

Dependencies

Slide Series 4

Content

Modifiability, extensibility and testability

Information hiding

Design decisions

Single responsibility and Dependency inversion principles

Associations

Separating construction and usage

Law of demeter

Modules

Abstract classes, inner local classes

Iterator

Modifiability, Extensibility and Testability

Must be possible to modify program

- We will not get it right on the first try, must be able to modify

Programs must be able to extend

- Many (most) program will expand, adding new features. Must be able to grow

Must be possible to test program

- Testing is our main method to ensure quality

Impact of Dependencies

If too much dependencies impossible to modify, extend or test!

Any modification will ripple through the whole application!

No isolated parts to test, everything is dependent (a big ball of mud, spaghetti code)

Causes of Dependencies

Different parts of application knows (and uses) too much from other parts

Part of application have too many responsibilities

Information Hiding

David Parnas first introduced the concept of information hiding around 1972. He argued that the primary criteria for system modularization should concern the hiding of critical design decisions.

- Original paper on course page > Links

Hiding information isolates clients from requiring intimate knowledge of the design to use other parts, and from the effects of changing

The fundamental principle

Some Critical Design Decisions

Many design decisions emanate from the problem (the object model), the problem has (hidden) dependencies. Which knowledge should objects have, more to come ...

Technical things to hide

- Data representations
- Algorithms - e.g, sorting or searching techniques
- Input and Output Formats
- Ordering of low-level operations, process sequences
- Platform/machine dependencies, e.g., byte-ordering, character codes
- ...

Abstraction

"Abstraction is a process by which concepts are derived from the usage and classification of literal ("real" or "concrete") concepts, first principles, or other methods"

"Abstractions may be formed by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose"

//Wikipedia

Let's try some Abstraction

The group to the left has something in common that the group to the right doesn't have

- What is it???

Here we [go...](#) (Bongard problems, select any)!

Encapsulation

Technique to hide the data representation i.e. bundling hidden data and operations on data

- Create a class with appropriate methods to work on the hidden data representation
- Of course by default we always use private for instance variables (and prefer final)

Info hiding violated if class has set/get methods for all attributes

Object Composition

*"In computer science, **object composition** [composition]... is a way to combine simple objects or data types into more complex ones."*

"In programming languages, composite objects are usually expressed by means of references from one object to another; depending on the language, such references may be known as fields, members, properties or attributes,..." // Wikipedia

Class Dependencies

In Class diagram lines or arrows between classes indicates dependencies (UML Spec. says semantic relation)

- Kinds: **Association**, Aggregation, Composition (we only use the first ...)

UML Notation

- Associations between objects: Solid line or arrow
- Associations has multiplicity, how many objects involved: 1,2,..n or *, ranges 0..1 or 1..n on line or arrow
- Usage of types as parameters and return values: Arrow with dashed line (a dependency)

Code and UML

```
public class A{  
    private B b;  
}
```

```
public class B{  
    private A a;  
}
```

```
public class A{  
    private List<B> bs;  
}
```

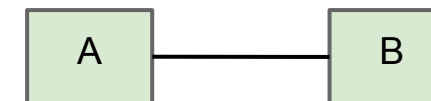
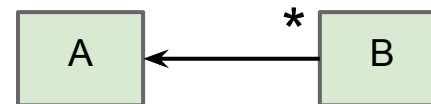
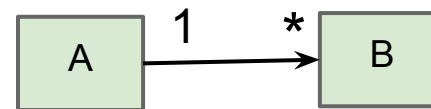
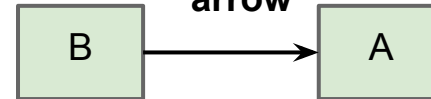
```
public class B{  
    private A a;  
}
```

```
public class A{  
    private B b;  
}
```

```
public class B{  
    private A a;  
}
```



**UML
association
arrow**



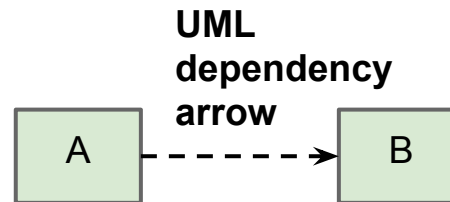
**Mutual
association!**

Code and UML, cont.

