Iteration 2 [All phases]

Slide Series 6

Process Recap

1. Requirement Elicitation
   -What are we going to build?
2. Analysis
   Build a model for the problem
3. Design
   Adapt (parts of) model so it can be implemented
4. Implementation
   Implement (parts of) model

We'll focus on this

Iteration 2: Requirement Elicitation

Hmmm... possible have missed something?

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

Iteration 2: Analysis

If changes to RE probably changes to analysis parts

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

Iteration 2: Design and Implementation

So far ...

- Have a basic design model (design model = runnable version of analysis model)
- Have a pair of UCs running (see MP 0.1 on course page)

Design implementation objectives iteration 2

- Implement a few more more use cases
- Design review of model
- Testing and Coverage
- Overall design
- Adding a GUI
- Possible some service

MP: More Use Cases

We'll implement UC's; Buy and Sell

- Have to start distinguish different Spaces. Will subclass Space
- More detailed analysis (update RAD) and design models
- Try to identify an interface to model …

Inspecting MP 0.2 (on course page)

Design Review of Model

- Every class has well defined responsibility (represents one concept)?
- Possible: Split or collapse classes, introduce generalization
- Possible missing or unnecessary classes (convert to attribute)
- Directions of associations (was it awkward to implement the UCs?)
- It should not be possible to form a cycle traversing the associations (no mutual associations). Introduce classes to break cycles (if A depends on many Bs and B on many As introduce association class)
- Model in one package, interfaces in model classes mostly for technical reasons (no info hiding)
- Interface(s) to model (model package) to use by others. Facade pattern

MP: Design Review of Model

Known issue

- Heterogenous collections for Spaces. Many kind of spaces handled in single list (easy for GUI to render)
- But also need to distinguish spaces (can't buy/sell Chance or Tax)
- Will avoid **instanceof** (generally bad, better with polymorphism)

Solution

- Visitor pattern
- See **MP** 0.3

Parameterization of the Model

The model normally has quit a few parameters

- In MP; number of players, how much money for each player, number of spaces of board,...)

Possible to have model parameters in class parameter belongs to but...

- Parameters possible set from GUI
- Parameters spread out all over

MP Design decision (as presented earlier)

- Use dedicated options class

Testing the Model

When model starts stabilizing we start constructing tests for non-trivial classes

- Unit testing (single class), possible some integration testing (more classes)
- We use JUnit

Methods to test

- We test public methods (if need to test private possible a design flaw)
- If void method must have (introduce) get method to inspect state

To make it possible to test we must design for testability

- Minimize dependencies
- Avoid inline use of static classes, Singletons or any global dependency
- No inline use of **new**
- All dependencies passed in via constructor (later possible to mock)

A Few Notes on Test

[Naming of tests]

[The A-A-A pattern]

[A-TRIP]

[Stack overflow: What make a good test?]

Code Coverage

Testing using code coverage will give much higher quality

- During testrun, let some utility record all methods and statements executed
- Code not run in test is not tested, … have to write tests that cover high percentage of code. Review tests!
- Plugin to Netbeans, [Jacoco] (need modification of pom.xml, see web)

After previous steps the model should be in a good state, nice design and tested. So leave it for now and start design of full application!

