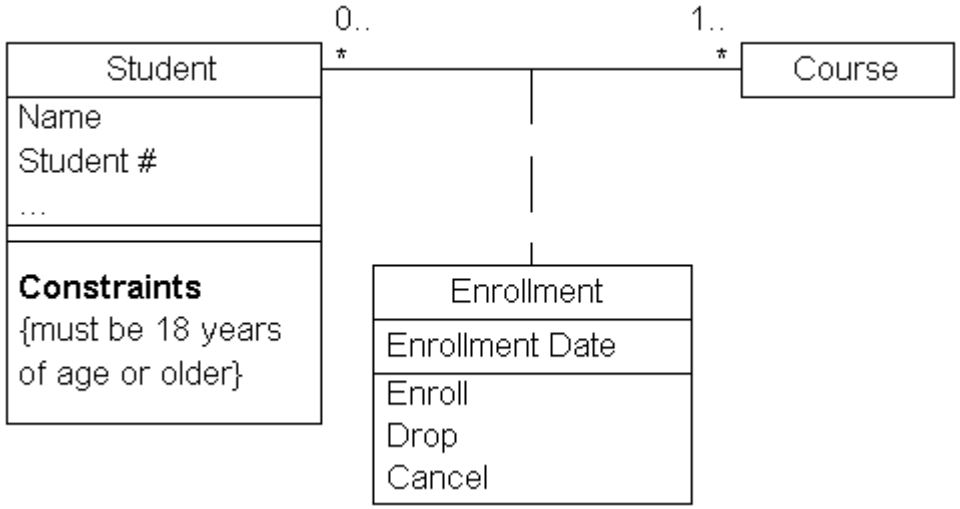
```
public class Customer {
    ...
    private List<Order> orders;
    ...
}
```

```
// Lookup class
public class OrderBook {
    Map<Order, Customer> orderCustomer = new HashMap<>();

    public Customer getCustomerFor( Order o ){
        return orderCustomer.get(o);
    }

}
```

Association Class

Mutual dependencies are bad, mutual many to many even worse

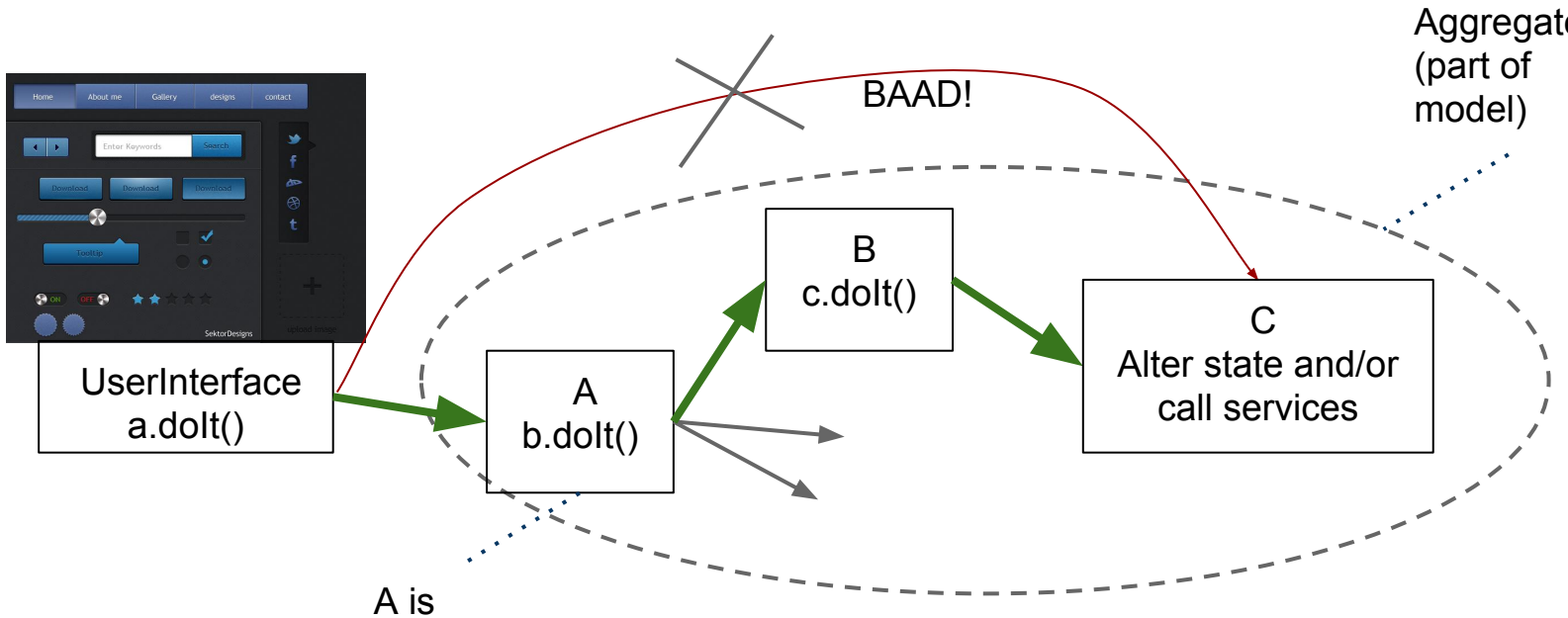
- Resolved using "extra" **association class**



Aggregates and Call Chains

Application should establish call chains

- Some classes acts as aggregate-roots (entry to a cluster of classes)
- All calls pass through root



Note: This is different from Law of Demeter

Design Review of Model

- 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 (was it awkward to implement the UCs)
- 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)
- Canonical form

MP: Design Review of Model

Pass .. (?)

Building of model, OK (have factory)!
Parametrization, OK (have Options class)!
Model in one package, OK!
Cyclic associations or dependencies OK!
Aggregates and call chains , OK (?) it seems …
Classes … (?) OK for now …
Absent values: Player.NONE (alternative to null for missing street owner) OK!
Canonical form, at least equals()/hashCode() for Player, … more needed?

