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

Design goals

In this course

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

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 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

Presented so far:

- Interface segregation principle
- Dependency inversion 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 you possible violate SRP
- 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 upcoming…)
- Easier to understand (smaller more focused)
- Easier to test
- Easier to combine with other (easier to replace, reuse)

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)
- Related to to info hiding
- Similar to SRP, Separations of concerns

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

Example: 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.

Open Closed Principle

*"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" //*R.C. Martin

I.e. we should not modify tested (proved) code to extend functionality, instead we should add new code

Often implemented by subclassing

- Example Lab 1c: EventTranslator
- Also see Inheritance slides

Design Levels

Design is a multilevel activity

- Literals
- Attributes
- Methods
- Interface/Classes
- Class aggregates or similar
- Modules
- Application
- System ... (a small one in this course)
- Big Systems ... (not in this course)

Literals

Normally no literals in code

Replace with constant (public static final…) or enum

```
// Bad usage of literal
if( v == 45 ){  // What is 45? Why??

}

// Should have used
public static final int MAXIMUM_ANGLE_IN_DEG = 45;
```

Attributes

As few as possible, all private, preferably final (again: reduce state)

One attribute for each fact (DRY, no shared)

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

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

Methods

Have seen

- Reasoning using invariants
- Mutators vs. Accessors
- If override put @Override
- … more issues …

Methods, cont

- Limit number of parameters (prefer objects to primitive types). Have impact on testing, combinatorial explosion of test cases
- Closureness, return same type as parameters (ease functional style, if immutable).
  - Example: public Matrix add(Matrix a, Matrix b)
- Returning "this", makes chained invocation possible
  - o.doIt().doOther().doFinal()
  - Aka **fluent programming** (how to debug..?);
- Booleans return values (as signals, … prefer exceptions, if it's an exception, more to come…)
- All overloaded methods should be in same class

Nullable Types

Java allows any reference variable to be set to null

- null-value, represented by the literal **null**, is the only value in the Null-type
- null is a value representing "not a valid object" (but it's not an object)
- *"The direct supertypes of the null type are all reference types other than the null type itself"* // JLS 4.10.2
- null == null always true

```
// In practice we have this ...
StringOrNull s = ...;
IntegerOrNull i = ...;
```

Who Invented Null

Invented by C.A.R. Hoare (computer science gigant)

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

NullPointerException (NPE)

RuntimeException, easy to locate, but sometimes hard to find cause

```
// Must check before ... but what if we forget?
if( map.containsKey("svea")){
    Person p = map.get("svea");

// Or check after ... but what if we forget?
Person p =  map.get("svea"))
if( p != null ){
```

Handling null's

Have a type (null) not "handled" by the type system

What to do?

- Object (class)-internal null's accepted
- Incoming null's (method-parameters)
- Outgoing null's (method-results)

Handling Incoming null's

Hard, not much to do ...

- Checks doesn't help much
  - Accept NullPointerException
  - Throw IllegalArgumentException (with a possible better error message)
- Always check for null if value should be stored in some collection or sent along to other object
  - Throw IllegalArgumentException (more to come how to...)

Handling Outgoing null's

Never (or at least very rarely) return nulls

If result is a Collection

- Return empty Collection (String return "")

If result is a single value. Hard..

- ... quick look at Haskell way... (next slide)