Dependencies on parameter or return type, weaker dependency (so "weaker" line). Normally more interesting between packages.

```
public class A{  
    public B dolt( ){  
        ...  
    }  
}
```

```
public class B{  
    public void dolt(A a ){  
        ...  
    }  
}
```



Reducing Class Dependencies

Have to control dependencies

Ideal we would like the dependencies to form a tree or a graph with no circuits

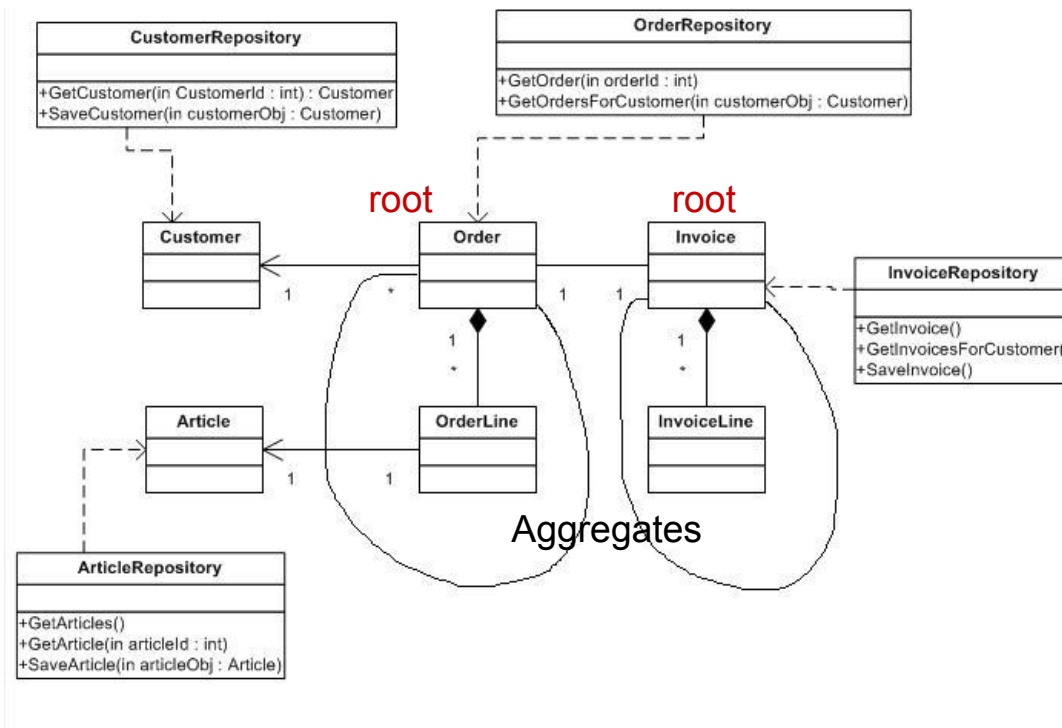
- No circular dependencies

Basic technique: Reduce/remove change associations

- Again: Change attributes to local variables, will reduce dependencies and state!

Class Aggregates

Clusters of associated objects treated as a unit



Order and Invoice are aggregate roots

Objects in aggregate accessed only via root (no direct association to others)

Root has global unique identity

Others have identity inside aggregate only

Resolving Mutual Associations

Mutual associations are bad

- Tight coupling
- Domino effects (change one, affect other)
- Classes not understood in separation

Resolve by

- Ignore one Direction
- Lookup one direction
- Association class

Mutual: Ignoring 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 {  
    ...  
}
```

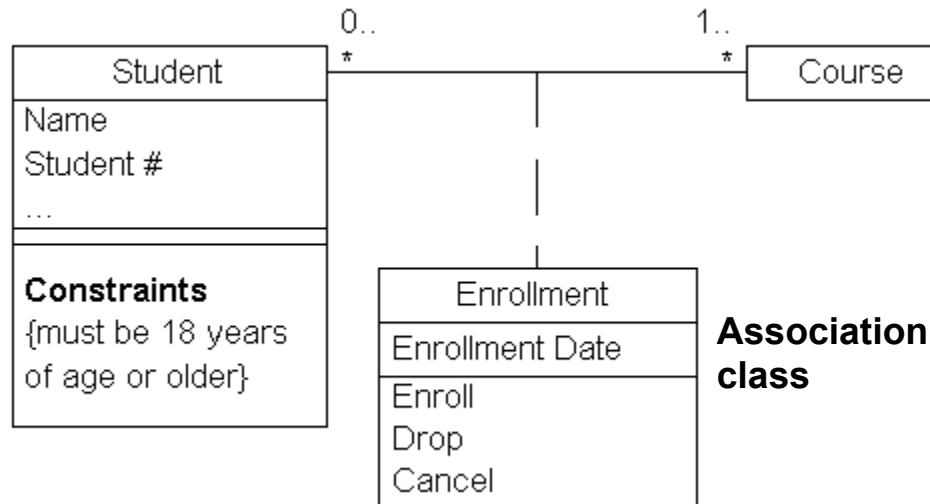
```
public class Customer {  
    ...  
    private List<Order> orders;  
    ...  
}
```

