Design

Slide Series 5

Content

Design goals
Design principles
Single responsibility Principle, KISS, ...
Design of methods, classes, ...
Canonical form (part I)
Service Modules
Facade, Proxy, Observer Pattern
Model dependencies, messaging
MVC model
Exceptions

Word of Wisdom

"Perfection [in design] is achieved, not when there is nothing more to add, but when there is nothing left to take away." // Antoine de Saint-Exupéry

Design goals

In this course

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

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

Design Principles

Software design principles represent a set of guidelines [no laws] that helps us to avoid having a bad design

- Important to notice that: do not shift all the principles to extremes, because in real cases is impossible to achieve them from all point of views

Have seen some

- Interface segregation principle
- Dependency inversion principle
- .. more to come: Single responsibility principle

Design Principles, cont

Two often mentioned collections of principles

SOLID by Robert C. Martin (early 2000's, hmm...)

GRASP by Craig Larman

Principles overlap, also with other notions (design patterns)

A general approach is KISS

Single Responsibility Principle

Everything should have one single (well defined) responsibility

At a basic level this means: Attributes, methods, classes and modules

- If you can't find a good attribute/method/ class-name possible violating SRP
- Easy to state, sometimes hard to implement
- Aka Separation of Concerns

Reasons for SRP

Will reduce dependencies and

- If many responsibilities the "reason to change" increases (if changing one responsibility in a non-SRP class, possible other responsibilities affected). Violates OPC-principle
- Easier to understand (smaller more focused)
- Easier to combine with other (easier to replace)
- Easier to reuse

Examples upcoming...

Refactoring

"Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"

// Martin Fowler

Techniques

- Split, collapse classes/interfaces/methods
- Replace classes with interfaces
- Move methods between classes
- Move classes/interfaces between modules
- ... (many supported by Eclipse)

Design Levels

Design is a multilevel activity

- Attributes
- Methods
- Classes
- Class aggregates or similar
- Modules
- Application
- System … (a small one in this course)
- Systems … (not in this course)

Design Levels: Attributes

As few as possible (reduce state)

One attribute for each fact (no shared)

Always accessed with set/get (if really needed, avoid set/get,)

```
public class MyClass {
    public int i;    // Bad not possible to do i < 0
}                     // any other check/action/...
```

Design Levels : Methods

Have seen

- Handling nulls, in- and out
- Reasoning using invariants
- Mutators vs. Accessors

Other issues

- SRP/Law of Demeter
- Limit number of parameters (prefer objects to primitive types). Have impact on testing (combinatorial explosion of test cases)
- Closureness (int->int->int)
- Returning "this", booleans as signals, ...

Principle of Least Surprise

"The design should match the user's experience, expectations, and mental models."

"In more abstract settings like an API, the expectation that function or method names intuitively match their behavior is another example. This practice also involves the application of sensible defaults." // Wikipedia

How does your methods behave??

System.in is an InputStream and the read() method definition says it returns an int. But calling System.in.read() does NOT return the integer you type at the keyboard.

Design Levels : Classes and Aggregates

Have seen

- Reduce state
- SRP/Coupling (dependencies)/Cohesion/avoid new
- Reducing associations (dependencies)
- Reasoning using class invariants (no representation exposure)
- Failure Atomicity
- Single entry points (aggregate roots)

Other issues...

Class Design: Information Expert

"Information Expert will lead to placing the responsibility on the class with the most information required to fulfill it"

- The class has the data (the knowledge), it should perform operations on the data ... not just pass it around to others classes (i.e. set/get)
- Similar to info hiding, SRP

Design Levels : Interfaces

Have seen

- Interface segregation
- No creation methods in interfaces, separate construction from use
- Construction is part of the implementation not the specification (use factories)

Classes: Canonical Form

Issues common to all objects of all classes

- HashCode (fairly unique numerical id)
- Equality
- Copying
- Utilities, string representation
- Serializing

Many of these have default implementations in java.lang.Object

Classes: HashCode*

*"In the Java programming language, every class must provide a **hashCode()** method which digests the data stored in an instance of the class into a single hash value (a 32-bit signed integer). This hash is used by other code when storing or manipulating the instance"* // Wikipedia

- Used with HashMap and other collections
- Default implementation in java.lang.Object (so inherited by all)
- Let Eclipse generate!

Classes: Equality*

Very common to be able to compare instances

- Default implementation in java.lang.Object.equals() uses == (so by reference)
- Normally "by values" more useful
- If so; have to override equals()-method (tricky when inheritance, more to come...)
- Let Eclipse generate
- Two objects which equals() says are equal must report the same hash value (else equal objects possibly end up in different locations and thus not found).

Classes: Copying*

Also very common to produce a copy of an instance

- Default implementation in java.lang.Object.clone() creates a **shallow copy** (i.e. copy by value, any attribute value is copied)
- Will create shared references
- If other behaviour **(deep copy)** must implement own clone() (overriding and make public)
- Tricky and strange, must handle unnecessary exception, implement marker interface and use a specific "idiom", problems with final. No constructor may be involved !?!... see later at inheritance...

Classes: Other Issues

Besides equal often need less than/greater than (sorting etc.)

