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. He stressed hiding "difficult design decisions or design decisions which are likely to change."

"Hiding information in that manner isolates clients from requiring intimate knowledge of the design to us a module, and from the effects of changing"

The fundamental principle

Some Critical Design Decisions

Typical things to hide

- Data representations
- Algorithms - e.g, sorting or searching techniques
- Input and Output Formats
- Ordering of low-level operations, process sequences
- Separating policy and mechanism
  - Separate interfaces from implementations
- Selection of third party software
- Platform/machine dependencies, e.g., byte-ordering, character codes
- ...

Many design decisions emanate from the problem, the problem has (hidden) dependencies.

Encapsulation

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

Of course by default we always use private for attributes

Basic Class Dependencies

Associations between types

Usage of other types as parameters and return values

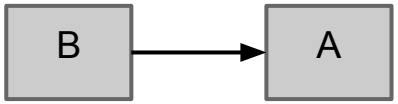
Associations

Dependencies via attributes

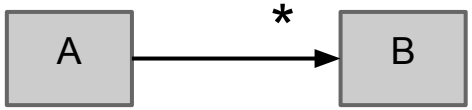
```
public class A{
    private B b;
}
```

UML association

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

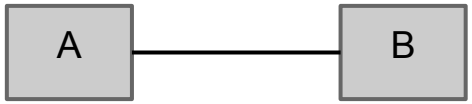



```
public class A{
    private List<B> as;
}
```




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

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



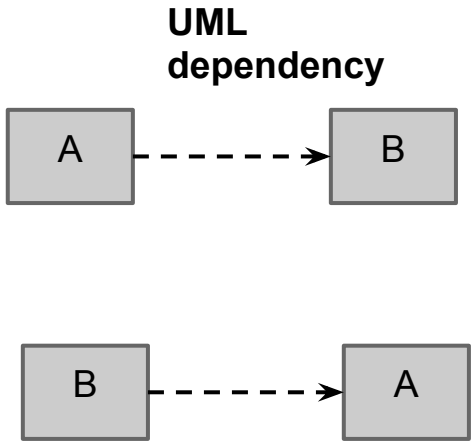
Mutual association!

Associations, cont

Dependencies on parameter or return type, weaker dependency (so "weaker" line)

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

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



Reducing Class Dependencies

Have to control associations (attributes)

Ideal we would like the dependencies to form a tree

- No circular associations

Basic technique: Reduce/remove associations

- Again: Change attributes to local variables reduces dependencies and state!

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
- Use of interfaces (normally between packages/modules)

Associations: 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;
    ...
}
```

Associations: Lookup One Direction*

Lookup Customer given Order (Orders unique)

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

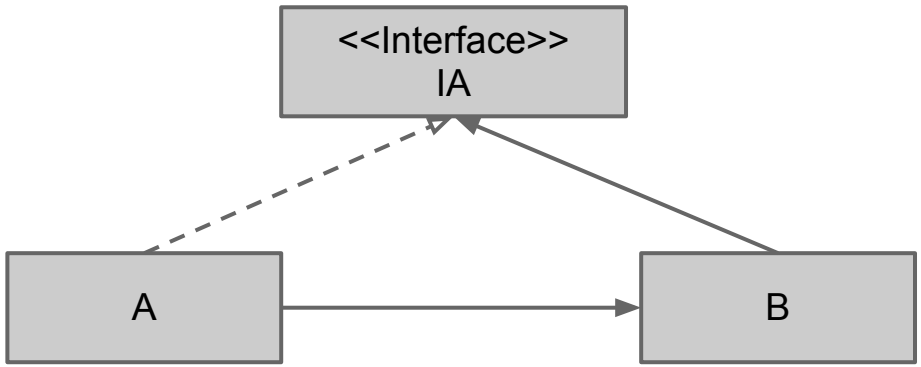
```
public class Customer {
    ...
    private List<Order> orders;
    ...
}
```

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

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

}
```

