Design Patterns (in Java and C#)

Citigroup

August 2015

garth.gilmour@instil.co

Overview

- This is a three day course
  - Core hours and breaks are flexible…
- The course has four goals
  - Introduce pattern based development
  - Explain the classic 'Gang of 4' patterns
  - Show how these work in Java, C#, Ruby, Scala etc…
  - Discuss patterns used in functional languages
- Please control the course
  - Ask as many questions as possible
  - Speed up or slow down the pace
  - Request extra examples and exercises
  - Don't sit in misery!!

Introducing Patterns

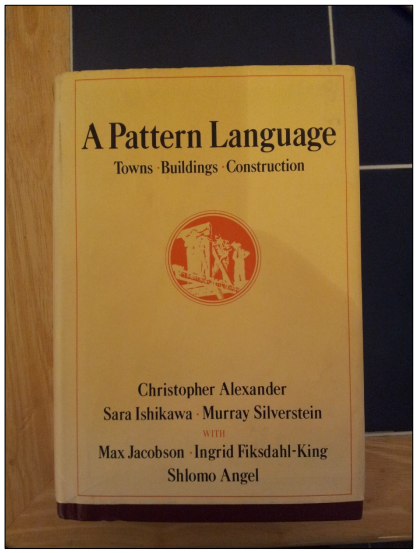
Why Design Patterns Matter

garth.gilmour@instil.co

What is a Pattern?

- The idea of patterns originates in architecture
  - From the books and ideas of Christopher Alexander
- Patterns were designed to provide a common language
  - Which would enable those involved in a new building or town to discuss and decide how the construction should progress

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
>
> Christopher Alexander – A Pattern Language



Patterns In Architecture

Light On Two Sides Of Every Room

Locate each room so that it has outdoor space outside it on at least two sides, and then place windows in these outdoor walls so that natural light falls into every room from more than one direction.

Ceiling Height Variety

Vary the ceiling heights continuously throughout the building, especially between rooms which open into each other, so that the relative intimacy of different spaces can be felt. In particular, make ceilings high in rooms which are public or meant for large gatherings (10 to 12 feet), lower in rooms for small gatherings (7 to 9 feet), and very low in rooms or alcoves for one or two people (6 to 7 feet).

Christopher Alexander – A Pattern Language

Patterns Are Everywhere! ☺

tvtropes.org

Bullying a Dragon

"Let me get this straight: You think that your client, one of the wealthiest, most powerful men in the world, is secretly a vigilante who spends his nights beating criminals to a pulp with his bare hands. And your plan is to blackmail this person? [beat] Good luck."
— **Lucius Fox**, *The Dark Knight*

See that guy over there? The one that can make Your Head A Splode with his Psychic Powers? What a *weirdo*. Let's throw rocks at him!

This is the fate-tempting and suicidal tendency of characters to bully, persecute, or in some other way provoke people or things they *really* shouldn't be messing with. That weird loner who sits in a corner reading? Fine. That sweet cute girl who can heal people? If you're that much of an asshole, go for it. The blind kid that somehow knows what you're about to do and is powerless to stop you? Yeah, jackass, whatever floats your boat. But the kid who can warp the fabric of reality and just wants to be left alone? **Bad idea**.

This can't end well.

A Sub Trope of All of the Other Reindeer, where the character is surrounded by tormentors even though they are *known* to have some incredible power conducive to being a Person of Mass Destruction, and most of the time *because* of this. This frequently crops up in Kids Are Cruel (in which case it would be "Kids Are Cruel And Also Too Retarded To Deserve To Exist"). It's usually a way of getting us to sympathize with the main character, but, really, bullies should be smart enough not to mock the "freak" Blessed with Suck and Super Strength. Even when logically —or at least using the basest level of human decency and smallest inkling of self preservation—these bullies should find a weaker target or cut the poor kid some slack. So, in a sense, Straw Bullies. Then again, Youth Is Wasted on the Dumb and the Prideful.

The Fettered especially have it bad, because they *choose* not to fight back, and often protect their tormentors from the Forces of Evil.

What is a Pattern?