MP: Design Review of Model, cont

Known issue

- Heterogenous collections for Spaces. Many kind of spaces handled in single list (easy to handle).
But also need to distinguish spaces (can't buy/sell Chance or Tax).
Should avoid **instanceof** (generally bad, better with polymorphism)
- Very many rules: Would like to be able to "plug-in" rules (if we find a missing one). For now hard coded. Ideas?...

MP Solving Design Issues

Handling heterogeneous collection of spaces

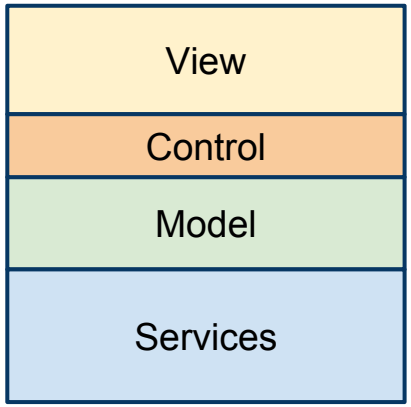
- Solution: Visitor Pattern
- Pluggable rules: Not solved, possibly let other part of application handle (i.e. not in model)

Demo: Design model MP 0.3 (still using command line)

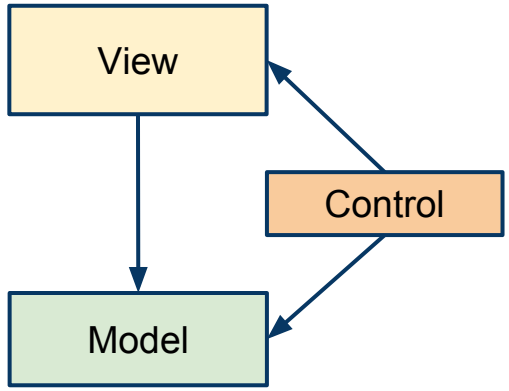
- And also more of testing

MVC Architecture

Application should use an MVC architecture



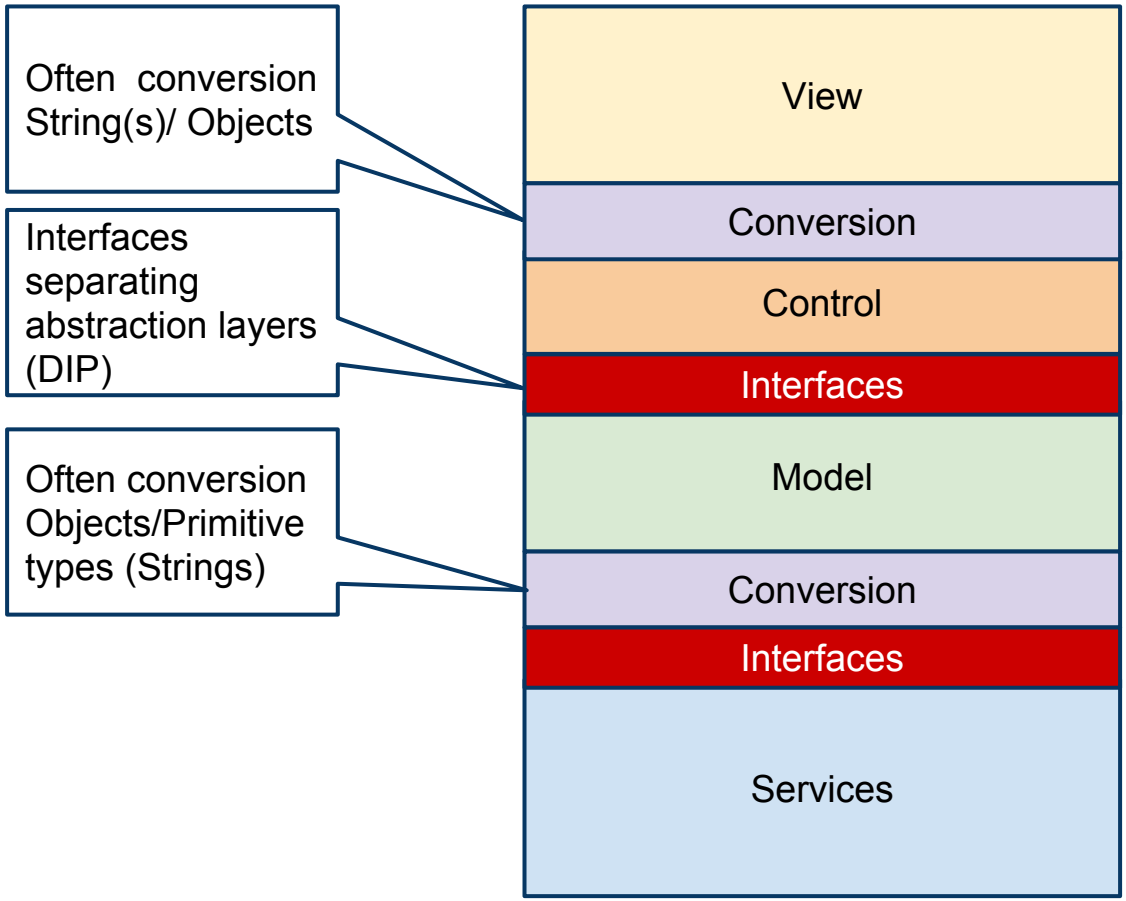
1. By abstraction layer. **Layering**



2. By parts (arrows = possible calls) . **Partitioning**

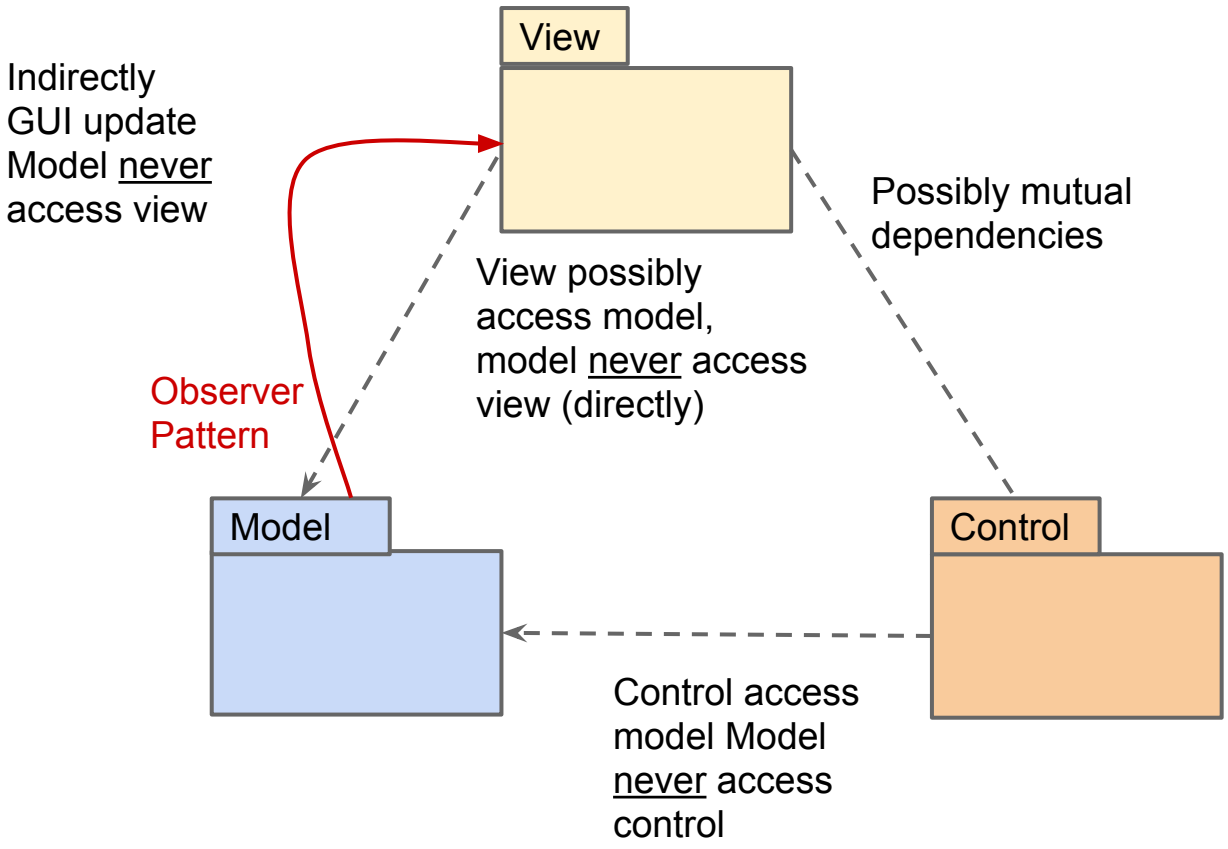
Layered View

In more detail



Partitioned View

In more detail



The Need for a Control Layer

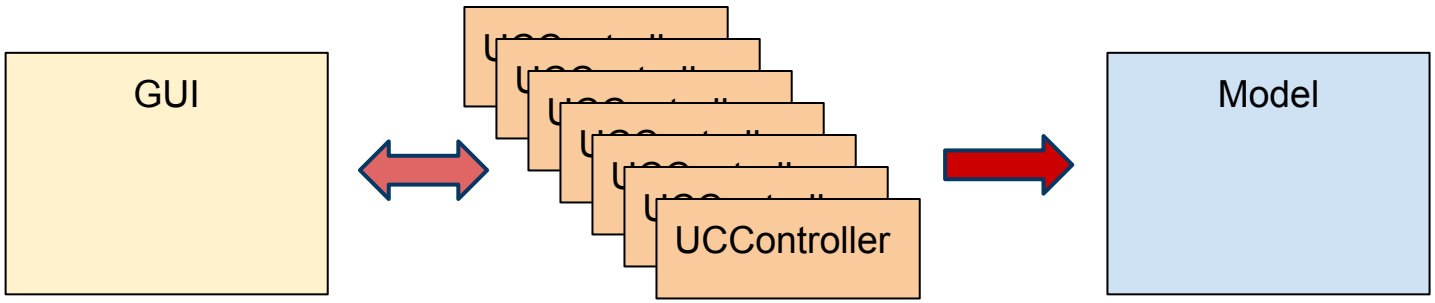
How should GUI and model interact

- Should model be updated after each tile?

Use Case Controllers

Control layer could also be comprised of "use case controllers"