Haskell Maybe Monad

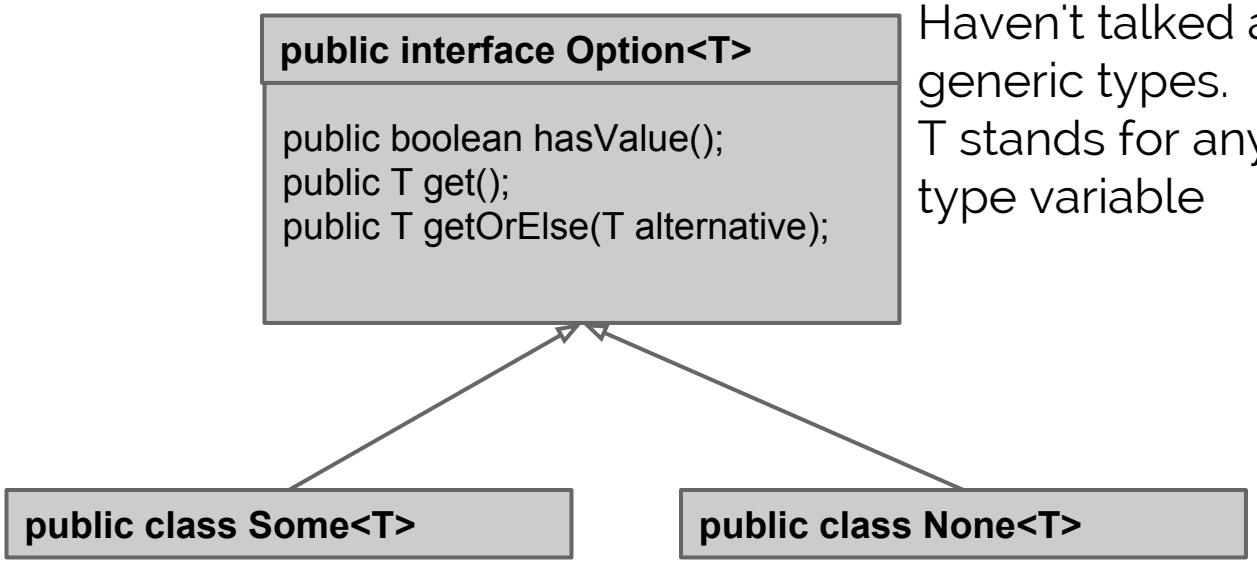
The Maybe monad represents computations which might "go wrong", in the sense of not returning a value

```
// Definition of Maybe
data Maybe a = Just a | Nothing
     deriving (Eq, Ord)

// Usage, Bob possible not in phonebook, what to do?
> lookup "Bob" phonebook
Just "01788 665242"
> lookup "Blblblb" phonebook
Nothing
```

Option<T>*

Possible to do something similar to Maybe in Java (also see Java 8)



Haven't talked about generic types.
T stands for any type, a type variable

Usage Option<T>

Can't use returned reference directly, check enforced by type system

```
// Impossible to forget to check
Option<SomeClass> o = m.getIt();
SomeClass value = o.getOrElse(...); //Possible default
value
```

Null: Arrays and Collections

"An array created using new Object[10] has 10 null pointers. That's 10 more than we want, so use collections instead, or explicitly fill the array at initialisation"

Still can get NPE from collections

```
// Hmm..
List<Integer> is = new ArrayList(4); // 4 not actual size
int i = is.get(2);
```

Interfaces

Have seen

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

Designing interfaces sometimes very hard, hard to get the “correct abstraction”

Classes and Aggregates

Design issues. Have seen

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

Other issues...

Canonical Form

Issues common to all objects of all classes

- Hashcode (fairly unique numerical id for object)
- Equality
- Copying
- String representation

All have default implementations in java.lang.Object, we often override

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)
- NetBeans can generate (Insert code…)!

Equality

- Default implementation in java.lang.Object.equals() uses == (so by reference)
- "By value" more useful
- If so; have to override equals()-method (tricky when inheritance, more to come...)
- Two objects which equals() says are equal must report the same hash value (else equal objects possibly end up in different locations in collections and thus not found
- More on general contract, see Inheritance slides

General Contract for equals()*

The equals method implements an equivalence relation:

- It is **reflexive**: for any reference value x, x.equals(x) should return true.
- It is **symmetric**: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is **transitive**: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is **consistent**: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
- For any non-null reference value x, x.equals(null) should return false.

If we have inheritance hard to fulfill all this, see Inheritance slides

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 (no shared references, **a deep copy**) class must implement own clone(), override and change access to public
- Access in Object: protected (can't call directly)
- Tricky and strange to implement, must handle unnecessary exception, implement marker interface and use a specific "idiom", problems with final. No constructor may be involved !?!... see later Inheritance slides...

General Intent for clone()*

The general intent is that, for any object x, the expression:

`x.clone() != x` will be true, and that the expression:

`x.clone().getClass() == x.getClass()` will be true, but these are not absolute requirements.

While it is typically the case that:

`x.clone().equals(x)` will be true, this is not an absolute requirement.

String Representation

*"In general, the **toString method** returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.toString()" //* Javadoc

Very useful during development

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 interface 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);
```

General Contract Comparable

- **anticommutation** : x.compareTo(y) is the opposite sign of y.compareTo (x)
- **exception symmetry** : x.compareTo(y) throws exactly the same exceptions as y.compareTo(x)
- **transitivity** : if x.compareTo(y)>0 and y.compareTo(z)>0, then x. compareTo(z)>0 (and same for less than)
- if x.compareTo(y)==0, then x.compareTo(z) has the same sign as y. compareTo(z)

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. Contract more complicated.