```
// Lookup class  
public class OrderBook {  
    Map<Order, Customer> orderCustomer = new HashMap<>();  
  
    public Customer getCustomerFor( Order o ){  
        return orderCustomer.get(o);  
    }  
}
```

Mutual: Many to Many*

Mutual dependencies are bad, mutual many to many worse

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



Using new

If using **new** all over we have hardwired dependencies on implementation (can't change to subtype)

```
// Using new , direct dependency on concrete class
// (implementation). Also: Hard to test
public class A {
    B b = new B(); // Fixed can't change (hard to test,
                  // can't pass in dependency)
}
```

Factory Method*

Factory method (always static) used to remove usage of new

```
public class MyClass {  
    // The Factory method  
    public static MyClass newInstance(/*possible args */){  
        // All new here (in class)  
        return new MyClass(/* possible args */);  
    }  
}
```

Establishing Dependencies

Many (most?) dependencies established during construction of application

- Objects should not (randomly) create other needed objects
- Careful construction process will clarify and reduce dependencies

Separate construction and use!

Static Factory*

A static factory is a pure static class used to construct objects or object graphs

Typical methods

- newInstance(), create new instance
- getInstance(), create new or return existing (singleton och cached)

Programming to an Interface

A design principle

- To avoid hard coded dependencies on implementations

"Programming to the interface reduces dependency on implementation specifics and makes code more reusable. It gives the programmer the ability to later change the behavior of the system by simply swapping the object used with another implementing the same interface." //Wikipedia

Interfaces

An interface is a contract/specification and an abstraction

- Should define behaviour from the point of the user (other software entities)
- No exposure of implementation details
- No methods for construction

Law of Demeter

```
// Hmm, this is bad! Method knows how to navigate
public void createScratchFile(){
String outPutDir = ctxt.getOptions().getScratchDir().
                                getAbsolutePath();

... (create file put in outPutDir)
}
```

```
// Better? No, will cause explosion of methods...!
String outPutDir = ctxt.
getAbsolutePathOfScratchDirectory();
```