- A pattern captures design experience
  - In object oriented software development
  - In a way that benefits other people
  - Using a standard template and UML
- A pattern is rarely original
  - It formalizes and classifies existing knowledge
    - This is why patterns have many names
  - Many have been around for as long as OO
    - All have been proved in different systems
  - They existed as part of the 'tools of the trade'
    - The wisdom you picked up with experience

What is a Pattern?

- A pattern addresses a problem
  - A situation within a design that cannot be modelled simply in terms of collaborating objects
  - The problem is not specific to a particular industry
    - Otherwise it couldn't be generic
    - The same issues recur in all OO projects
- A pattern is not specific to one OO language
  - Otherwise it is known as an Idiom
- A pattern improves design rather than code quality
  - Otherwise it is known as a Refactoring

The Standard OO Toolbox

- Classes and objects
- Fields (instance & static)
- Methods (instance & static)
- Polymorphism
- Encapsulation
- Inheritance
  - From concrete classes
  - From abstract classes
  - Implementing interfaces
- Composition
- Dependency

What is a Pattern?

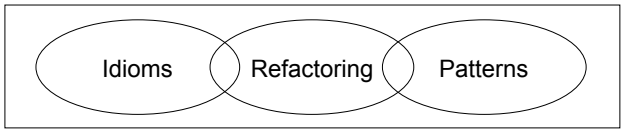
- A pattern provides a solution
  - An unusual arrangement of objects that can collaborate to produce required functionality
    - Patterns are unusual combinations of the OO 'toolbox'
  - It is expressed in terms of roles and responsibilities
    - Usually described with the aid of UML diagrams
- A pattern has consequences
  - Its use benefits the design in various ways
    - Especially if the maintainers of the code are 'pattern savy'
  - But there may be associated costs and trade-offs
    - Only use a pattern if it solves a problem for you

Patterns, Idioms and Refactorings

- There are three sources of help when trying to improve the design of your application
  - Patterns
    - OO guidelines for building a stable and flexible architecture
  - Idioms
    - Tactics and techniques for exploiting language features to improve design and performance
  - Refactorings
    - Incremental improvements that enhance code quality
- In each case the intention is the same
  - They are accumulated industry wisdom which has been generalized, standardised and documented

Patterns, Idioms and Refactorings

- Patterns, refactorings and idioms do the same things at different levels of your architecture
  - In a single language environment they can merge
- There is some overlap between the two
  - Similar idioms can exist in multiple languages
  - Big refactorings can resemble patterns
- Heavyweight IDE's are trying to automate all three
  - Via wizards, refactoring engines, intellisense etc…



Benefits of Patterns

- Patterns raise the level of abstraction
  - People remember 'chunks' of information and patterns enable you to see many classes as a single 'chunk'
- Patterns speed up development
  - When a pattern based solution is proposed the designers quickly grasp the positive and negative implications
  - Development tools can generate skeleton classes for you
- Patterns encourage discussion
  - The design is easier to describe and document
  - New developers become productive faster

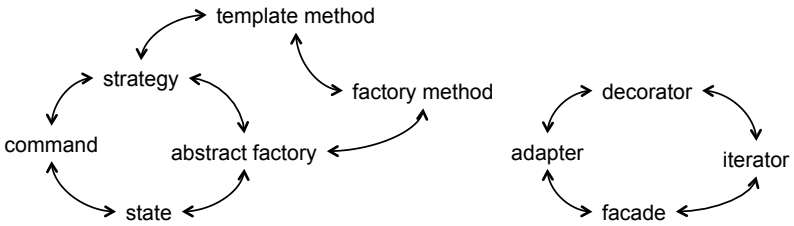
Problems with Patterns

- Projects can become ‘pattern happy’
  - Always searching for the right pattern when a simple ‘flat objects’ design using the normal OO toolbox would work
    - This increases complexity without any real benefit
  - Investing too much time in up front design
- Developers can become confused
  - By the number and variety of patterns available
    - This is the opposite effect intended
  - Knowledge of a few simple patterns goes a long way
    - A pattern should simplify the design rather than complicate it
    - Never introduce a pattern simply to try something new