```
// Only method in interface Comparator.
public int compare(T t1, T t2) {
...
}

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

Modules

In a stateful module 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, … prefer stateless modules

Compare: Trie/Dictionary-module vs Translator

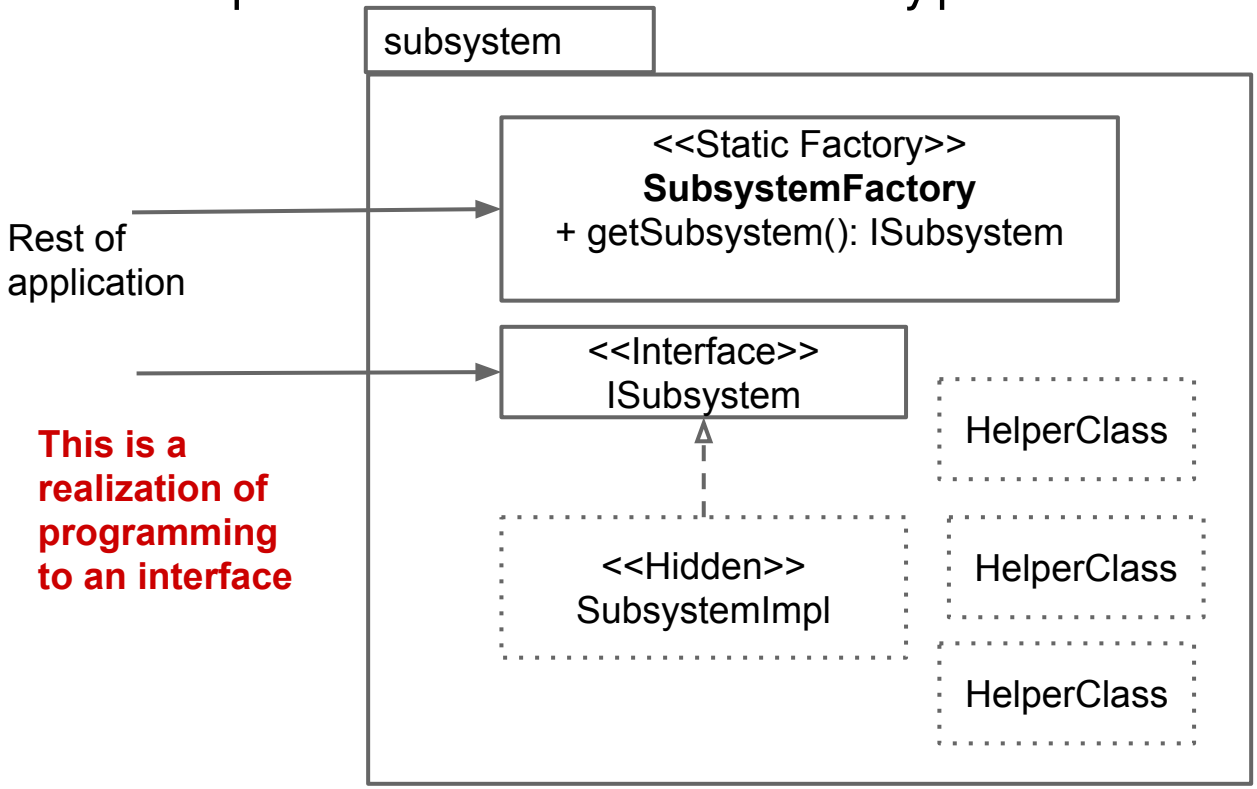
Implementing a Service Module