```
// Reasonable (supply some data, avoid too big method)
OutputStream out = ctxt.createScratchFileStream
(outPutDir);
```

Law of Demeter, cont

LoD states: Method *m* of class *C* should only call methods of

- *C* (itself)
- Attribute of *C*
- Argument of *m*
- Object created by *m*

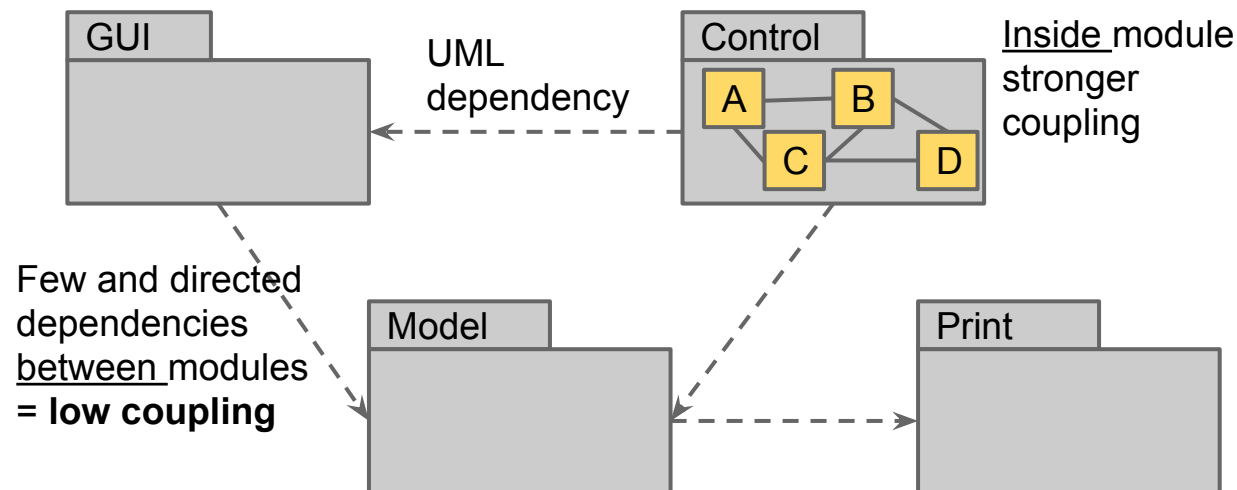
Modules

No generally accepted definition but we say

- Application composed of modules (parts)
- Modules composed of classes , interfaces, submodules, ...
- Modules in Java created by use of **packages**
- Modules often referred to as **subsystem** or **service**
- We use [UML package diagrams](#)

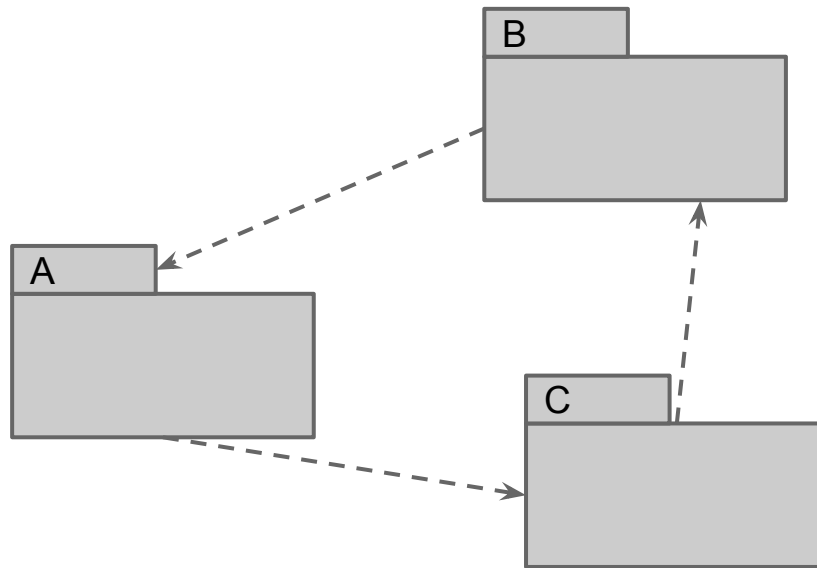
Coupling and Cohesion

Want an application composed of loosely coupled modules (packages) with high cohesion (samhörighet)



Circular Module Dependencies

This is bad!



In practise it's one single (large) module

Kinds of Cohesion

Coincidental cohesion (worst)

- Only relationship between the parts is that they have been grouped together (example: utilities)

Functional cohesion (best)

- Parts of a module are grouped because they all contribute to a single well-defined task of the module (example: printing)

Kinds of Coupling

Content coupling (pathological)

- One module modifies or relies on the internal workings of another module

Common coupling (high)

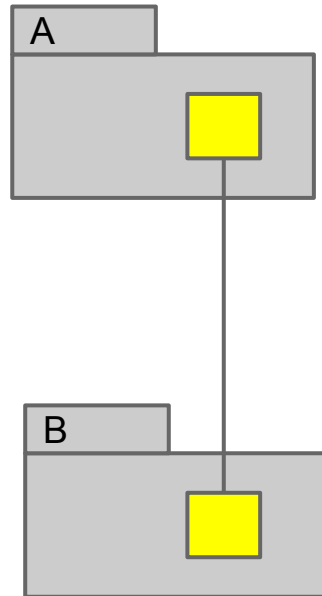
- Two modules share the same global data (a global variable (a Singleton), global coupling)

Message coupling (lowest)

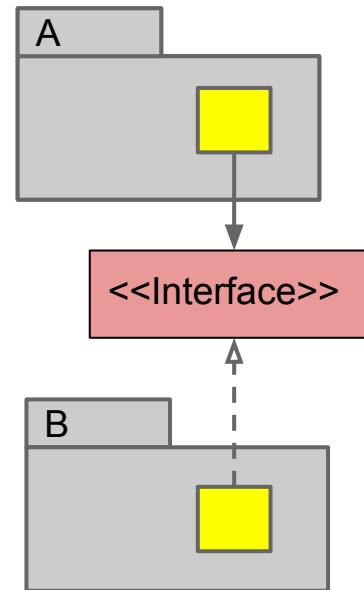
- Achieved by state decentralization (as in objects) and component communication with parameters/return values (or messages)

Resolving Pathological Coupling