Associations: Use Interfaces

Normally not between classes in same package

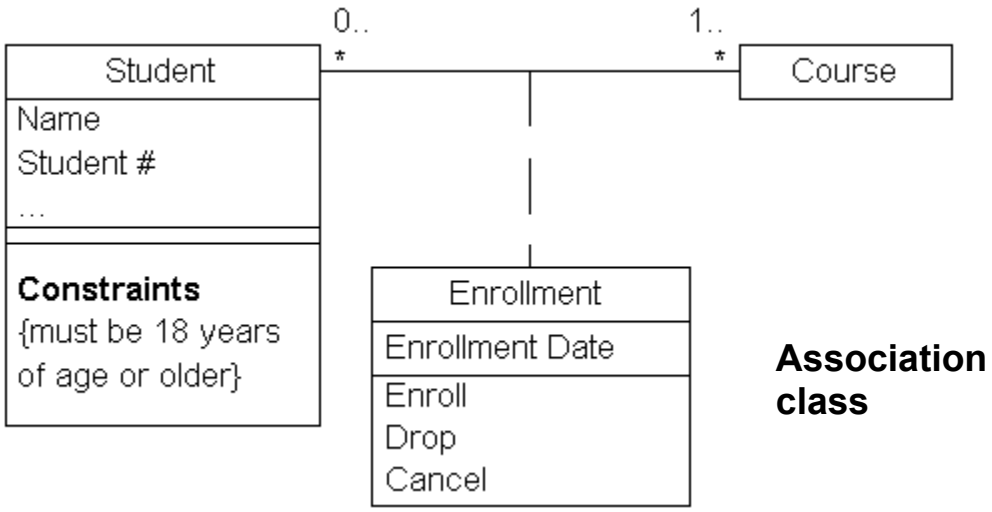
Use between modules, more to come...



Resolving Mutual Many to Many*

Mutual dependencies are bad, mutual many to many worse

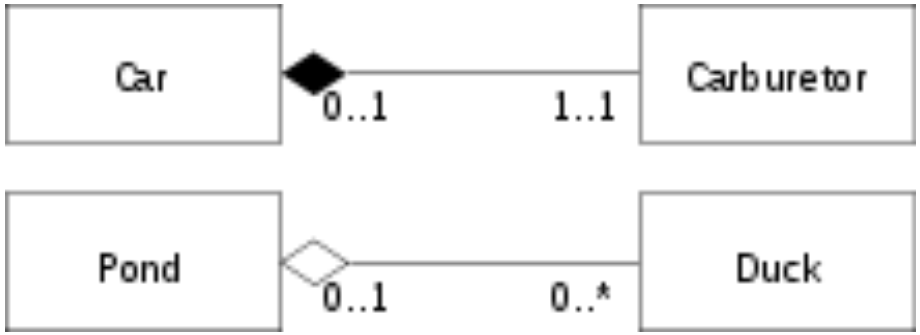
- Resolved using "extra" association class