Two standard interfaces

- java.util.Comparable
- java.util.Comparator

Comparable*

```
// Only method in Comparable (result = r)
// r < 0 (less), r == 0 (equal),  0 < r (bigger) than
other
// object
public int compareTo(T t){
...
}
```

Usage: The type has a "natural" ordering

Many standard classes implement (String)

```
// Usage (aCollection implements comparable)
Collections.sort(aCollection);
```

Comparable and Equals*

Should be consistent

- If equals() true then comateTo() should be 0

"This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals" // Javadoc Comparable

```
// Inconsistent equals and compareTo (says API but..?)
BigDecimal b1 = BigDecimal.valueOf(4.0);
BigDecimal b2 = BigDecimal.valueOf(4.00);
System.out.println(b1.compareTo(b2) == 0); // True
System.out.println(b1.equals(b2)); // False
```

Comparator*

Comparable have to decide outcome at class creation (comparision hard coded). Comparator is more flexible, passing in a comparator to sort methods

```
// Same outcome as comparator
public int compare(T t1, T t2) {
...
}

// Usage (possible to dynamically sort)
Collections.sort(a, myComparatorObject);
```

Module Design

Have seen

- Info hiding/cohesion/coupling
- Unified interface
- Stateless vs Stateful module

Stateful vs Stateless Modules

In a stateful subsystem data is remembered between calls

- Normally need to call methods in specific order (a dependency)
- Very careful design of API, how to react if methods called in wrong order (IllegalStateException)?
- Avoid

Compare: Trie/Dictionary-module vs Translator

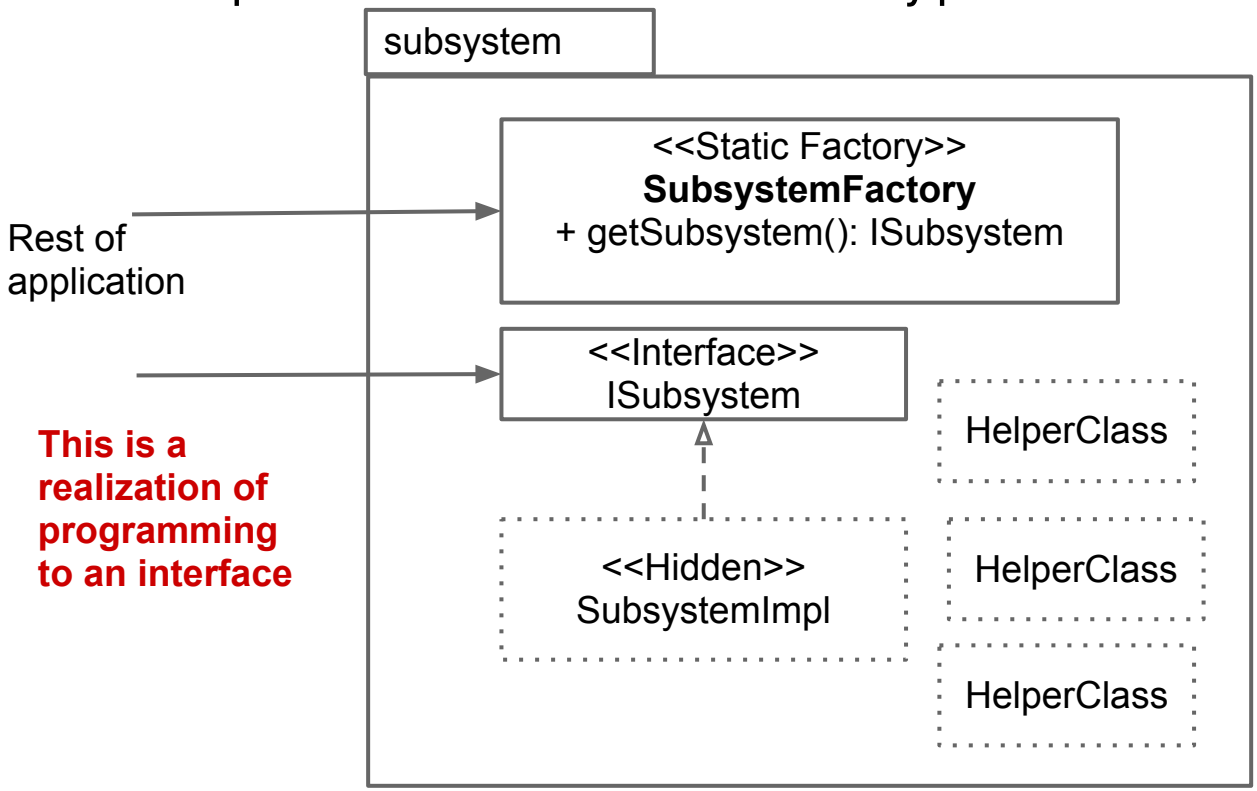
Implementing a Service Module

Service module = module with unified functional interface (SRP, low coupling, high cohesion)

Standard implementation technique, use the **Facade design pattern...**

Facade Design Pattern*

Only visible types are the factory and the interface (possible parameter and return types)



Model Dependencies on Services

Model classes don't handles technical services (they represent a model of the problem). SRP

```
// Model classes don't handle services, bad
modelObject.save();

// Use a service module!
fileService.save(modelObject);
```

... but some services intrinsically in model...

Services in Model

Need service in model, but SRP/OPC says... no!

- Logging
- Messaging!

....How to...??? Subclassing possible ... more to come...