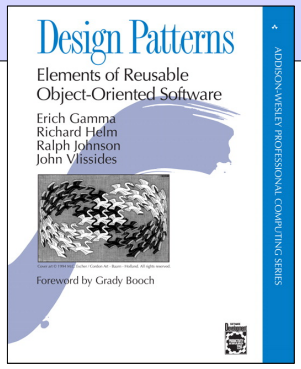
‘Golden Hammer Syndrome’

Creating Links Between Patterns

- The more you learn about patterns the more you will discover similarities and overlapping functionality
  - All patterns are clever combinations of standard OO concepts
  - You can find many diagrams mapping the links between patterns, but it is best to make your own

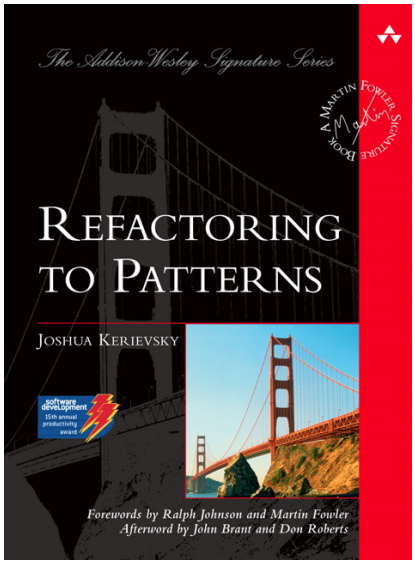


> The enormous success of design patterns is a testimonial to the commonality seen by object programmers. The success of the book Design Patterns, however, has stifled any diversity in expressing these patterns.
>
> *Kent Beck*



Refactoring To Patterns

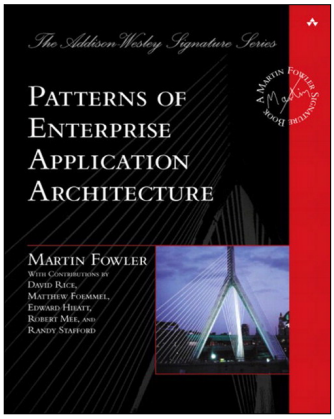
- An intriguing idea is using refactoring to get to patterns
  - We incrementally convert messy code into a pattern
  - The code starts with poor quality and ends up as a pattern
  - Invented in 'Refactoring to Patterns' by Joshua Kerievsky
- The benefits over a 'Grand Design' approach are:
  - The code is valid at each stage (it runs and passes tests)
  - The incremental stages can be checked into the VCS
  - We can pause the process if other priorities take over

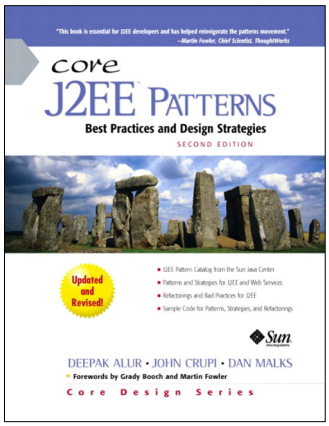


Other Forms of Patterns

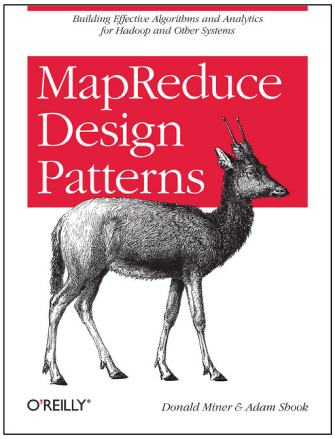
- Design Patterns are mainly associated with OO
  - But you can find patterns in many other areas
    - The 'toolbox' metaphor is widely applicable across IT
  - Although not everyone likes the term…
- Patterns are also used in:
  - Other types of programming
    - Such as functional languages (Scala, F#, Haskell etc…)
  - Certain types of architecture
    - Enterprise apps, networking, trading platforms etc…
  - Concurrency libraries