Forms of Associations

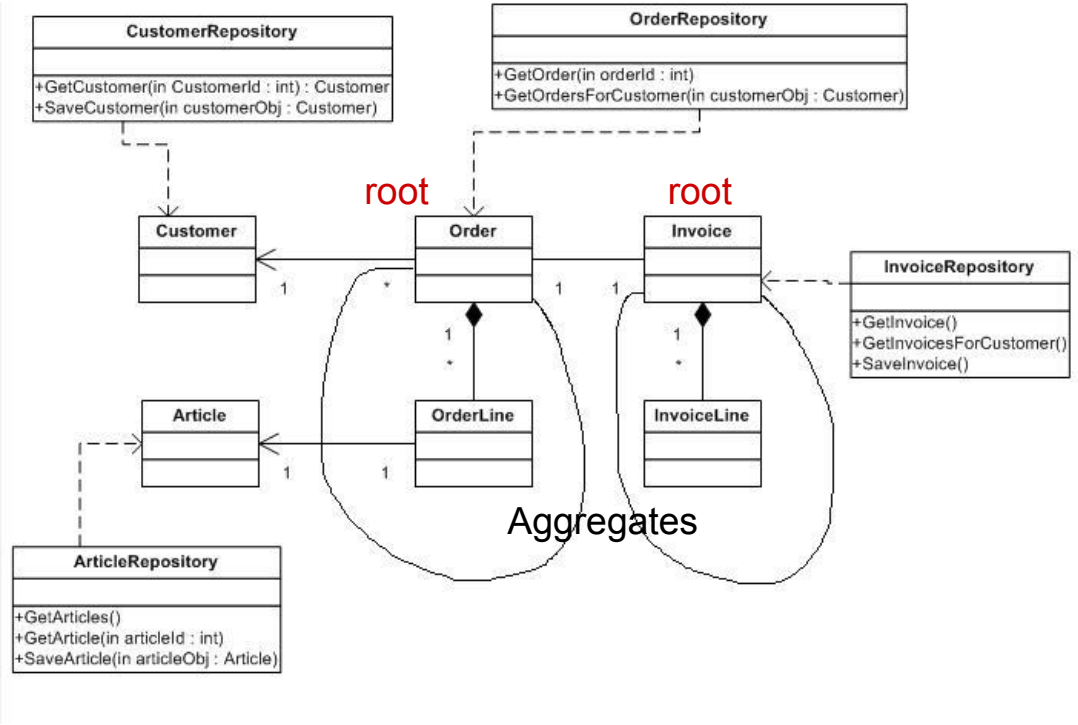
Confusing ...

- Aggregation, "part of" relationship (hollow diamond)
- Composition, more specific than aggregation (filled diamond). Objects have same "life cycle"



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)

Root has global unique identity

Others have identity inside aggregate only

Establishing Class Dependencies

Many (most?) class dependencies established during construction of application

Careful construction process will clarify and reduce dependencies

- Separate construction and use

Using new

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

```
// 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)
}
```

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."

Aside: This is impossible if using static classes

Separating Construction from Use

Dependencies passed in via constructor (best) or method (ok)

- No (few) new in code (Java standard classes ok)
- Preferable pass in interface type

```
// Passing in dependency in constructor
public class MyClass {
    // No new here
    private final IMyDependency m;
    public MyClass(IMyDependency m){     // Pass in!
        this.m = m;
    }
}
```

Centralize Construction Process*

Class dependencies passed in via constructor, but where does it happen?

- Use (static) factories. Design Pattern: Factory method

```
// Using a static factory to construct
public class MyFactory{
    // Factory method
    public static IMyModule getModule(/*possible param*/){
        // All new here
        IMyInterface i = new MyImplementation();
        IMyModule m = new MyModule(i);  // Pass in!
        return m;
    }
}
```

Aside: Interfaces

Interfaces never have methods for creation or destruction

- Construction is part of the implementation not the specification

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, explosion of methods...!
String outPutDir = ctxt.
getAbsolutePathOfScratchDirectory();