Design Goals for Application

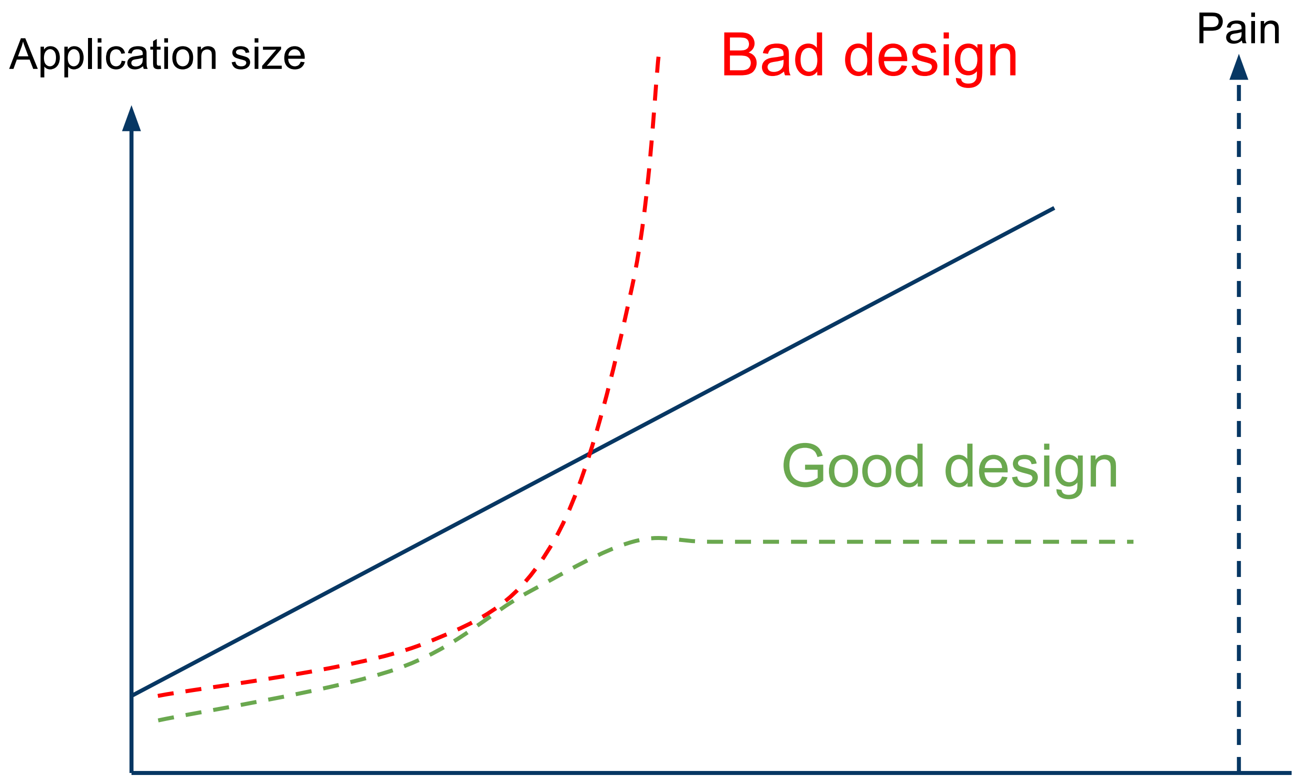
Over all design goals

- Create an identifiable program structure
- Enforce localization of responsibilities (core functionality in model)
- Minimize dependencies
- Control (minimize) state

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

- Testability on previous slide

The Impact of Design



On which curve are we..?