Proxy Pattern*

Proxy pattern lets one object "stand-in" for another

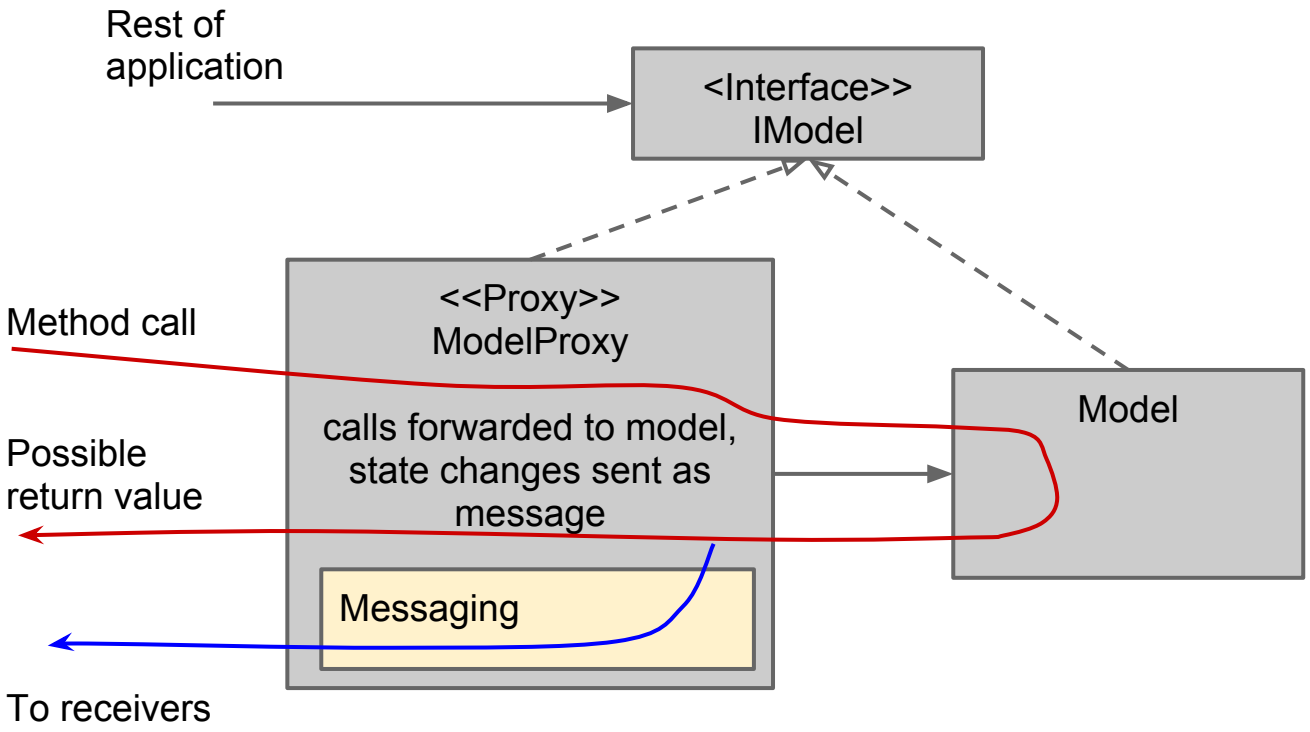
- Both have same interface

Types

- Virtual Proxy, lightweight stand-in for some resource intensive object (image)
- Access Proxy, controls access to resources (UserProxy)
- Remote Proxy, represents an object somewhere else (proxy on client, real object on server)

Using a Proxy for Model Messaging

Avoid messaging code in model



Applications with Graphical User Interfaces

Designing GUI applications non-trivial

- Is GUI a service (in/output to/from model)?
- Or is Model a subsystem used by GUI?
- How to control dependencies (GUI often has a lot of administrative (trivial) code).
- How to control the flow?
- ...

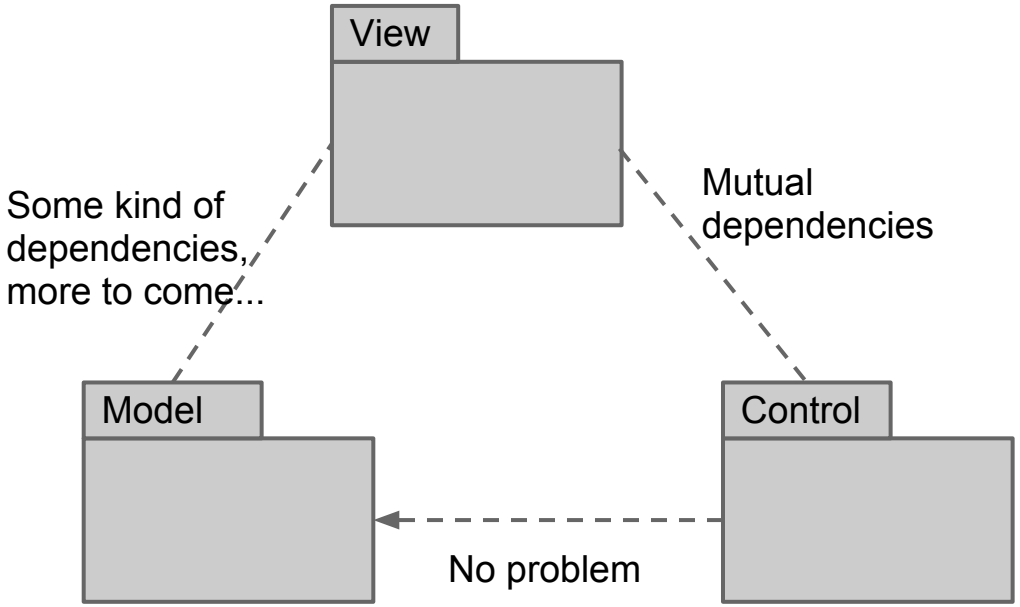
Model-View-Control

Common solution for GUI applications (MVC-design (architecture)). Partitioning application into

- View, the GUI, a view of the model (this part is very transient, frequently changed, also possible completely different technologies, PC/Mac-app, smartphone, web)
- Model, the OO-model (and associated services). If done correctly this should be pretty stable.
- Control, parts that coordinate the interaction between GUI and Model

Model-View-Control, cont

Basic parts and dependencies



Aside: Readonly

View will display state of the model

- Shouldn't be able to modify model
- Would be very nice if we had "read only" references (but we don't)
- Immutable or Frozen no problem (but in general a too hard restriction (must copy all time))

No default support....