// Reasonable (supply some data, avoid too big method)
OutputStream out = ctxt.createScratchFileStream
(outPutDir);
// Some other code use stream to write
```

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

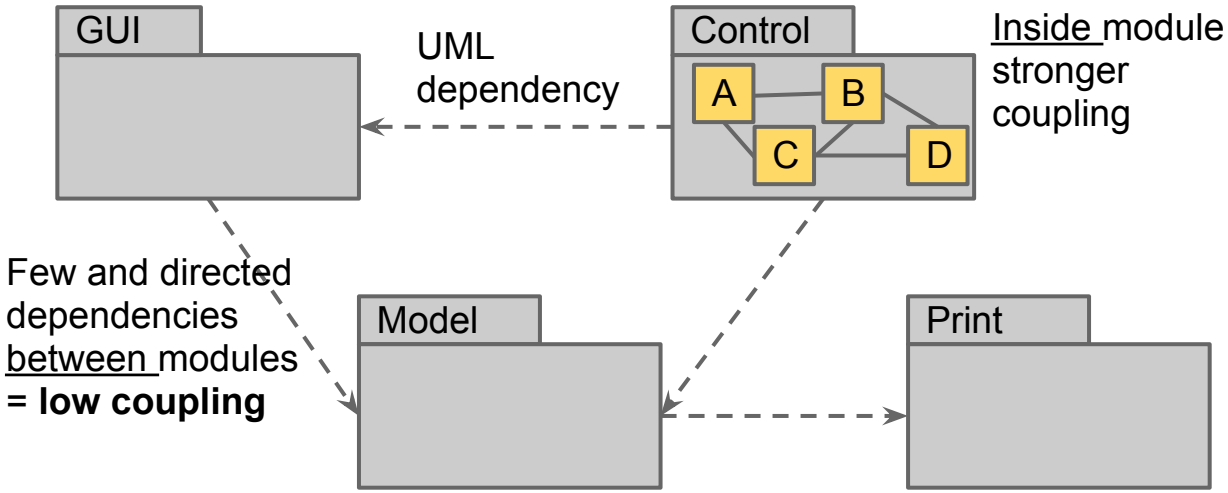
Modules

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**
- Module == Subsystem (in this course)

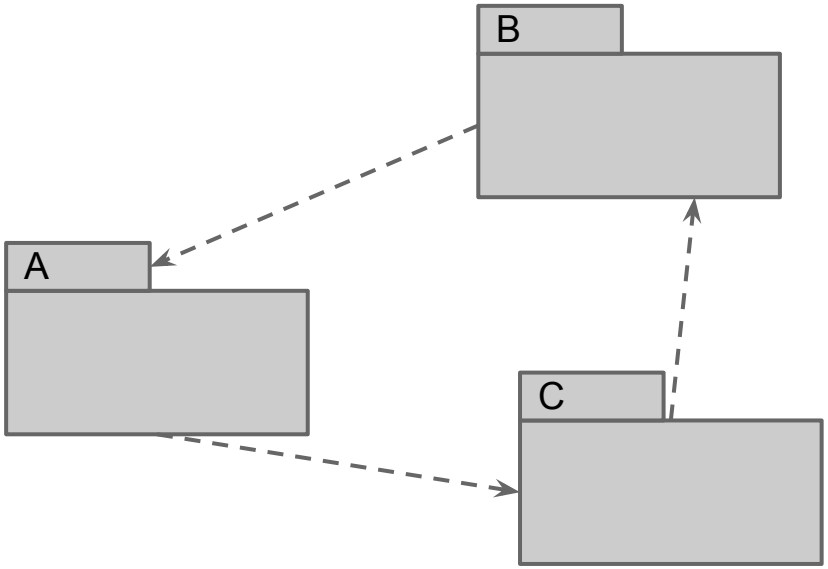
Coupling and Cohesion

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 (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)

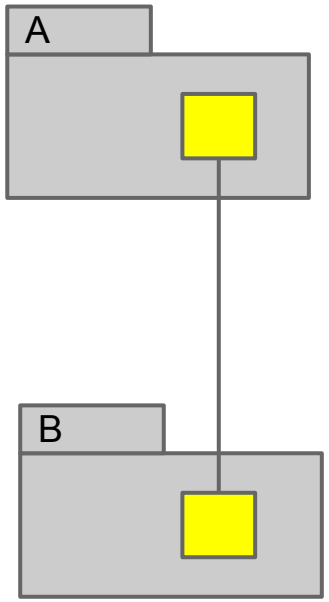
- Two modules share the same global data (a global variable, global coupling)

Message coupling (lowest)

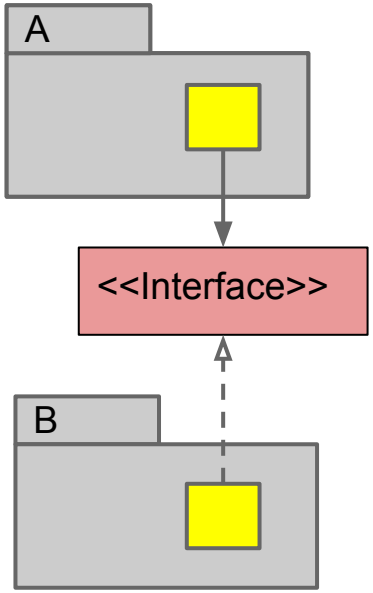
- Achieved by state decentralization (as in objects) and component communication with parameters/return values (or messages)