Design Principles and More

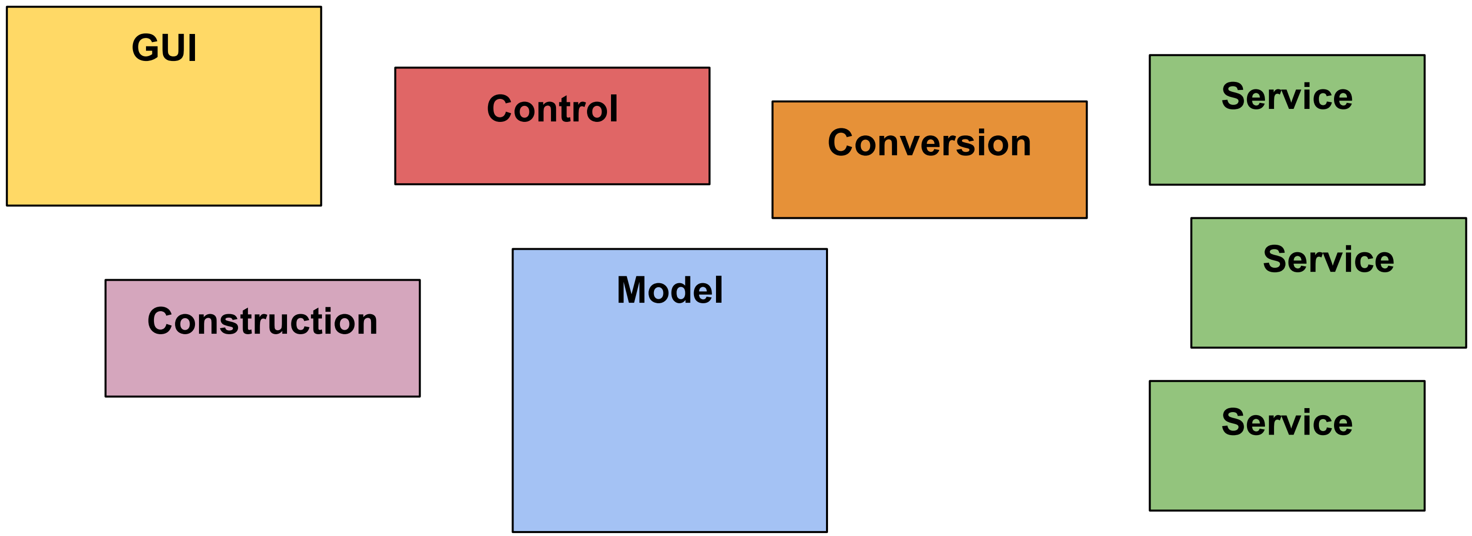
Some considerations

- 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? Remember Swing is single threaded!
- <u>MVC (must use, some kind of)</u>

During our work we must keep an eye on this, if violating, <u>refactor (and test)!</u> Run tools!

Design patterns possible can help ...

Design Pieces



How to get a good overall design out of this?

Construction

As noted we separate out construction
- Normally using factories or Factory method
- Also Builder pattern

Construction in **MP**
- Have already introduced Factory for model
- Possible Factory to build GUI, …?

Model and GUI

GUI requirements

- As noted: The GUI should be composed (not one single class)
- Possible a Factory to construct the GUI
- Listeners should do as little as possible i.e. call model or control. Possible do some updates of GUI (a local update just involving this class)
- No GUI code in model
- No model logic in GUI. Components enabled/disabled because of model state, not as a means for the GUI to control the state

Must use some kind of MVC - model

- GUI is a view of the model
- GUI is input to model

Implementing MVC

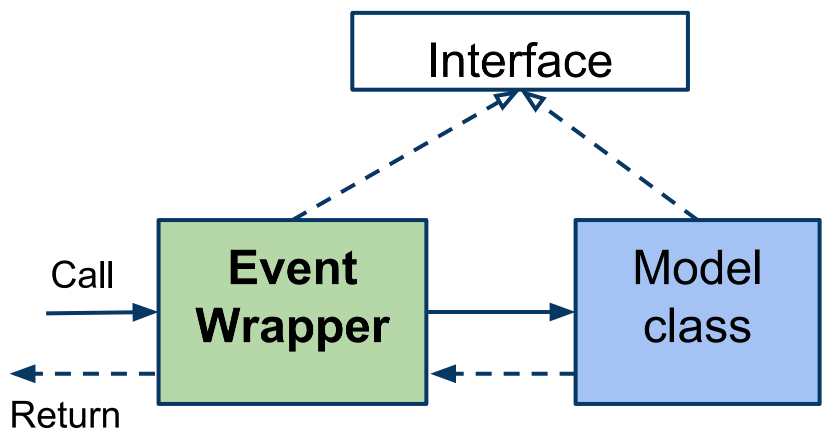
MVC not uniquely defined (this is one approach)

- Many classes in GUI need to access model (or some controller)
- Model need references to observers
- Messy to connect (Where in code? Threading references …)
- … a event based design far simpler. Let all connect to a common eventbus (as senders or receivers)
- Will decouple model and GUI even more