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

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

Facade Design Pattern*

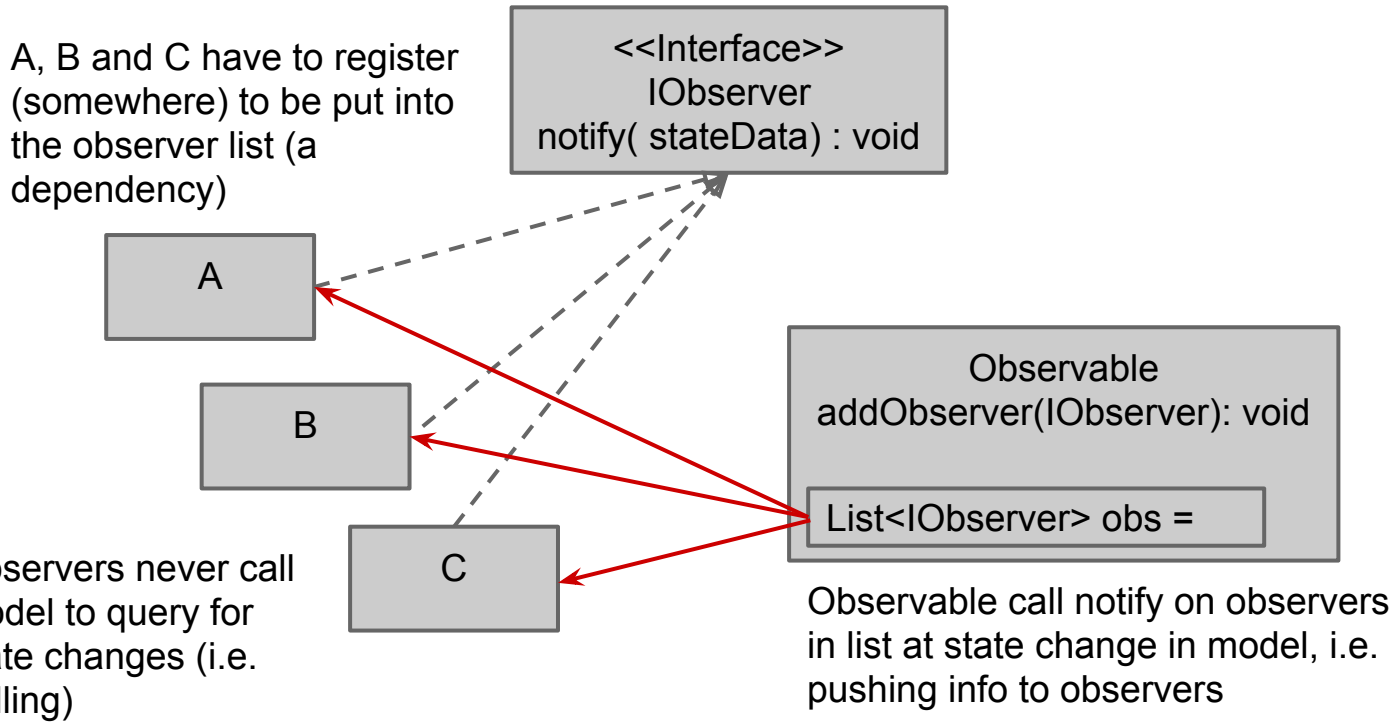
Only visible types are the factory and the interface (possible parameter and return types)



Observer Pattern

Decoupling observers from the observable

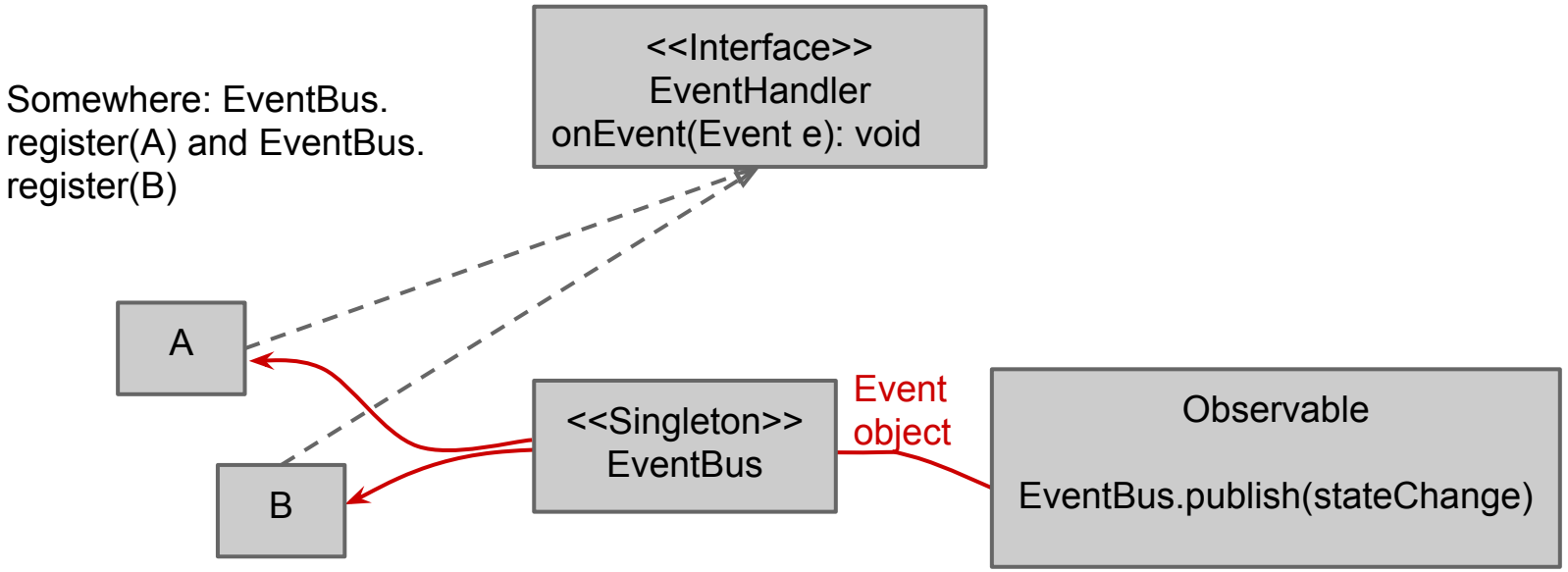
- A **push** design (vs **pull** design)