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

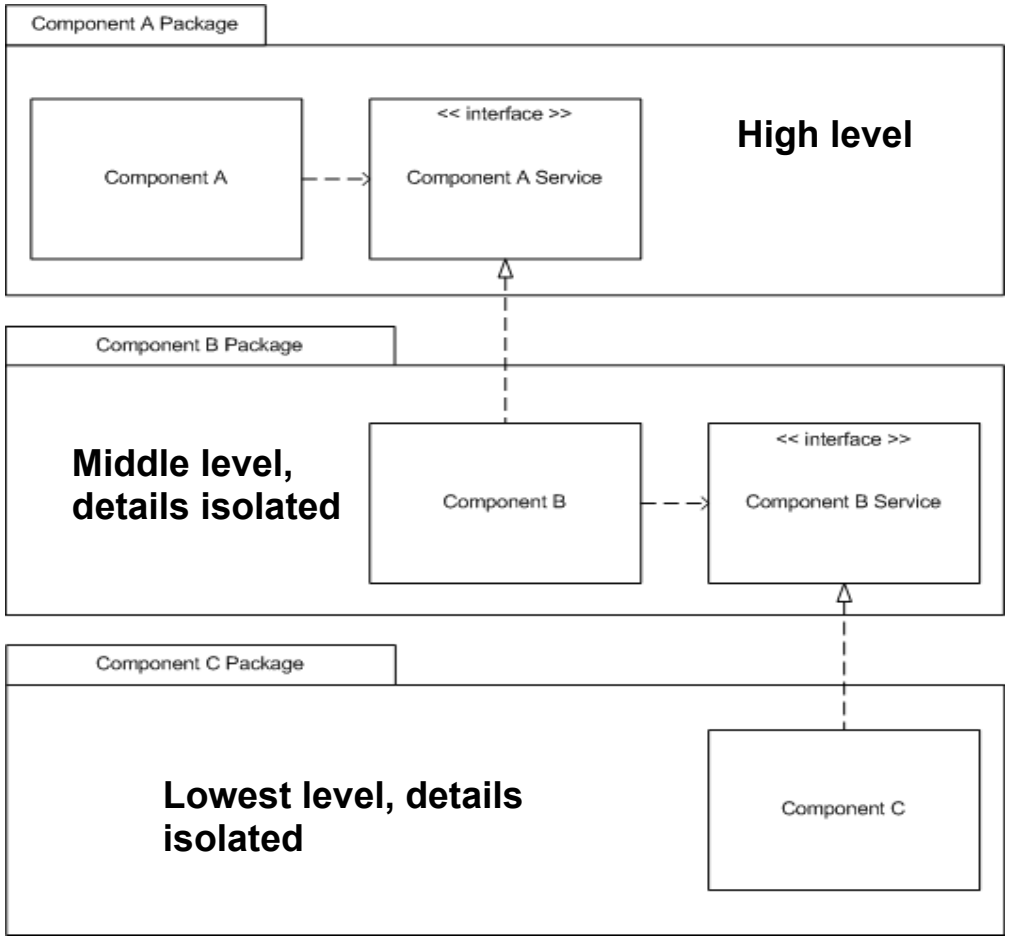
"The level of complexity by which a system is viewed. The higher the level, the less detail. The lower the level, the more detail. The highest level of abstraction is the single system itself. The next level would be only a handful of components, and so on, while the lowest level could be millions of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

(used in all lab's)



Tools to Inspect Dependencies

Many tools to inspect module dependencies

- STAN, Eclipse plugin
- JDepend
- .. many more

Other Miscellaneous Problems

... on the following slides...

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 (DRY)

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.

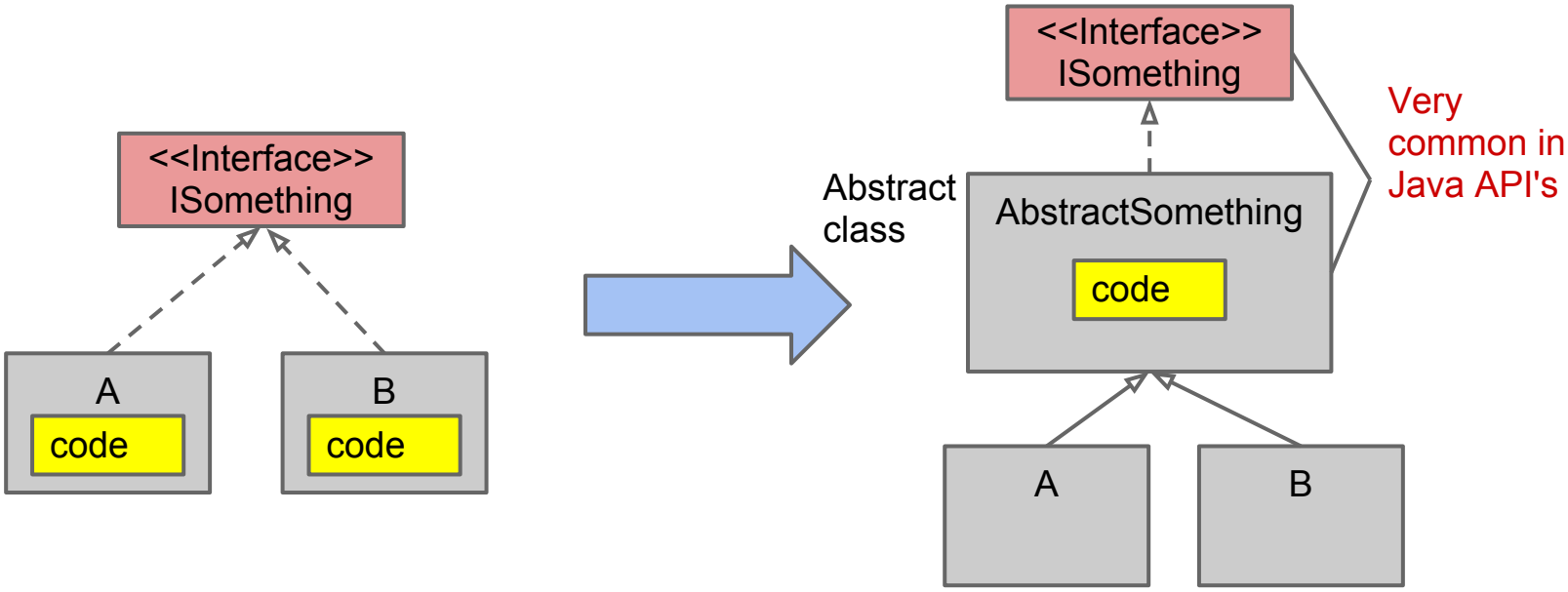
Abstract Method

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