MVC and the Model

View and Control could see Model is a stateful subsystem

- Model has interface
- Each (non read) call to model transform state (one finished complete transformation)
- After call possible to inspect new state (and act accordingly, possible display)

MVC and the Model, cont

```
// Imagine something like this
while( notFinished){     // Loop act as control
    indata = view.getIt();
    model.changeState(indata);
    outdata = model.getViewOfState();
    view.display(outdata); // Done indirectly (not like
this)
}
```

Notes:

Model as a stateful subsystem

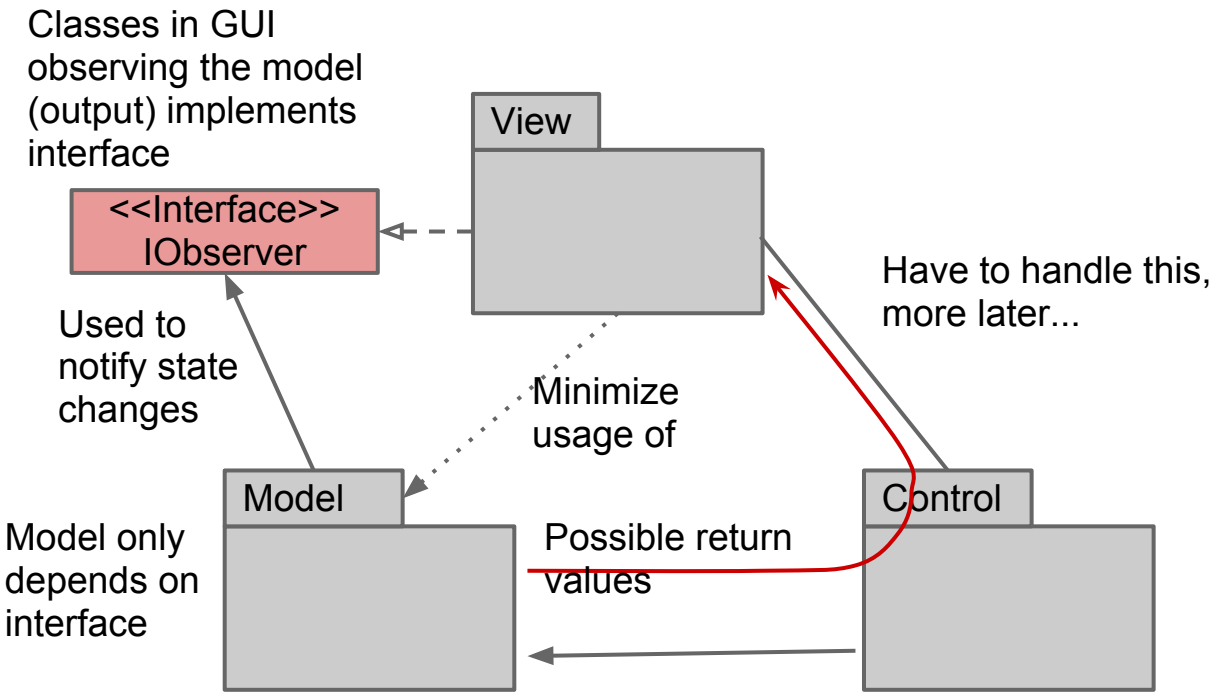
Should be possible to "run" model without GUI

Indata possible from many controls (toolbar, menu, button)

Outdata possible displayed in many different gui locations (display content, status bar, dialogs, ...)