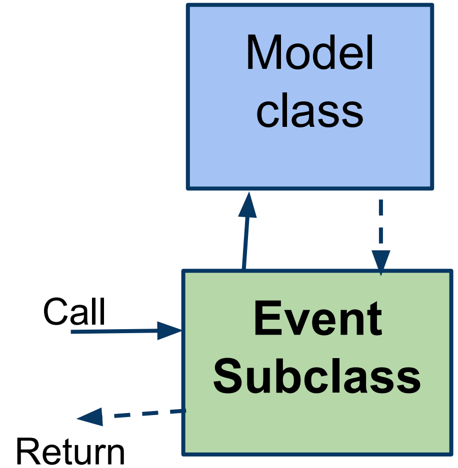
More on Control later …

Connect Model in Event Based Design

Should adhere to Open Closed Principle (code closed for modification open for extension)
- So shouldn't modify code in model, should add ...
- To possibilities to extend model with events

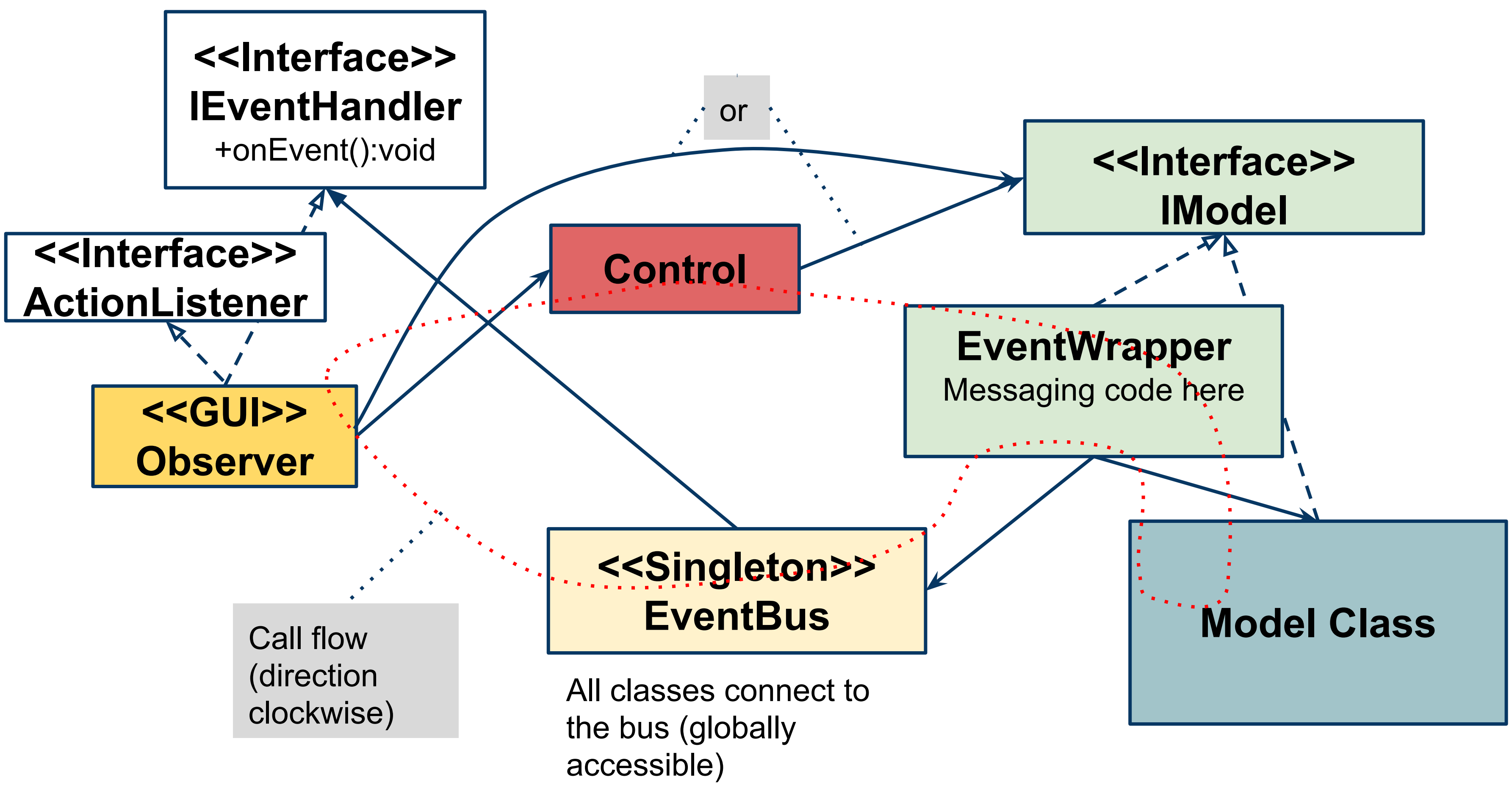


Alt 1: Event code in EventWrapper core functionality in Model (Wrapper delegates)



Alt 2: Event code in subclass core functionality in Model (Wrapper delegates)

Event Based Design of MVC



MP: Inspection of first GUI version

See **MP** 0.4

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

Aside: Swing Issues

Swing is a complicated creature, sometimes it just doesn't work ...??!

- Force repainting using; validate(), repaint() or possible both
- Use setSize() <u>or</u> pack()
- Some components fire events when data changed! Possible 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

Controlling Dependencies

Having implemented a larger part of the application need to control dependencies