```
// 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

Exposing Implementation

Creates a Dependency on Implementation Details

```
// A list is used
public List<A> getAllA()

// Possible better, don't know details, just a collection
public Collection<A> getAllA()
```

The Iterator Design Pattern

Makes it possible to traverse a collection without knowing the representation

Could be a

- List
- Linked List
- Set
- Tree
- ...

Iterator in Java*

```
// Interface to traverse some collection
public interface Iterator<E>
    public boolean hasNext();
    public E next();
    public void remove();

// A class capable of returning an iterator
public interface Iterable<T>
    public Iterator<T> iterator();
```

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
```

- Except Map (which isn't a Collection, doesn't implement Collection)

Implementing an Iterator*

Created by use of inner classes

Have seen

- Inner class (have reference to enclosing)
- Inner static (nested top level class)

There's also*

- **Local inner** classes, class defined inside method. Can use parameters and local variables <u>even after method has terminated</u> (variables must be final - declared)
- **Anonymous inner** classes, as local but anonymous

Implementing an Iterator, cont

Two principally different ways

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

- Remember size of collection when creates, check size before any operation

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

- Have to create a copy

More Java Iterators

```
// 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);

}
```

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 this here possible also will have at other locations, ...

Environment dependencies*

Environment

- Path file separator...

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