Observer: A Variation*

Use messaging (implemented as an EventBus)

- Will remove any direct dependency Observer <-> Observable (only dependent on EventBus)



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...

Adding Messaging to Model*

Using OPC, i.e. subclass the model class

- All model logic in model class (superclass)
- Add messaging in subclass
- Subclass call super to do “the work” then signals using messaging

NOTE: Code sample MVC, class EventModel

Messaging Frameworks

There are existing "frameworks" to support messaging

- Google Guava (also uses an EventBus)
- Java context and dependency injection (aka CDI, aka Weld)
- Will also reduce dependencies

If building real application should inspect these or similar

Applications with Graphical User Interfaces

Designing GUI applications is 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?
- How to (where) construct and connect the GUI (normally many very many objects …).
- …

Model-View-Control*

Common solution for GUI applications

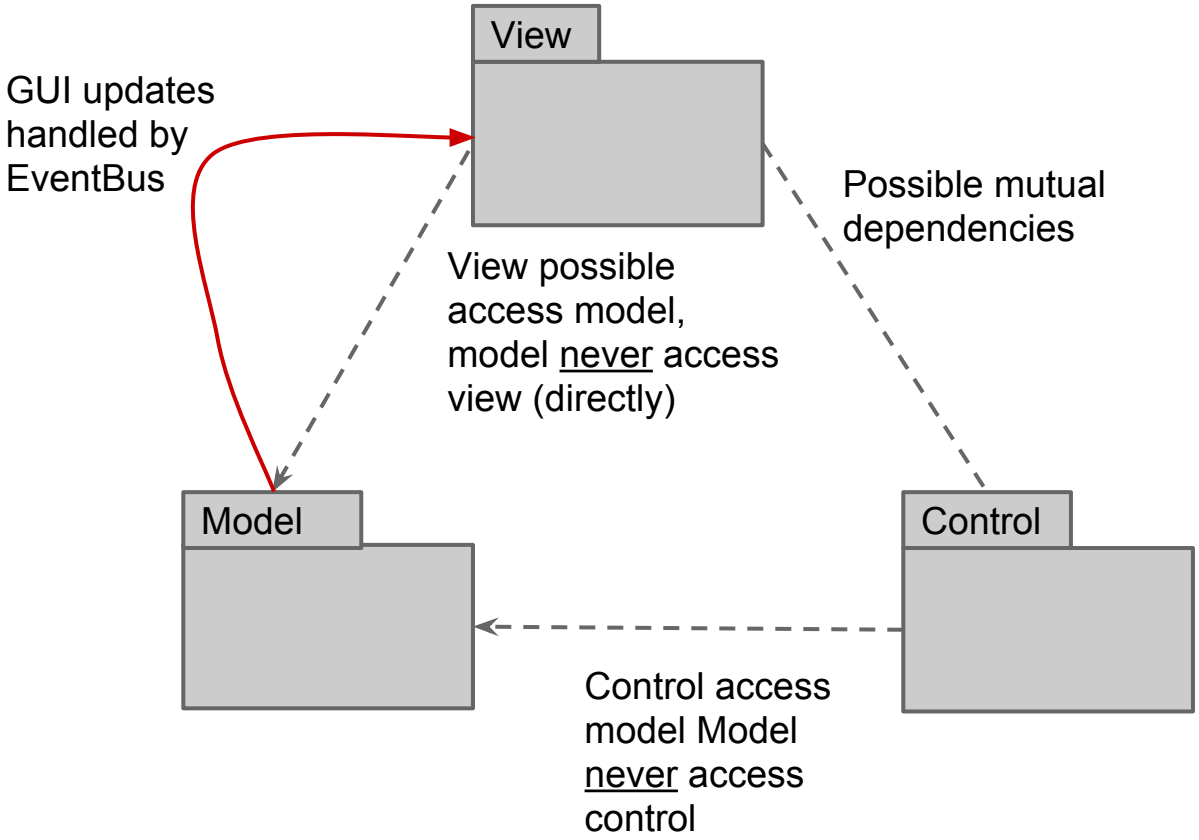
- MVC-design aka MVC-architecture

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, Linux, app, smartphone, web)
- **Model**, the OO-model. If done correctly this should be stable.
- **Control**, parts that coordinate the interaction between GUI and Model. A thin layer between GUI and model

MVC Design

Basic parts and dependencies



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).
- **Divided:** Some parts in control others in model. Have to use your skills...(principles, coupling, cohesion, abstraction, ,...).

Division of Labor: Example

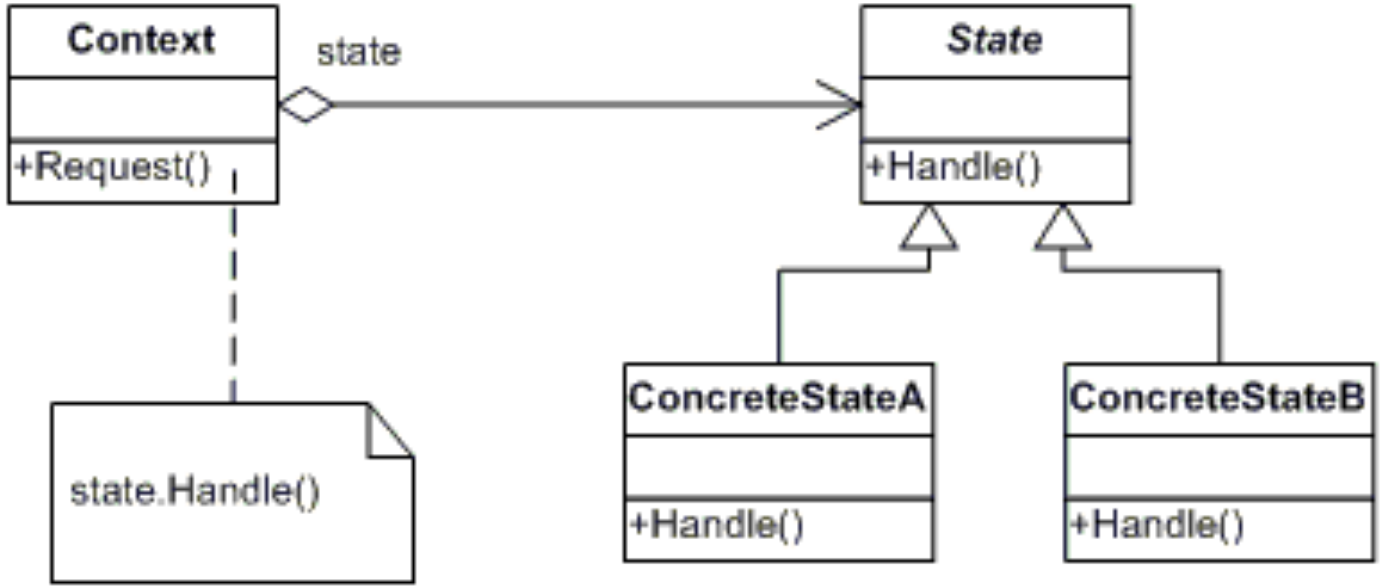
WordFeud, fat or anemic?

Do it in Control? …

or much in Model?