GUI can enable/disable but logic in model

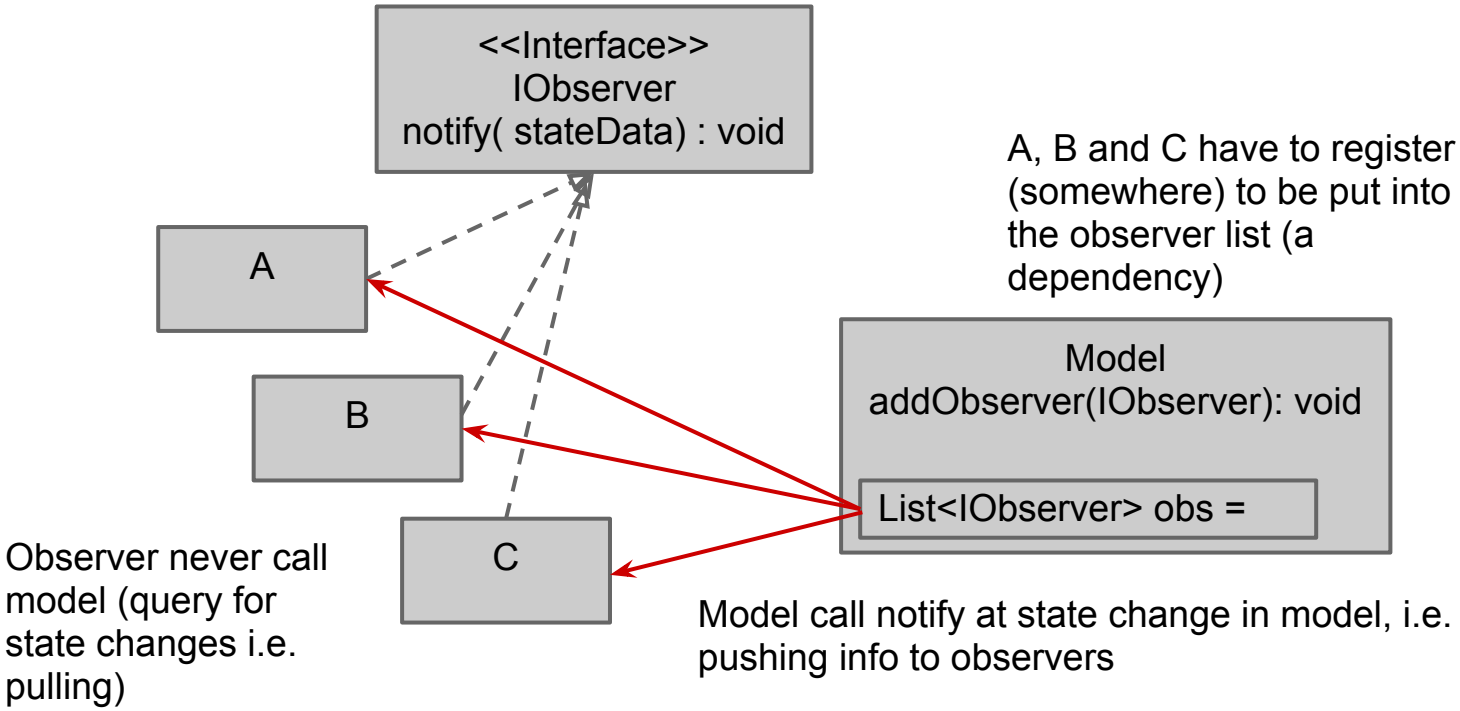
MVC Design using Observer*



Observer Pattern

Decoupling observers from the observable

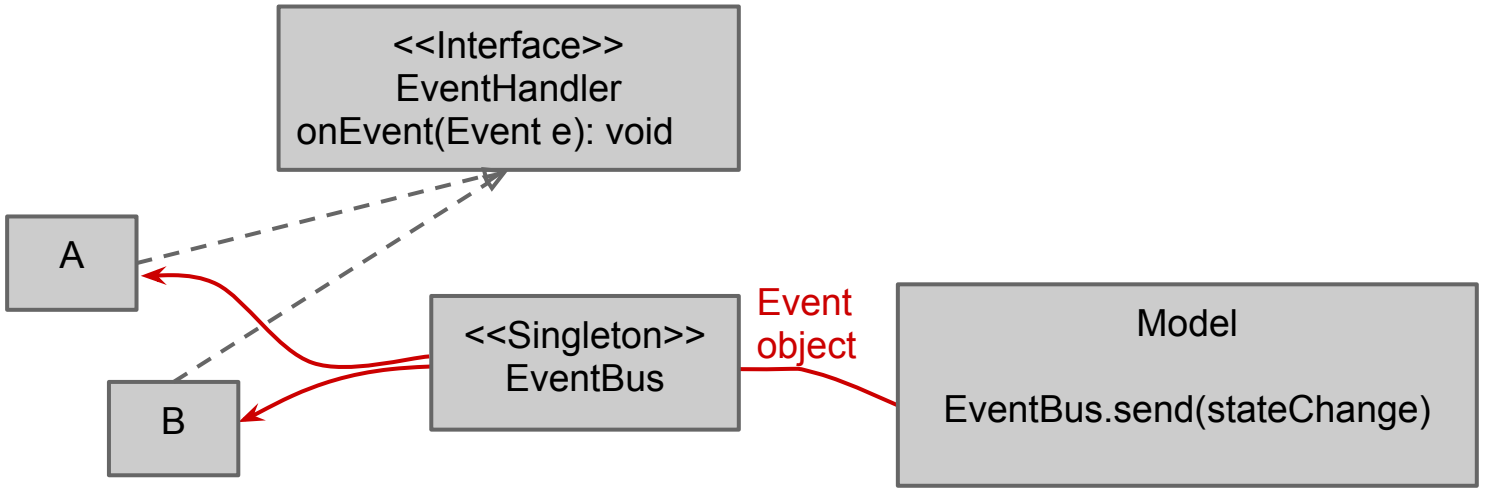
- Push design (vs pull design)



Observer: A Variation*

Use messaging (implemented as an EventBus)

- Will remove any direct dependency Observer <-> Observable



Somewhere: EventBus.register(A) and EventBus.register(B)

Aside: Messaging Frameworks

There are existing "frameworks" to support messaging

- Google Guava (EventBus also)
- Java context and dependency injection (aka CDI, aka Weld)
- Will also reduce dependencies, construct application (using @annotations)

If building real application should inspect these or similar

MVC Implementation

Model to view: Observer (possible Model-proxy and EventBus)

Control to model: No problem, control has direct reference to model, model never has (needs) reference to control.

View to Control: Tricky

Control and View

Controls need input from view (so V -> C)