Pathological, mutual dependencies on internal classes



Resolved using interface and removing one direction



Interface Segregation Principle*

"No client should be forced to depend on methods it does not use" // R.C. Martin

If interface too large, split into smaller

- Interfaces can extend!
- Create "role" interfaces

An Interesting Case

*"When designing the Collections API Joshua Bloch decided that instead of having very fine-grained interfaces to distinguish between different variants of collections (eg: readable, writable, random-access, etc.) he'd only have very coarse set of interfaces, primarily **Collection**, **List**, **Set** and **Map**, and then document certain operations as "optional". This was to avoid the combinatorial explosion that would result from fine-grained interfaces."*

//From the [Java Collections API Design FAQ](#):

Abstraction Levels

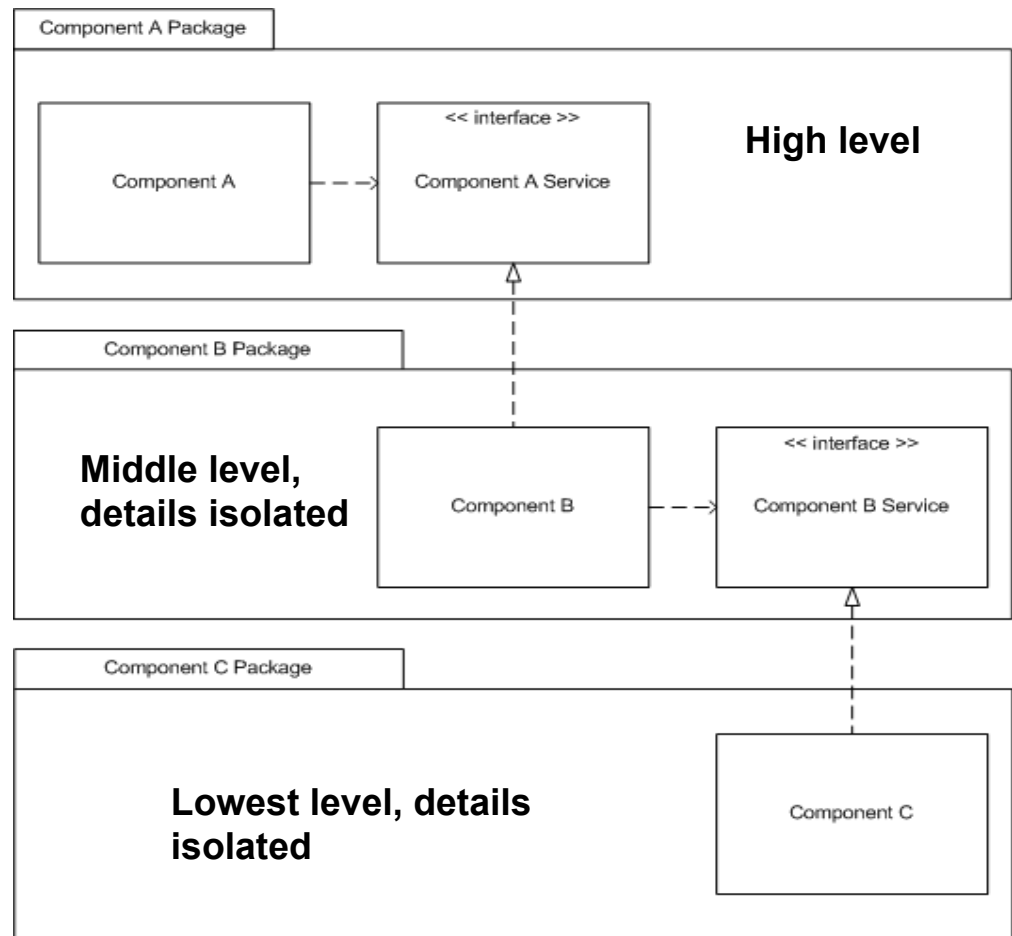
"The level of complexity by which a system is viewed. The higher the level, the less detail. The lower the level, the more detail. The highest level of abstraction is the single system itself. The next level would be only a handful of components, and so on, while the lowest level could be millions of objects." //Free Dictionary

- Low(est) level in code often general, possible to reuse, uses primitive types
- Higher levels have application specific knowledge uses objects
- Levels typically drawn bottom (low level) to top (high level)

Dependency Inversion Principle*

There should be interfaces between different abstraction levels (higher levels shouldn't be dependent on lower level details that possible will change)

NOTE: Used in lab's