- Now we focus on dependencies between packages
- No mutual or circular dependencies
- Use tools to check
- STAN
- … other quality tools: PMD plugin to NetBeans (= standalone application Findbugs)

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
- Communication, Network, …
- …

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
- Interface segregation principle!

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

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

Typically you <u>don't</u> implement subsystems for

- Graphics and sound, physics engines, .databases, data handling, XML, ...networking, ...

... find it somewhere! Look for high level abstractions

- Databases to Objects, very hard, use JPA, Hibernate, ...
- Network
    - Bad: Sockets, too low level.
    - Better: XML-RPC, KryoNet, RMI, ... other, ... much better (have protocol in place)

In-house Service Design

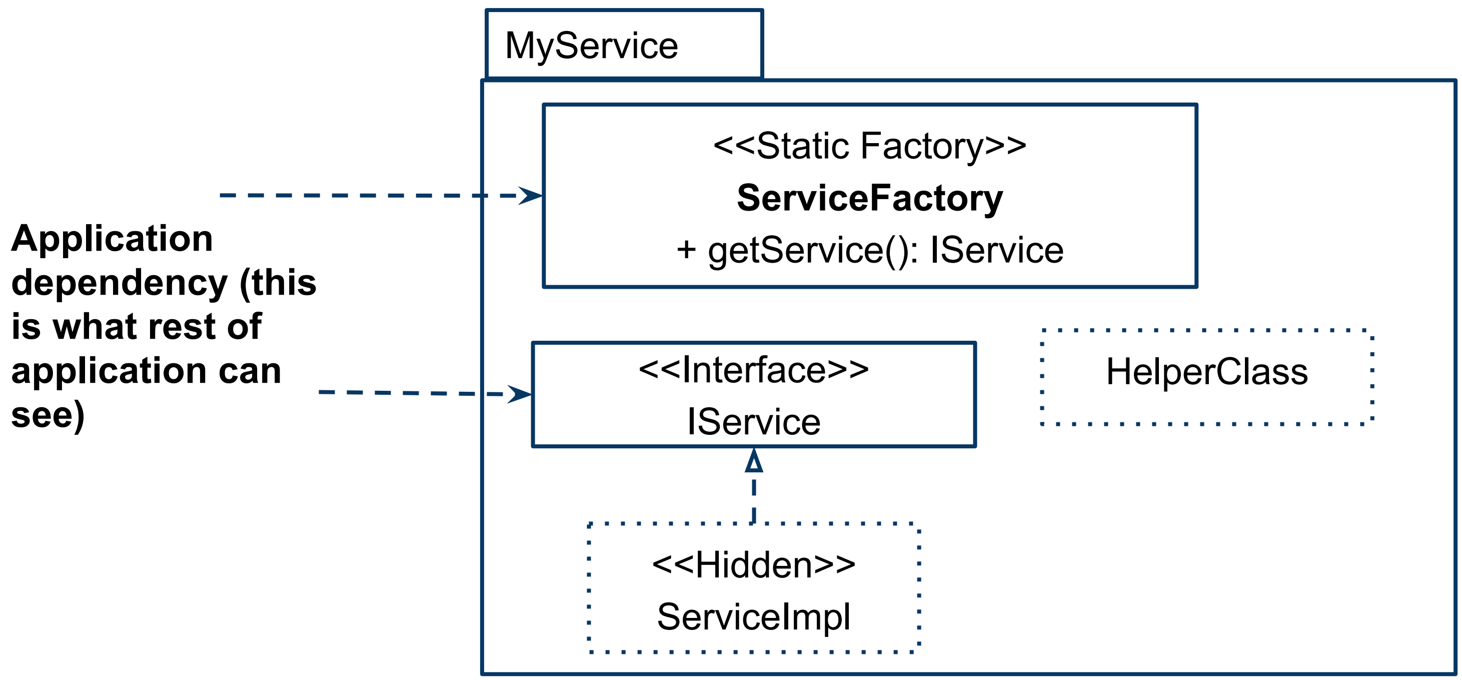
When design a service often beneficial to split services into steps