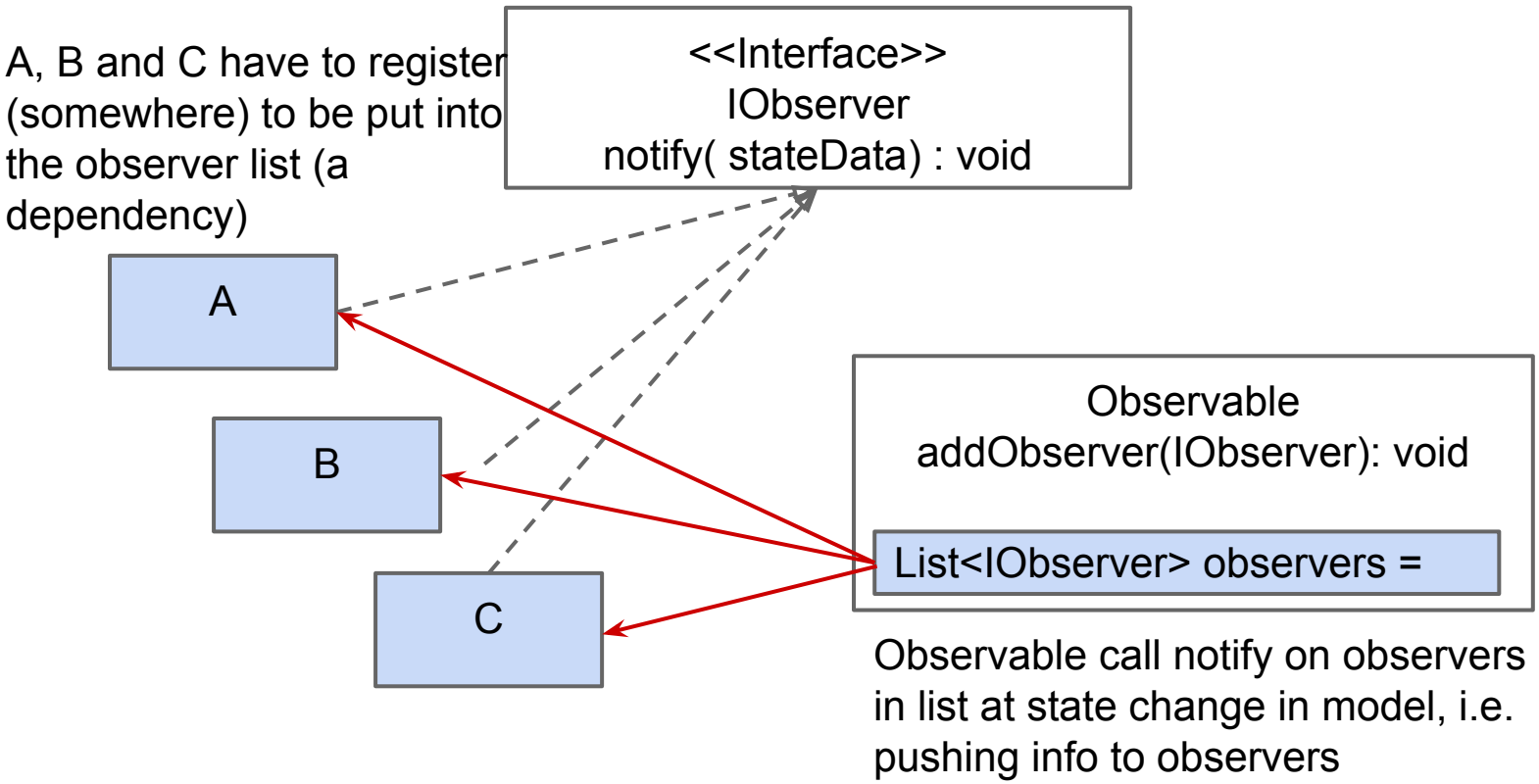
- Each UC handled by a controller class. Class runs UC parts not present in model or mediated UC between view and model (as shown in prev. slide) or between model and services.
- Could this solve the "pluggable rule problem"????



Standard Implementation Observer

Observables must get references to observers.

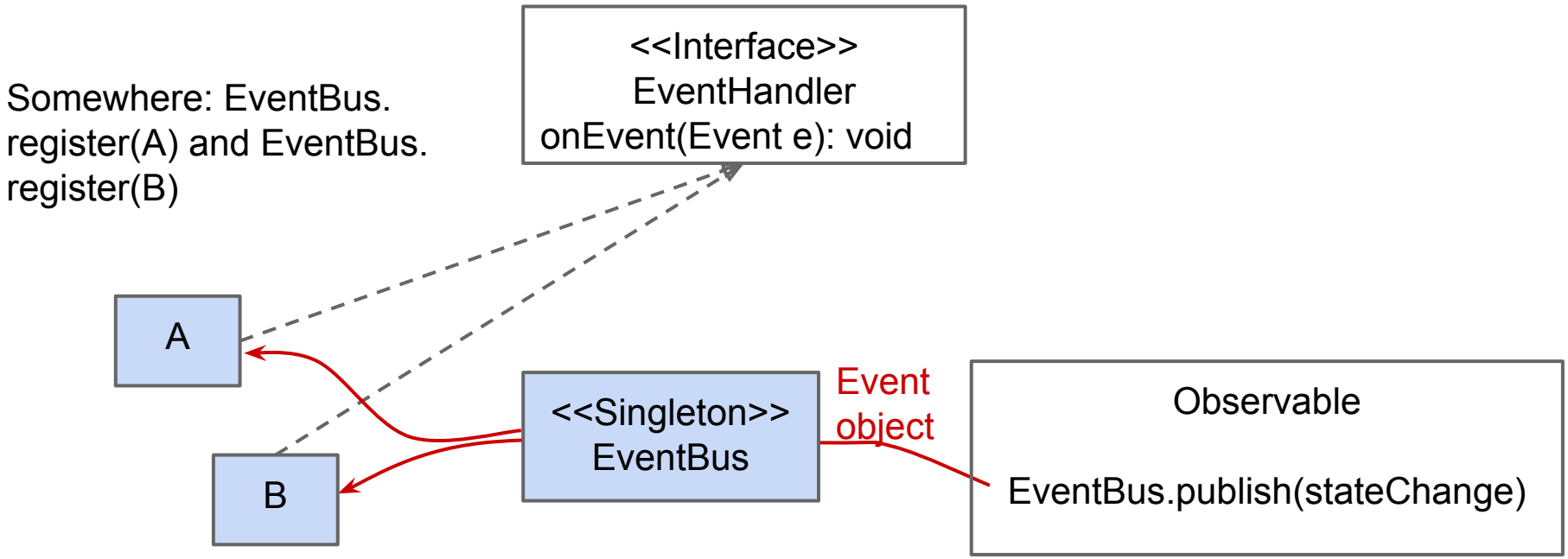
- This is i push design.