Inspecting Dependencies

Many tools to inspect module dependencies

- [STAN](#) (no plugin for NetBeans, have to run standalone)

Don't Repeat Yourself (DRY)

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." //Wikipedia

One possible violation is duplicate code...

Duplicate Code

Duplicate code (literally or logical) means dependencies

- If modifying must keep in sync

We never ever allow duplicate code anywhere

- Each fact should be stated in exactly one place

Possible use **abstract class** to eliminate duplicate code

Abstract Class

"An abstract class is a class that is incomplete, or to be considered incomplete "

"Normal classes may have abstract methods (§8.4.3.1, §9.4), that is, methods that are declared but not yet implemented, only if they are abstract classes" // JLS 8.1.1.1.

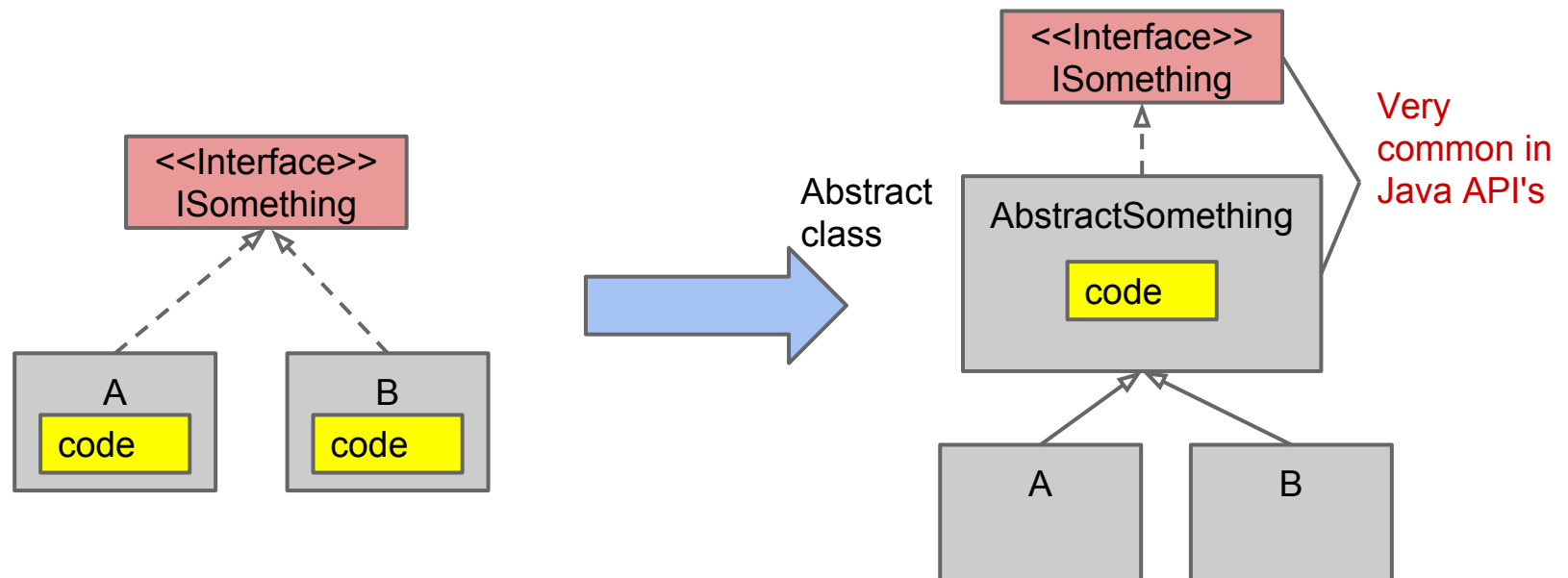
Abstract Method

A class C has abstract methods if any of the following is true (JLS 8.1.1.1.):

- *C explicitly contains a declaration of an abstract method (§8.4.3). [Using method modifier **abstract**, can't have method body]*
- *Any of C's superclasses has an abstract method and C neither declares nor inherits a method that implements (§8.4.8.1) it.*
- *A direct superinterface (§8.1.5) of C declares or inherits a method (which is therefore necessarily abstract) and C neither declares nor inherits a method that implements it.*

Usage Abstract Classes*

Used to eliminate duplicate code and/or implement default behaviour



Other Usage Abstract Classes

Impossible to change interface if "published" i.e. many others use it

- Example: Assume all users of our module must use parameters of type "ISomething" vs. AbstractSomething