- 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



Services and Testing

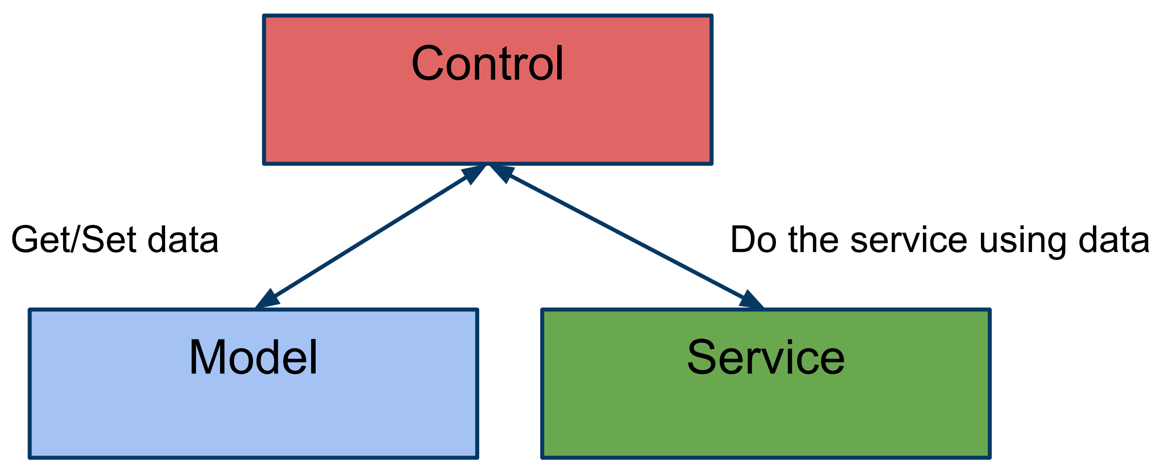
Of course we write tests for the service

- Must know service is working before integrate into application
- Services are well suited for TDD, should have few dependencies on model, possible parameters/return values Should be possible to test in (close to) isolation
- If problems with dependencies use layering inside service
- Possible use of generics can remove dependencies

Model and Services Interaction

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

- Put interaction model - service in control layer



MP Services and Control

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** 0.5

More on Control

Use of control layer

- As seen: Interaction with Services
- Any more complex interaction GUI - Model (Wizards)
- Possible temporarily storage, going from GUI to model (collect data before updating model, so we don't need to restore model if some data wrong)

Exception Handling

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

If <u>not</u>, ... handle in view or control layer

- Close to GUI, possible to show dialogs etc.
- Avoid checked exceptions (possible by tunneling = wrap checked exception in runtime exception, rethrow)

Possible central exception handler?