Controls possible need to manipulate GUI

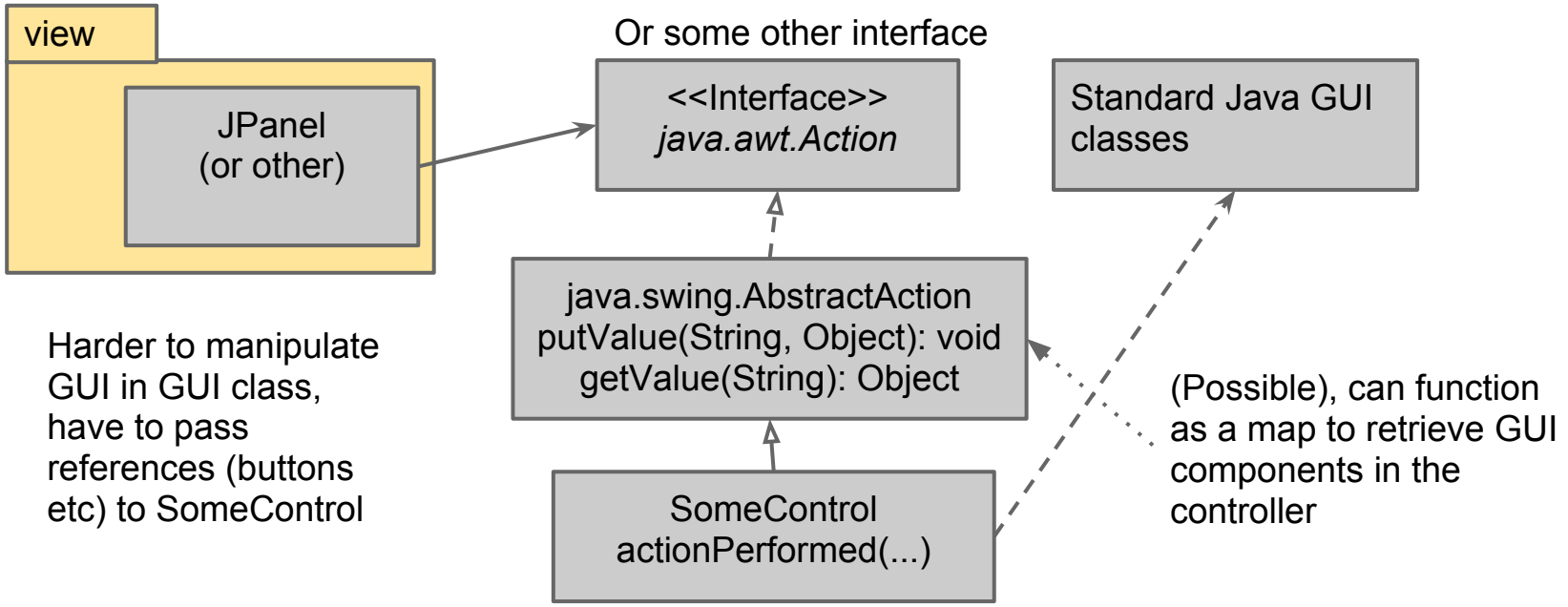
- Enable/disable button, etc.
- If control methods has return values (so C -> V)

Possible implementations (remove mutual dependency)

- Let controls be listeners. GUI dependencies on interface. Possible to connect to buttons etc.
- Let controls use interface (IOutputable or similar) to GUI (or standard Java classes like JButton)

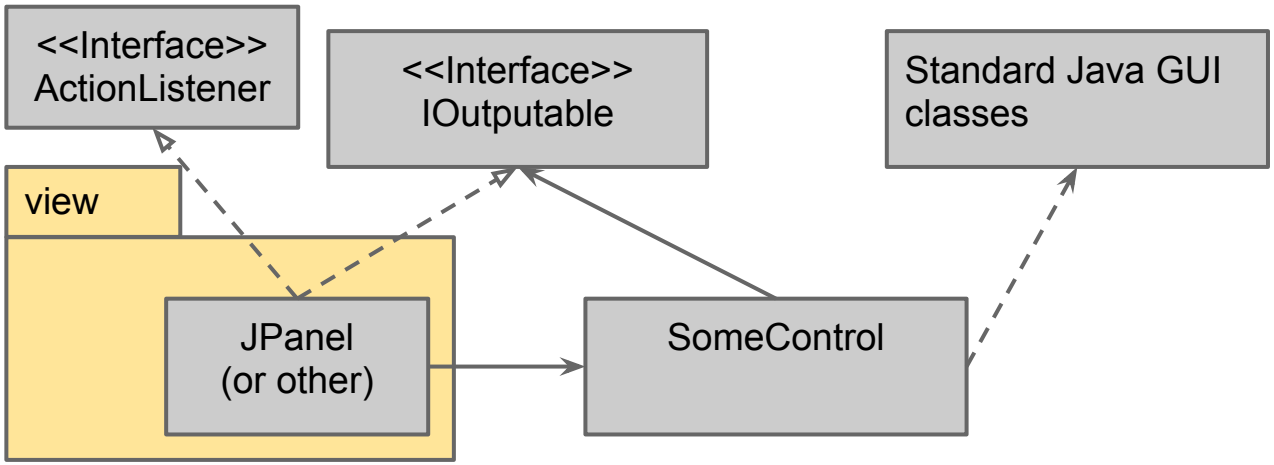
Control and View: Implementation 1*

Controls acts as listeners, connect directly to buttons etc. in GUI



Control and View: Implementation 2*

Have listeners in GUI, listeners calls controls



Possible to manipulate GUI in listener before/after calling control

Division of Labor: Control vs. Model

How much should be done by control vs. model

- **Anemic model:** All work in control. Model pure data (violates information expert)
- **Fat model:** Most of (all) work in model. Normally need thin abstraction layer over model (control). Something manipulating the model
- **Divided:** Some parts in control others in model. Have to use your skills...(principles, coupling, cohesion, abstraction, ,...).

Division of Labor: Example

WordFeud, fat or anemic?

Do it in Control? …

or much in Model?