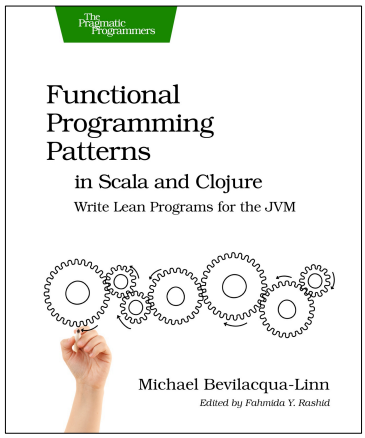
Other Forms of Patterns





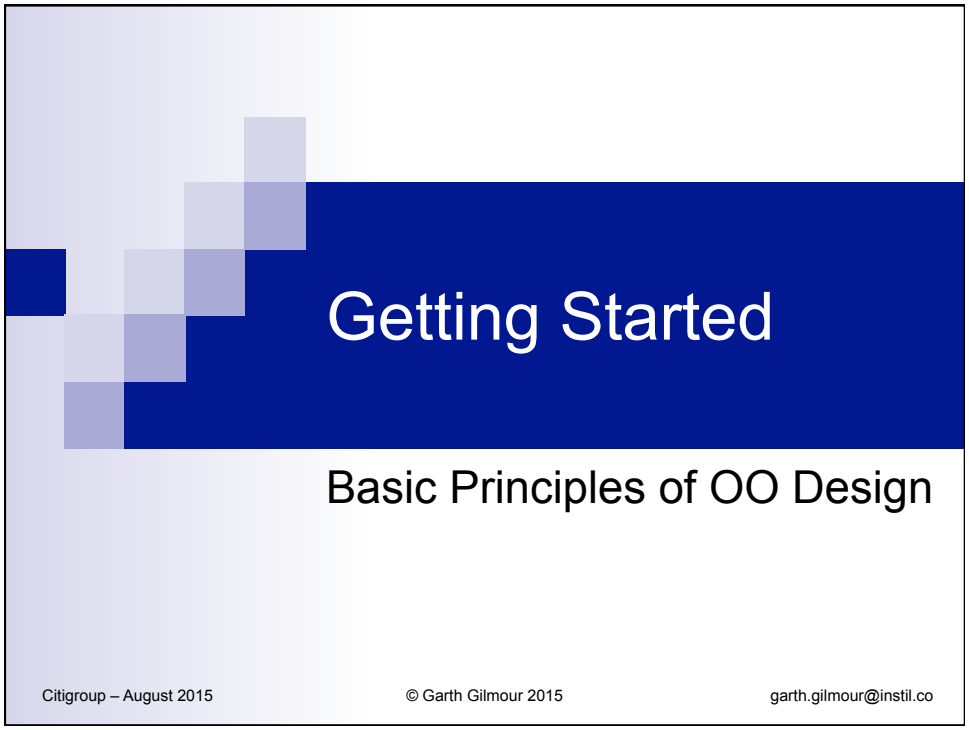
Other Forms Of Patterns





Patterns and the OO vs. FP Debate

- Patterns exist because your 'toolkit' is limited
  - You do not have a specific tool for every coding task
- Both the OO and FP toolkits have limitations
  - As the two approaches are orthogonal these limitations are often the mirror image of each other
- Blending the two makes some patterns redundant
  - C# has been an OO/FP hybrid since VS2008
  - Java became a hybrid with the release of version 8
  - Scala, F#, Swift etc… were conceived as hybrids



Getting Started

- There are certain principles of software development that live above the level of design patterns
  - They apply to any type of 'modern' software project
  - Often they contradict what was taught in the 70's and 80's
- The most important principles of OO design have been grouped under the acronym SOLID
  - The term was first coined by Robert Martin circa 20003
  - The principles themselves have been around for decades
- There are other principles worth considering
  - For example the 'Law of Demeter'

Don't Repeat Yourself

- The DRY principle permeates the IT industry
  - It was first defined in the 'Pragmatic Programmer' book
- Database normalization is an example of DRY
  - If the same information is encoded twice then the update or deletion of one entry can make the DB inconsistent
- Refactoring aims to eliminate code duplication
  - E.g. If we find subclass methods that perform the same logic we replace them with a single base class method

> "every piece of knowledge must have a single, unambiguous, authoritative representation within a system"
>
> The Pragmatic Programmer

One Example of DRY'ness

> Every non-key attribute 'must provide a fact about the key, the whole key, and nothing but the key'
>
> *Bill Kent*