Event Based Observer

Better use a messaging mechanism, an EventBus

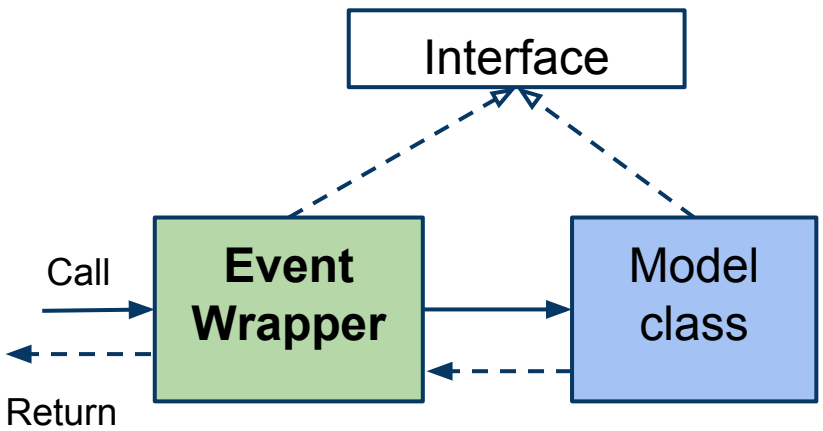
- No direct connection between observer and observable
- Easy to (un)register



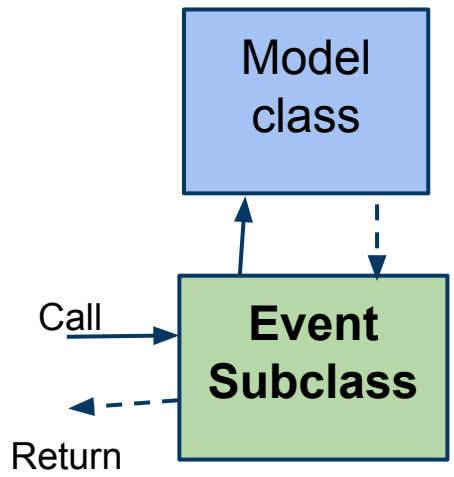
The Model in Event Based Design

How to keep model clean (from messaging code)

- At least 2 different ways (OPC)



Alt 1: Event code in EventWrapper core functionality in Model (Wrapper delegates)



Alt 2: Event code in subclass core functionality in Model

View

During this iteration we start implementing a real GUI

- Have a mockup from RE
- Identify input/output elements from mockup
- Will change in response to the design and implementation of the model, so avoid too many details
- Plan for techniques/tools to use

If using a different GUI-style (animated, 3d) you need to start "technical" prototyping right now!

- More later

GUI Tools

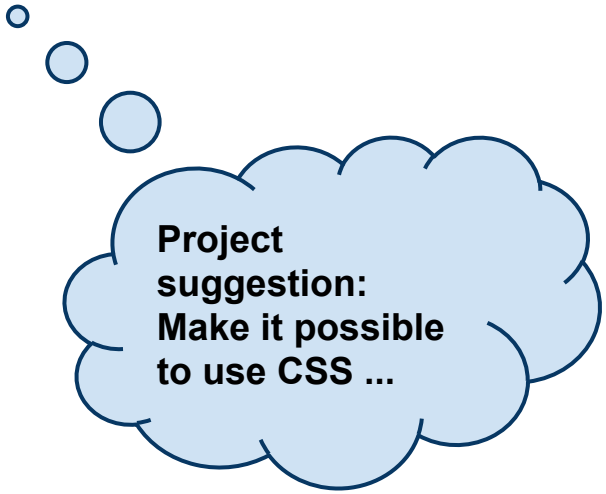
GUI-drawing tool often generate horrible code

- Possible many times as much code...
- Checkout before using in full scale (clean up generated code)

NetBeans have built in (unpopular?) GUI-builder

Using XML to define the GUI

- SwiXML
- BerylXML
- Links from course page
- Others...???