State Pattern*

Usage: Remove the double switch idiom (switch state/ switch action). Ease introduction of more states



Exceptions

Large, difficult not very well understood topic

Exceptions used for "exceptional" events, 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

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 Jim");
```

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

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**)

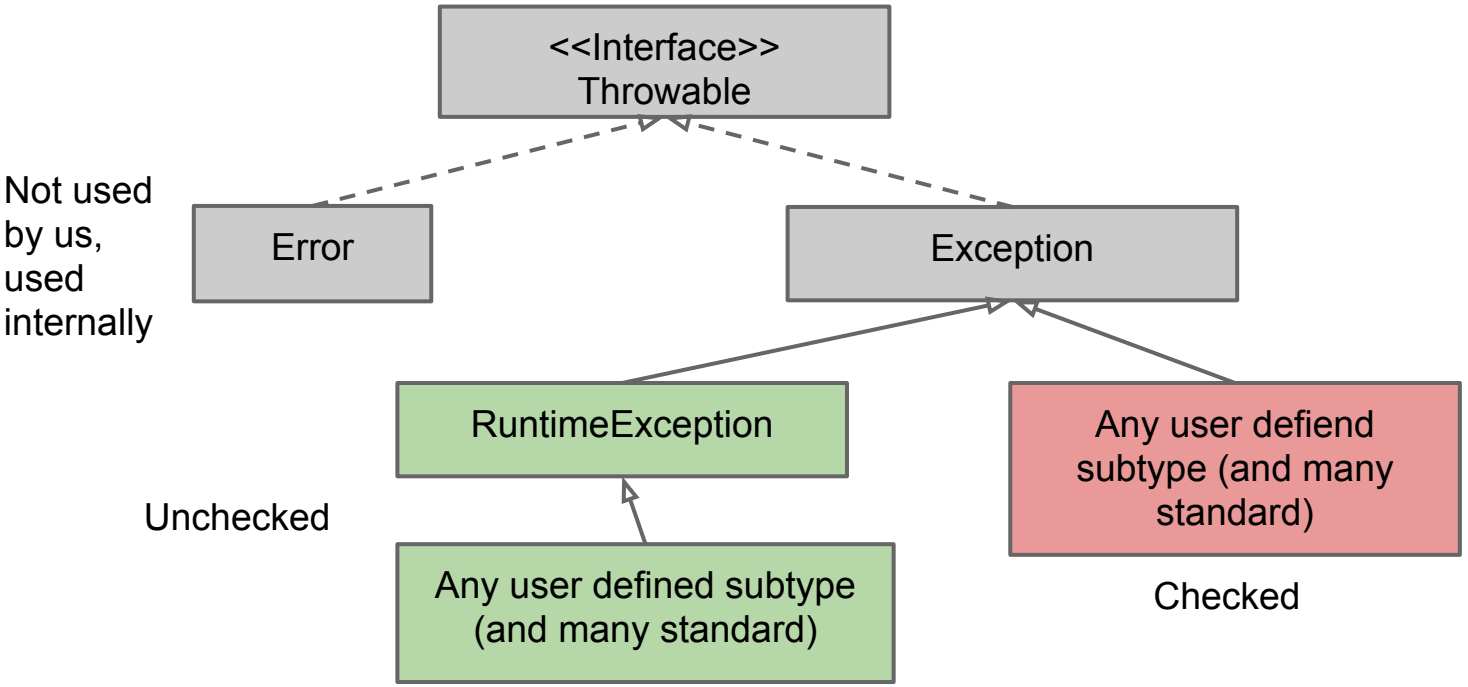
Basic Exception Handling

```
// A methods that throws (using a throws clause)
public void openFile() throws IOException {
… // File not found will generate IOException
… // Not handled here passed to caller using throws
}


// Somewhere in call chain, handle exception
try{
    // Possible far away o.openFile() may throw;
    // No statements after executed if an exception!
} catch (IoException e){  // A handler, exception caught
    // Try to recover...
}
```

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)

General Form of try*

```
// General form
try( someAutoClosableResources ) {
    // Possible exception here ...
} 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){
```

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

Exception Swallowing

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

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

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!

Best Practices*

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)

Best Practises, 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

Interfaces

- Avoid checked exception (don't know if implementations will throw, don't force implementor)

Summary

- Quite a few principles (some may overlap and sadly, contradict)
- Handling nulls
- Canonical form is to be considered
- Facade design pattern
- Keep model clean, no services in model
- MVC model for GUI applications is theoretically simple but implementation has many forms
- Exceptions: No best practice