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

Interfacing with Other Paradigms

All the way we have been using the OO paradigm

All software is not OO...!

- 3d graphics is built on an rendering pipeline (have to adapt MVC model)
- Relational databases are not OO they are sets!
- The web is not OO it's just strings (don't use any web in this course)

If using any above have to incorporate different paradigms in one application

- ...hard... (often have to put adapting layers (interfaces) in between)
- Not covered in course

Documenting the Design

Design documented in Software Design Document (SDD)

- Design model (using package diagram(s), following slide)
- 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

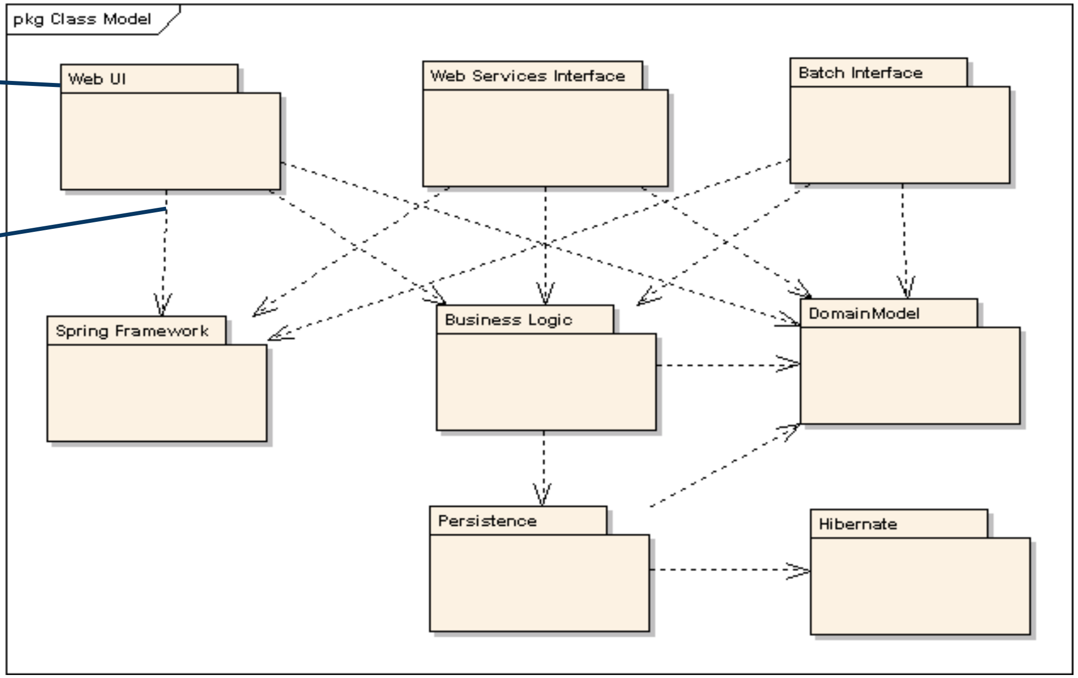
Package Diagrams

Design uses package diagrams (preferable with dependencies)

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

Package

Dependency



Top level package missing

MP : SDD

SDD on course page, inspect

- Not in sync with code, just a general example
- The common problem...

Documenting 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 <u>hard to understand/technically advanced/unusual</u> …

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

Why on earth
????????????
?????

- <u>NO</u> Javadoc needed.

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

A Note on Benefits of OOA/D/P: Tracing

It should now be possible to trace the development

- Running objects (classe) should be able to trace backwards, possible all the way to the UC's
- Because we use the domain language, names should be the same all the way, classes, attributes should be evident, they have a meaning in the real problem (domain)

Summary

We started with a running version of model

Relaying on design principles, best practices, refactoring, tools, TDD we

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

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

Done…!

Hmm...


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