> …so help me Codd
>
> *common addition*

The More Minimal the Better

- When in doubt choose to write less code
  - At the design stage try to introduce fewer abstractions
  - The more entities you put in your design the greater the effort required to understand and extend it
- This contradicts the way OO used to be taught
  - We were encouraged to generalize and invest extra effort in producing code that would be reusable across designs
  - This implies that we can successfully predict the future
- Only one version of a solution should be in a design
  - Older versions commented out is a 'code smell'

The More Minimal the Better

The More Minimal the Better

Code Is Overhead

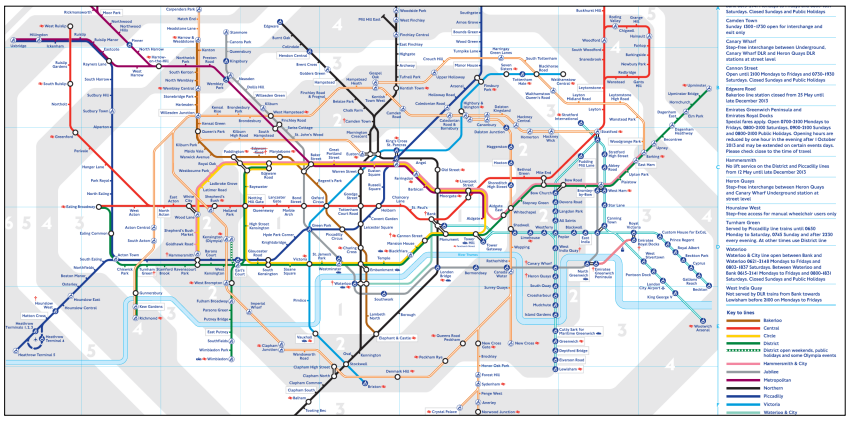
The Simpler the Better

- The simpler the design the better
  - When learning a language or framework there is a natural desire to apply as many features as possible
    - This also applies to Design Patterns and UML Diagrams
  - Experienced developers strive for simplicity
    - They try to use as few features as possible
- Simplicity is often achieved in stages
  - As we become more familiar with the problem
- The simplest solution is not always obvious
  - The simplicity may only become apparent once you have gained enough familiarity with the problem domain

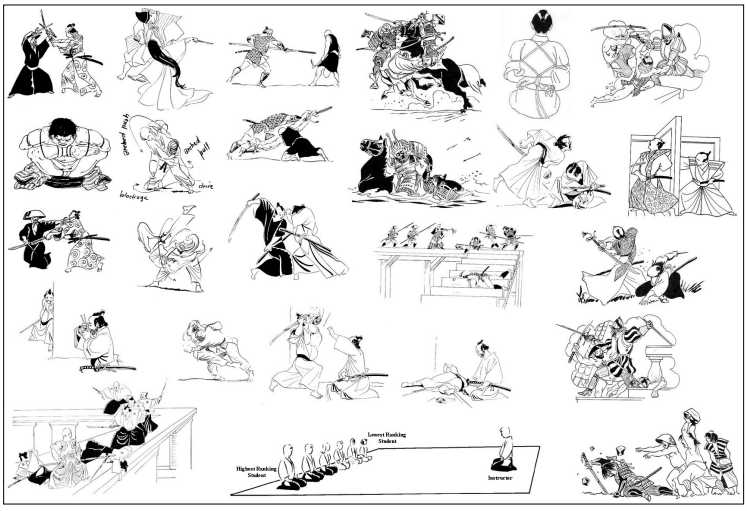
The Simpler the Better

- In an ideal world the simplest solution would also be the one that gave the best performance
  - Unfortunately this is not always the case
- During earlier decades performance was king
  - Leading to some horrible hacks in C and C++
- The modern approach is more pragmatic
  - We never knowingly introduce performance issues
  - But we only add complexity to improve performance when testing has proved that it is worth the effort
    - Today's optimization is tomorrows maintenance problem
    - Improvements in compiler/VM design often deprecate them