Swing Technical

Some Swing Issues

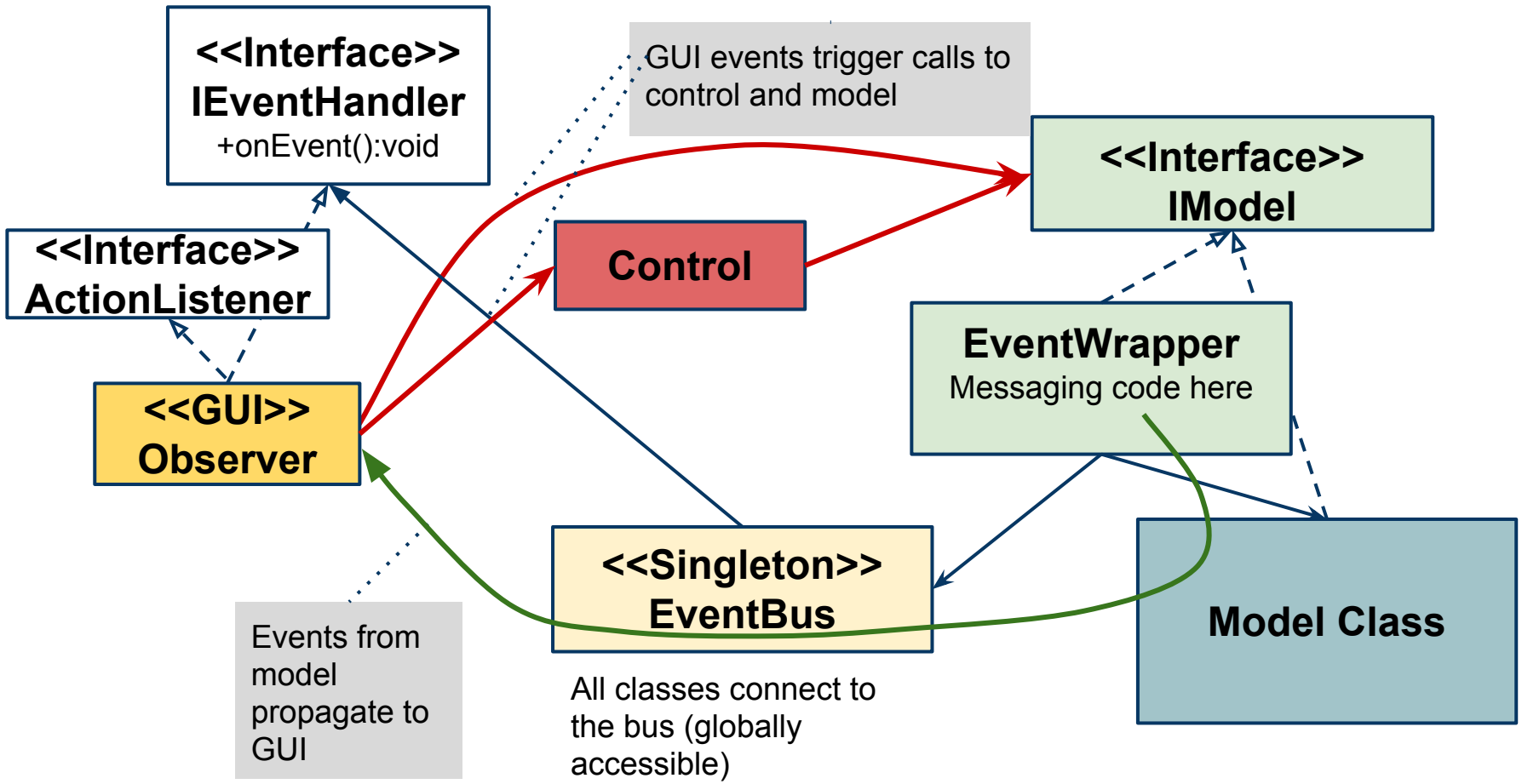
- The GUI shouldn't be monolithic (i.e one huge JFrame)
- GUI is composed of many panels (custom components also good)
- Separate construction code (JButton b = new JButton()) and event code (listeners)
  - Use factories
- No model logic whatever in GUI
  - Don't use GUI for logical behaviour i.e. disable a button to prevent. Button disabled because of model state!
- GUI can use non-visible classes: Options, Preferences, etc. (holding "look"-data (non-model data, data not part of the domain problem))

Swing Issues

If using Swing...??!

- Force repainting using; validate(), repaint() or possible both
- Use setSize() or pack()
- Some components fire events when data changed! Possibly have to disable when updating component models
- All drawing in Swing thread, possible have to hand over form other thread (SwignUtilites.invokeLater)
- Time consuming operations use SwingWorker

MVC Design Overview (Swing)



Alternative MVC style

If non-Swing, using other libraries/frameworks

- Main rule: **No framework code in model**
- Often have to design a MVC pull model.

```
// Framework MVC style
while( true ){     // Game loop
    input = framework.getInput()
    model.update(input)  // State change in model
    framework.render(model.getState())  // Pull model
}
```

Tower Defence Design

GUI/MVC choices

- Swing 2d?
- .. other?

We chose other: LibGDX

- LibGDX (have built in game loop)
- We have a very quick look ...

NOTE: There's an Maven archetype to create LibGDX projects

Services

Model need services to run, some typical

- Persistence (store/retrieve)
- Printing
- Rule systems (business/game rules), filters, …
- Engines, simulation engine
- Processors (text formatter, spell checker)
- Security, authorization module
- Mappers, mapping between formats
- Event buses
- …

Services Defined by Interfaces

Any service has a coherent interface

- The interface it what's matter (what do you need, think hard!)
- Rest of application uses interface
- Prefer stateless services!

Implementation/technique is secondary (at least for now)

- Persistence (saving) could be done using any of, flat files, XML, database, …
- Keep open mind
- Create mock object emulating service, implementing interface

Services: In House or ?

"In house" = we implement system

- ... possibly many man-month needed for complex services!

Typically you don't implement subsystems for

- Graphics, sound, physics engines, databases, XML, ...networking, ...