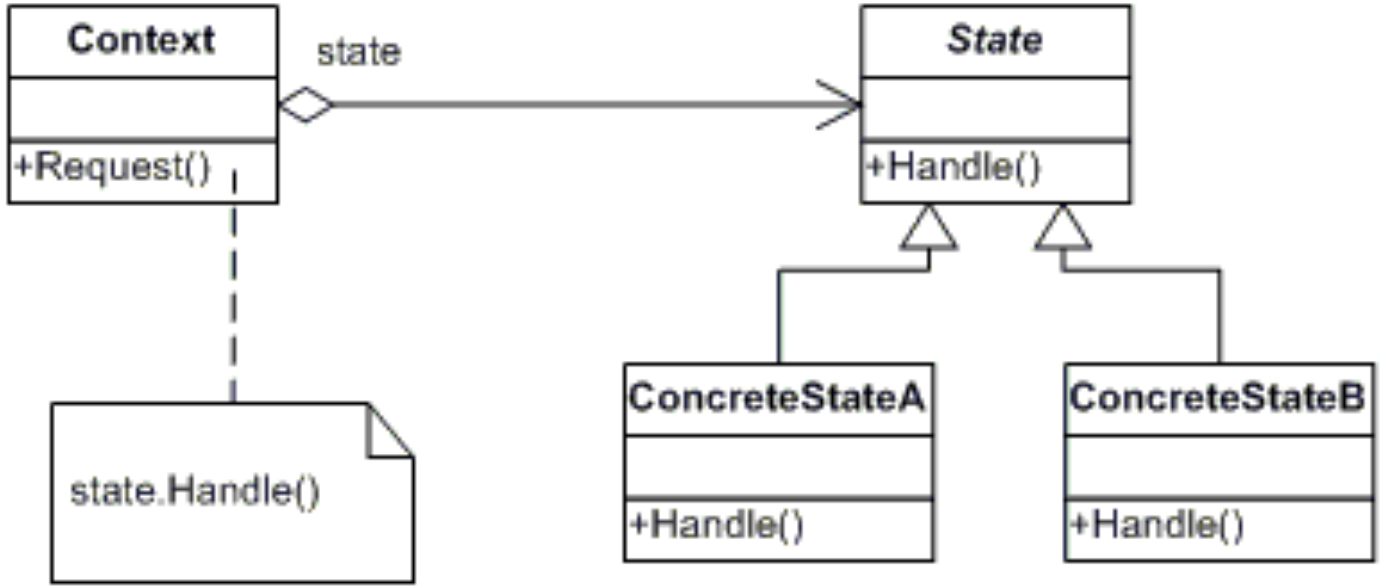
Constructing a MVC Application

1. Model (relevant services) constructed (possible using a factory) in public static void main... (using a Swing-idiom, because of threads, more later...)
   a. ...or just construct application main window (later call to File > New to construct model and GUI, use special StartControl)
2. Let GUI factory construct the GUI, pass in model interface
3. Let GUI factory connect model, control and GUI and return the GUI
4. Often need EndControl to clean up save options etc.

Use factories!

Advanced Control Issues

State pattern*



Exceptions

Large, difficult not very well understood topic

Exceptions used for "exceptional" event not for control flow

- A datafile is missing.. program probably can't handle, exception ok
- Looking for an element in a list, it's not there.. possible to handle, ... no exception
- If looking for an elements using an invalid index... collection (ArrayList) will throw Exception, ok

Exceptions vs Return Values

```
// Using return values will clutter up code
r1 = s.call();
if( r1 != null){
    r2 = r1.call();
    if( r2 != null ){
        r3.call();
        ...

        }else {
        ...
    }else{
    ...
}else{
...
```

AVOID!

Java Exception Handling

"...the Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer."

"An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred"

"Every exception is represented by an instance of the class Throwable or one of its subclasses"// JLS 11

Java Exception Handling, cont

"During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution … This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception"// JLS 11

If no handler found program terminates

Runtime Handling of Exceptions*

"When an exception is thrown (§14.18), control is transferred from the code that caused the exception to the nearest dynamically enclosing catch clause, if any, of a try statement (§14.20) that can handle the exception."
/ JLS 11.3

So will possible jump through many method calls and end up in very different part of program (**non-local transfer**)

Basic Exception Example

```
// A methods that throws (using a throws clause)
public void doIt() throws IOException {
...// File not found will generate IOException
}


// Handle exception
try{
    // Somewhere, possible far away o.doIt() may throw;
    // No statements after executed if an exception!
} catch (IoException e){  // A handler, exception caught
    // Try to recover...
}
```