The Simpler the Better



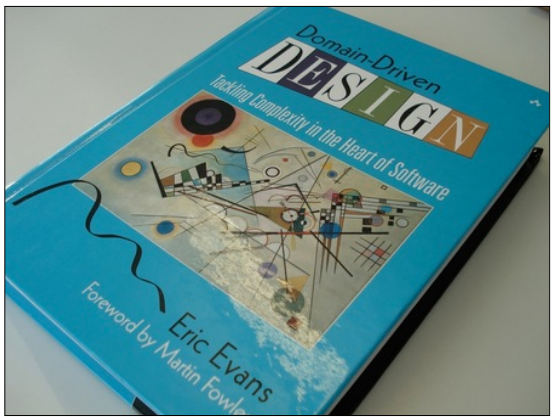
The Simpler the Better



Choosing Good Symbol Names

- Picking proper names is an obvious best practise
  - But it is often a source of problems and confusion
- Some best practises are:
  - Draw names from your language and problem domain
    - Make sure all problem domain terms are documented
    - Never use abbreviations unless they are unambiguous
  - Always be consistent in method names
    - E.g. don't used ‘get’, ‘find’ and ‘retrieve’ interchangeably
  - Names should reveal your intention
    - Package names represent the project and subsystem
    - Class and method names represent the abstraction
      - Class names should be nouns and method names verbs

Choosing Good Symbol Names



Anglo-EU Translation Guide

What the British say	What the British mean	What others understand
I hear what you say	I disagree and do not want to discuss it further	He accepts my point of view
With the greatest respect…	I think you are an idiot	He is listening to me
That's not bad	That's good	That's poor
That is a very brave proposal	You are insane	He thinks I have courage
Quite good	A bit disappointing	Quite good
I would suggest…	Do it or be prepared to justify yourself	Think about the idea, but do what you like
Oh, incidentally/ by the way	The primary purpose of our discussion is…	That is not very important
I was a bit disappointed that	I am annoyed that	It doesn't really matter
Very interesting	That is clearly nonsense	They are impressed
I'll bear it in mind	I've forgotten it already	They will probably do it
I'm sure it's my fault	It's your fault	Why do they think it was their fault?
You must come for dinner	It's not an invitation, I'm just being polite	I will get an invitation soon
I almost agree	I don't agree at all	He's not far from agreement
I only have a few minor comments	Please re-write completely	He has found a few typos
Could we consider some other options	I don't like your idea	They have not yet decided

Symmetry and Leveling

- Try to bring out underlying symmetries
  - Make sure your names reflect this symmetry
- Organize code at similar levels of abstraction
  - Don't mix high and low level code in the same method

```
public void foo() {
    open();
    doStuff();
    close();
}
```

```
public void foo() {
    begin();
    doStuff();
    end();
}
```

```
public void foo() {
    begin();
    doStuff();
    close();
}
```

```
public void foo() {
    begin();
    doStuff();
    try {
        conn.close();
    } catch(Exception ex) {
        …
    }
}
```

The SOLID Principles

- The main principles of good modern OO design are summarized via the SOLID acronym:
  - S = Single Responsibility Principle
  - O = Open / Closed Principle
  - L = Liskov Substitution Principle
  - I = Interface segregation principle
  - D = Dependency Inversion Principle
- Architectures that follow the SOLID principles should be straightforward to test, maintain and extend
  - Modern best practices like TDD, Domain Driven Design and DI containers push you towards the SOLID ideals

Single Responsibility

- A class should do a single job well
  - It should never have multiple responsibilities
  - There should not be an ‘and’ in the class description
- The is reflected in the classic MVC pattern
  - A **view** class communicates with the outside world
  - A **model** class represents a problem domain concept
  - A **controller** class implements a business process
- MVC violations are a maintenance nightmare
  - GUI classes that contain business logic
  - Business logic classes that contain DB code
  - Model classes that depend on frameworks

Open / Closed

- We do not want to alter working code
  - When new requirements arrive we would rather extend working code than rewrite it (possibly introducing bugs)
  - Our classes should therefore be “open for open for extension but closed for modification”
    - Bertrand Meyer invented this principle back in 1988
- One example is the ‘Template Method’ pattern:
  - The base class provides a separate method for each logical step in performing its single responsibility
    - Any of these methods can be overridden in a subclass
  - This is the approach used in the Struts web framework