Find it (use Maven)! Look for high level abstractions

In-house Service Design

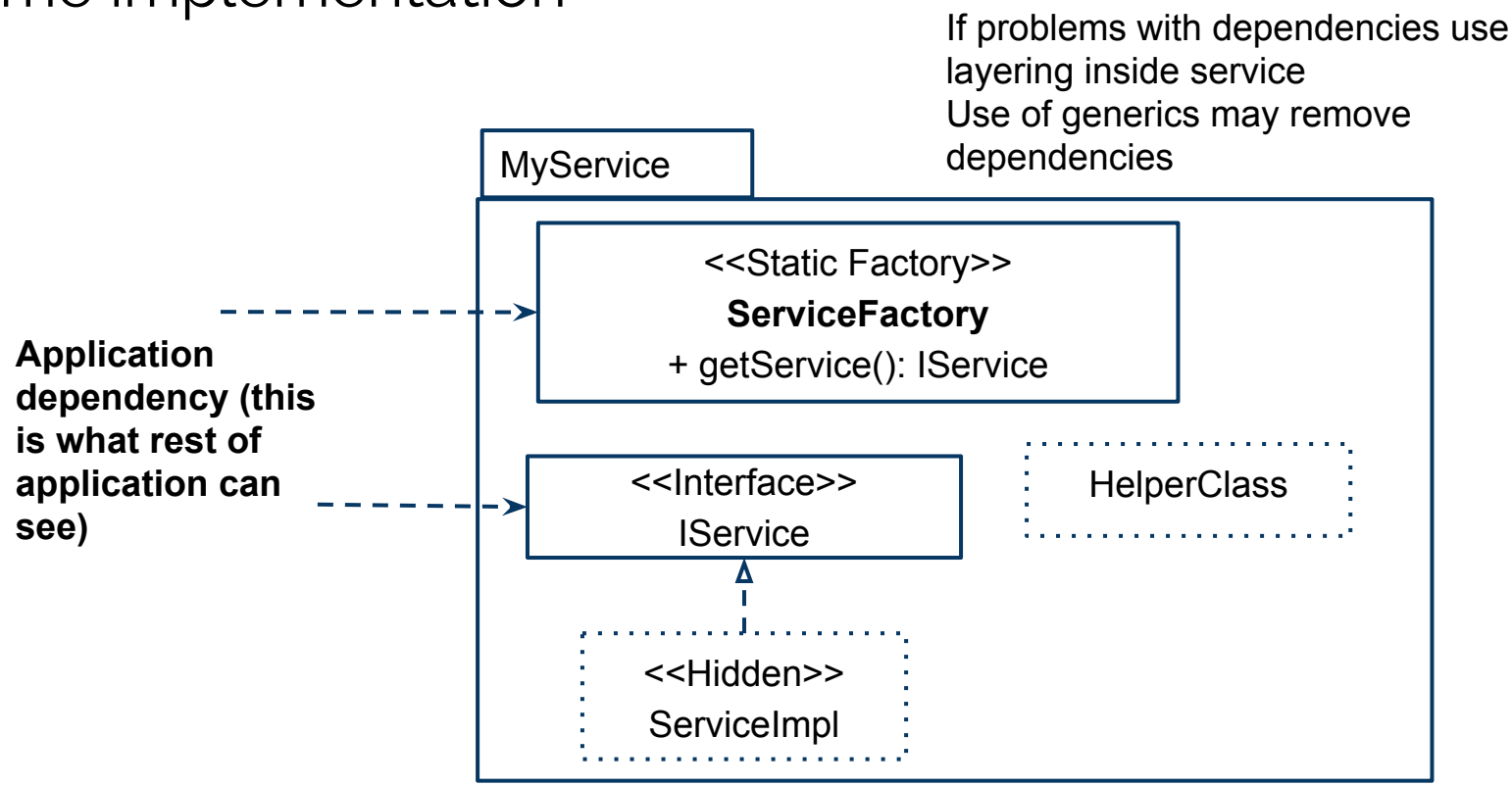
When designing a service often beneficial to split services into steps

- Step 1: Basic low level often working with primitive type, Strings (possible reusable)
- Step 2: Conversion step; using step 1, convert from/to objects (application specific)

Implementing Service, cont

Services implemented using Facade pattern

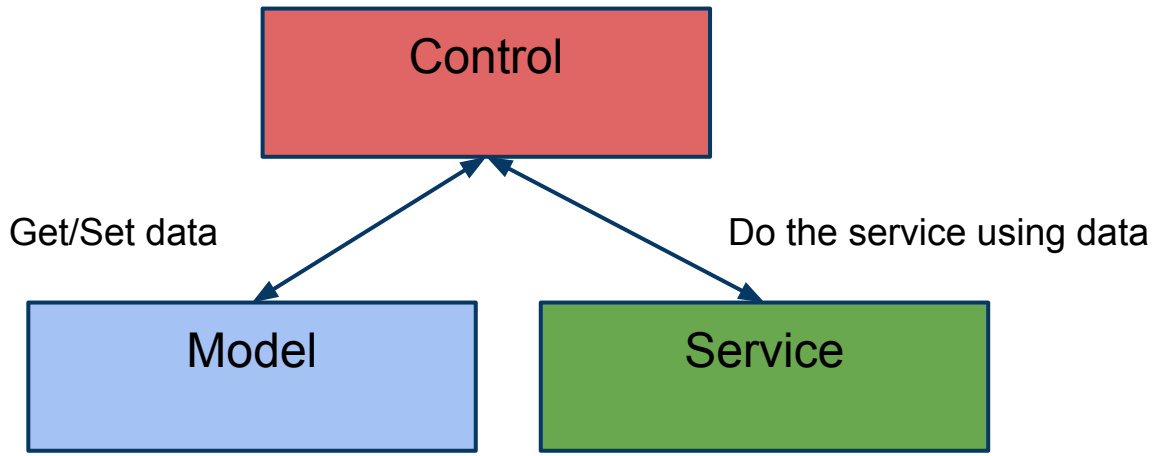
- I.e. an interface used by model (or other) and a Factory to get some implementation



Model and Services Interaction

Would like to avoid service code in model (as before with messaging)

- Put interaction between model - service in control layer



Resources

Application need resources

- Config files, icons, sprites, images, localization, i18n, ...

Resource handling

- java.util.ResourceBundle, File automatically read and converted to Java "map-like" object, for texts in GUI
- For images use ClassLoader class getResource(s), findResource(), ...
- Implement as service
- Frameworks normally has their own ways

MP Services

Will use external XML API, XStream, for saving games

- Wrapped in Persistence package
- No special conversion step (for now at least)
- Interaction service <-> model as pointed out (using class SaveAsCtrl)
- See MP v 0.4

MP: Inspection of first GUI version

To note:

- Separate packages for GUI and EventBus
- Non-monolithic GUI
- When application starts only MainFrame created
- Remaining parts of GUI constructed in MainFrame (so no GUI Factory for now)
- All GUI parts register as EventHandlers and implement ActionListeners (if applicable)
- Messaging code in MonopolyEventWrapper
- Construction of model in NewGameCtrl

Demo: MP 0.4

Controlling Quality

Having implemented a larger part of the application need to control general quality
- No mutual or circular dependencies
- Dependencies between packages
- Use STAN