Causes of Exceptions

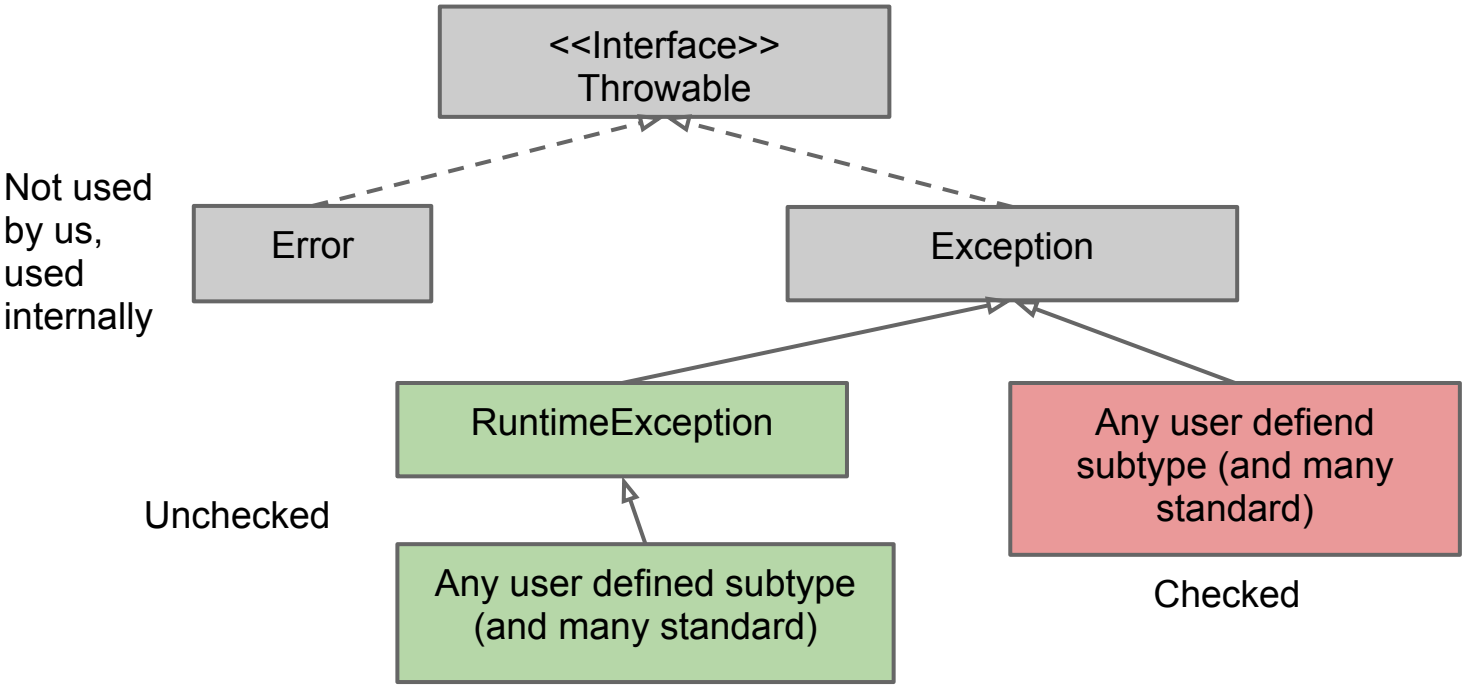
Causes

- An abnormal execution condition was synchronously detected by the Java virtual machine
- A **throw** statement was executed

```
// Explicit generation of exception
throw new IllegalArgumentException("He's dead");
```

Java Exception Types

For all classes: Constructor take String argument specifying cause of exception.
Retrievable with e.getMessage()



Checked vs. Unchecked Exceptions*

"The unchecked exception classes are the runtime exception classes and the error classes. (color as previous slide)"

"The checked exception classes are all exception classes other than the unchecked exception classes. That is, the checked exception classes are all subclasses of Throwable other than RuntimeException and its subclasses and Error and its subclasses (color as previous slide)."

"The Java programming language requires that a program contains handlers for checked exceptions which can result from execution of a method or constructor" // JLS 11

Checked exceptions checked compile time!

Exception Compile Time Checking

It is a compile-time error if a method or constructor body can throw some exception class E when E is a checked exception class and E is not a subclass of some class declared in the throws clause of the method or constructor.

It is a compile-time error if a class variable initializer (§8.3.2) or static initializer (§8.7) of a named class or interface can throw a checked exception class.

It is a compile-time error if an instance variable initializer or instance initializer of a named class can throw a checked exception class unless that exception class or one of its superclasses is explicitly declared in the throws clause of each constructor of its class and the class has at least one explicitly declared constructor.

Note that no compile-time error is due if an instance variable initializer or instance initializer of an anonymous class (§15.9.5) can throw an exception class. ... much more...

It is a compile-time error if a catch clause can catch checked exception class E1 and it is not the case that the try block corresponding to the catch clause can throw a checked exception class that is a subclass or superclass of E1, unless E1 is Exception or a superclass of Exception.

It is a compile-time error if a catch clause can catch (§11.2) checked exception class E1 and a preceding catch clause of the immediately enclosing try statement can catch E1 or a superclass of E1. // JLS 11.2.3

The (Checked) Exception Debate

Most languages don't have checked exceptions

Pros checked exceptions

- Designed to reduce the number of exceptions which are not properly handled
- Part of the contract between the implementor and user of the method or constructor

Cons checked exceptions

- They often pollute APIs. Exception may accumulate (very many exceptions in throws clause).
- Not part of signature but affects the API (see override, upcoming)
- Checked exceptions make sense only when there is a clear and documented way to recover from the exception
- Exception swallowing (effectively cancel the first Pros point)
- Non local jumps (similar to goto)*

Exceptions: Overriding*

"A method that overrides or hides another method, including methods that implement abstract methods defined in interfaces, may not be declared to throw more checked exceptions than the overridden or hidden method."

"More precisely, suppose that B is a class or interface, and A is a superclass or superinterface of B, and a method declaration n in B overrides or hides a method declaration m in A. Then:

- If n has a throws clause that mentions any checked exception types, then m must have a throws clause, or a compile-time error occurs.

- For every checked exception type listed in the throws clause of n, that same exception class or one of its supertypes must occur in ... the throws clause of m; otherwise, a compile-time error occurs. // JLS 8.4.8.3

General Form of try

```
// General form
try( some AutoClosable resources ) {
    // Possible exception;
} catch (E1 e){
...
} catch (E2 e){
...
} finally {
    // This will always be executed, no matter...!*
}

// Also possible in Java 7 (nice, more compact)
} catch (E1 | E2 e){
```

Exception Swallowing

By far the worst way to handle exceptions, strictly forbidden!

```
// BAD; BAD; BAD
try{
    // Possible exception;
} catch (E e){
}                   // Empty, nothing here, but exception
                    // handled program will continue...
                    // So exception unnoticed for now...
```

In Practice How To?

No common agreed upon best practices*

- Of course if possible to handle the exception do so

Possible

- Skip checked exception. Catch checked exceptions, wrap in RuntimeException and re-throw **(exception tunneling)***
- Catch and send exception to central ExceptionHandler (handler can act as observable to propagate exceptions to GUI)

In Practice How To? cont

Exception translation*

- Catch and rethrow at appropriate abstraction level (meaningful for the level). Especially for end users who doesn't understand strange technical messages

Summary

- Design goals
- Canonical form is to be considered
- Quite a few principles (some may overlap and sadly, contradict)
- Some design patterns: Facade, Proxy, ...
- Keep model clean, use services
- The MVC model is theoretically simple but implementation has many forms
- Exceptions: No best practice