```
// Assume used by very many others
public interface ISomething {
    void doIt();
    int doOther();
    void doYetOther;
}
```

Can't change this will break clients (they all have to implement new methods or even worse remove method ...)

```
// Assume used by very many others
public abstract class AbstractSomething {
    void doIt();
    int doOther();
    void doYetOther;
    // Next version adds this method
    void someNewMethod(){
        // Default implementation
    }
}
```

If using this possible to add methods with default implementation

Local Inner Classes

Have seen

- Inner class (have reference to enclosing)
- Inner static (nested top level class, no reference)

There's also

- **Local inner** classes, class defined inside method. Can use parameters and local variables even after method has terminated (variables must be final - declared, so read only)
- **Anonymous inner** classes, as local but anonymous (nameless)

The Iterator Design Pattern

Makes it possible to traverse a collection without knowing the representation (how the collection is constructed)

Collection could be a

- List
- Set
- Tree
- ...

Iterator and Iterable*

```
// Java interface to traverse some collection  
// E is a type variable, more to come...
```

```
public interface Iterator<E>  
    public boolean hasNext();  
    public E next();  
    public void remove();
```

```
// Java interface for a class capable of returning an  
// iterator
```

```
public interface Iterable<T>  
    public Iterator<T> iterator();
```


Implementing an Iterator

Two principally different ways (both using inner local classes)

Fail fast iterator, modification of collection not allowed during traversal, will throw `ConcurrentModificationException`

- Remember size of collection when creates, check size before any operation
- Most Java standard iterators are fail fast

Fails safe iterator, will never throw, creates a copy of collection, then traverses the copy

Traversing Collections*

Almost all collection classes implements Iterable

```
// If iterable can use the short for-loop
for( A a : o.getCollectionOfAs() {    // Prefer!
}

// If removing element must use Iterator explicitly
// (Iterator.remove()), else
// ConcurrentModificationException
```

- Exception: Map (which isn't a Collection, doesn't implement Collection)
- This will probably be done in a more functional style in Java 8

ListIterator

Another Java iterator

```
// ListIterator can move forward and backward
public interface ListIterator<E> extends Iterator<E> {

    // All next methods are here and the following...
    public boolean hasPrevious();
    public E previous();
    public void set(E e);

}
```

Dependencies and RTTI

"The problem with type checking is you are making assumptions about the type hierarchy which might not be true after further development of the software. For instance, if you conditionally perform some action based on whether or not the type of an object *O* is *T*, you're not accounting for the possibility of that same action being necessary for another type of object. If you use polymorphism properly you can avoid this problem."

"Putting it another way: if you perform different actions depending on the type of an object, you are assuming that the object can only be one of those certain types. It's easy for this assumption to be later rendered invalid. If you have lots of type-checking code, and you add a new type, you need to revise all that code; if you make use of polymorphism, you only need to edit the code in the one place - in the definition for the new type". `//davmac`

Example : Bad Usage of RTTI

```
// Bad, what if we would like to add another animal
```

```
if (animal instanceof Cat) {  
    Cat cat = (Cat) animal;  
    cat.meow();  
} else if (animal instanceof Dog) {  
    Dog dog = (Dog) animal;  
    dog.woof();  
} else if (animal instanceof Hippopotamus) {  
    Hippopotamus hippopotamus = (Hippopotamus) animal;  
    hippopotamus.roar();  
}
```

```
// Solution, use polymorphism
```

```
animal.say()
```

This is sometimes called "**the switch statement**" smell, if having check here possible also will have at other locations, ... if change here ...!

[Code smell](#)

Refactoring

How to fight dependencies? Have seen some examples often referred to as **refactoring**!

Refactoring = restructure code without changing the functionality

- Introduce methods
- Move code to abstract base classes
- Split/Collapse classes
- Move classes between packages
- Split/Collapse packages
- Introduce interfaces

Summary

- Some more principles: Single responsibility, Dependency inversion and Law of Demeter
- Minimizing dependencies (associations classes and between modules and more)
- Separating construction and usage
- Abstract classes, inner local classes
- Design pattern: Iterator