- … other quality tools: PMD plugin to NetBeans
Findbugs (in project pom.xml), see MP v 0.4 for usage)

Exception Handling

If possible to recover, do so at any location in application

If not, ... handle in view or control layer

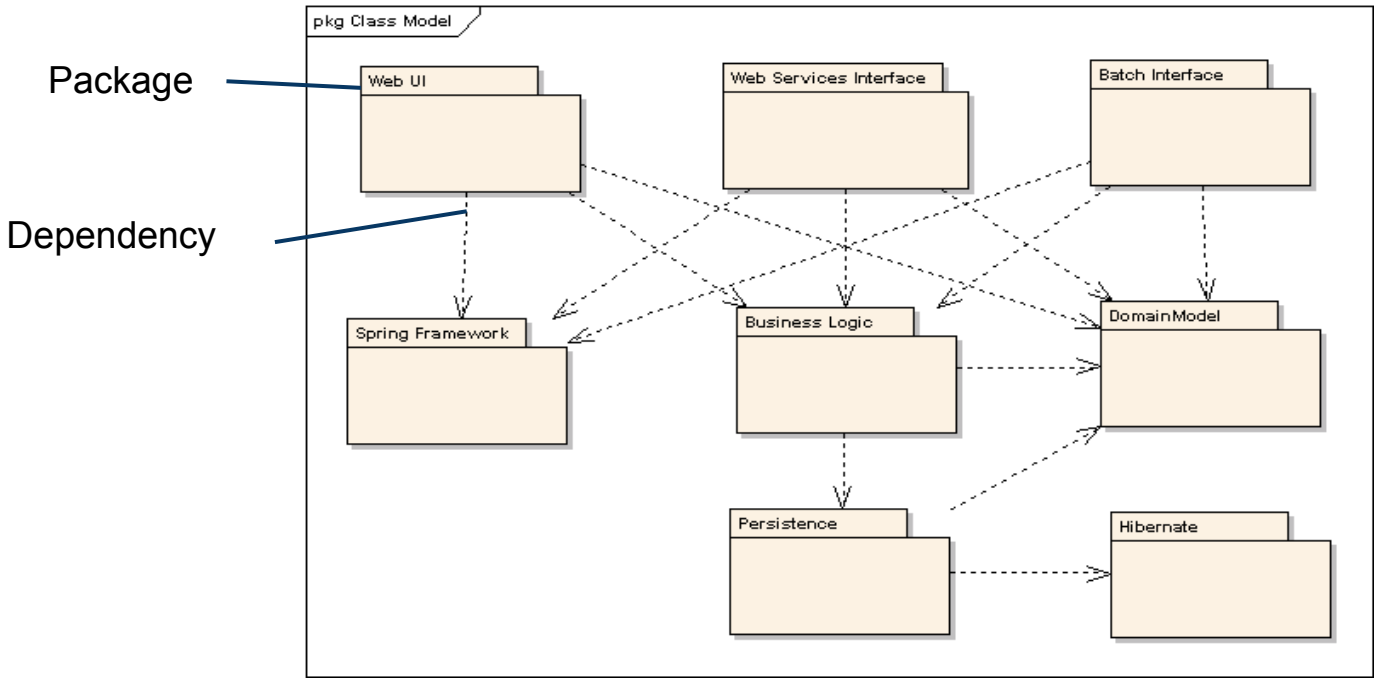
- Close to GUI, possible to show dialogs etc.
- Avoid checked exceptions (possibly by tunneling = wrap checked exception in runtime exception, rethrow)
- **NEVER** empty catch (exception swallowing)!

Possibly central exception handler?

Package Diagrams

Design uses package diagrams (preferable with dependencies)

- Can't have all in same class diagram
- Need a "higher level" view of application



Top level package missing

Documenting the Design

Design documented in Software Design Document (SDD)

- Design model (using package diagram(s))
- One class diagram/"interesting package" (i.e. not GUI, class diagrams with arrows)
- At least 2 sequence diagrams (in appendix)
  - Sequence diagrams very time consuming, 2 will do
  - Note: Sequence diagrams also expanded when GUI added (also later when full MVC model) . Possible have to partition into more diagrams.
- Any kind of other high level information easing the understanding of the application; layering, MVC-style, service, use of design patterns, data formats, interfaces ...

SDD, updated after each iteration

MP : SDD

SDD on course page, inspect

- Not in sync with code, just a general example
- The usual problem …

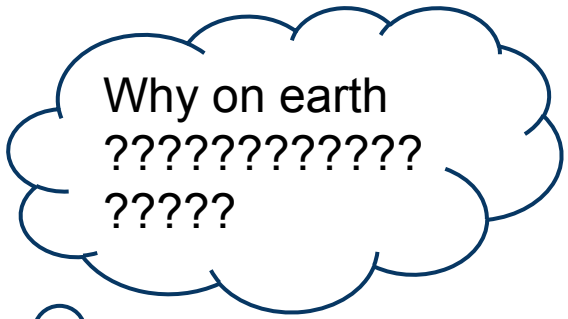
Documenting the Implementation

No documentation of implementation in SDD

- Code it the ultimate low level documentation
- Tests also counts as documentation
- During implementation put comments in code when it's hard to understand/technically advanced/unusual ...

```
//Worst comment ever
x = x + 1;  // Increment x with 1
```

- NO Javadoc needed.




```
if (this == null) {
    return 0;
}
```

Summary

We started with a running version of model
Relaying on design principles, best practices, refactoring, tools, testing we

- Expanded the model
- Implement a full MVC application with the model as the core
- Added a service
- Todo: More UC, more GUI, more services, exceptions, resource handling, …

Should, from here, be able to quickly
(and controlled) expand the
application until finished criteria
from RAD fulfilled

Hmm...


Our Disaster Recovery Plan
Goes Something Like This...
HELP!
HELP!