Liskov Substitution

- Liskov Substitution is simple to understand
  - When you design a subclass its instances must be usable everywhere an instance of the base class would be
  - Without altering any of the programs 'desirable properties'
- In some ways this is enforced by the language itself
  - E.g. an overriding method cannot be less accessible than the base version or throw addition types of exception
- The principle is often related to 'design by contract'
  - Every class defines a contract via its public methods
  - A subclass cannot weaken any of the pre-conditions / post-conditions or invariants defined by this contract

Interface Segregation

- This principle deals with 'interface pollution'
  - The situation where a class is forced into implementing methods it does not need to meet its responsibility
  - It states that "clients should not be forced to depend on interfaces that they do not use"
- Consider the design of a GUI library
  - Where we are using interfaces to define event handlers
  - We want an interface for mouse-specific events
    - But there are many different types of event relating to the mouse
    - Entered, exited, dragged, moved, pressed, released etc...
  - If we define one interface client classes will suffer
    - They will have to 'stub out' all the methods they don't need

Other Useful Principles

- There are other best practises besides SOLID:
  - Try to follow the 'Law of Demeter'
  - Favour composition over inheritance
  - Favour wide class hierarchies over deep ones
  - Keep classes immutable wherever possible
  - Consider the runtime environment
- Also try to avoid the 'Not Invented Here' syndrome
  - Mistrusting external tools / frameworks / methods etc... Simply because they did not originate from your culture
  - But note that this also has to be balanced against the risk of coupling your design to an externally managed entity

Following the Law of Demeter

- This is a lesser known rule of OO design
  - It is summarized as 'only talk to your friends'
- An object should only make method calls against objects that it is directly connected to
  - Objects which it references through associations, dependencies or which are in a global scope
  - E.g. 'project.calcManagersSalary()' instead of 'project.getDept().getManager().calcSalary()'
- The Law of Demeter produces cleaner designs
  - You can understand each section of code in isolation
  - There is no need to trace links through the source code

Following the Law of Demeter

- The Law of Demeter enables Unit Testing
  - One of the biggest problems testing a class in isolation is invisibly mocking out its dependencies
  - The more dependencies the class under test can see the more mock classes you need to create
- Fewer links makes approaches like TDD feasible
  - Unit tests can fully isolate one class from all the others
  - Tools like 'JMock' are more likely to work for you
- Obeying the law is not all or nothing
  - Full compliance requires too many intermediate methods
  - Try to build in compliance from the design stage onwards

Composition Verses Inheritance

- Inheritance is bad for encapsulation
  - As encapsulation is more important we should be willing to give up inheritance when required
- Inheritance introduces a binary dependency
  - Clients have to be given the full definition of the base
  - There is no way of withholding the base class definition
- Changes to the base class can break subclasses
  - This is especially true for interfaces, where adding a new method automatically breaks all implementations
- Only use inheritance for 'IS A KIND OF' relationships
  - In all other cases prefer composition

Modelling Inheritance Hierarchies

- Class hierarchies should be wide rather than deep
  - The deeper the hierarchy the greater the number of fields and methods you need to take account of
  - Hierarchies which are 'bushy' are easier to maintain
- Only leaf classes should be concrete
  - All base classes should be abstract (where possible)
- All fields should be private without exception
  - Protected or package level accessor methods can be used to allow derived classes guarded access to base fields
  - This is more effort but pays off in the long run

Creating Immutable Classes

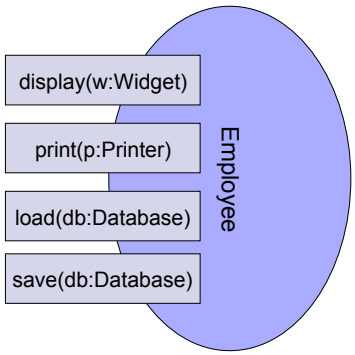
- Immutable classes are side-effect free
  - They can be passed around between layers and shared between threads without introducing any errors
- If a class can be immutable it should be
  - This is the case for core classes like 'String' and 'Integer'
  - Objects that pass values between layers are good candidates for immutable classes
- Immutable classes introduce performance issues
  - On every state change the instance needs to be cloned

The MVC Pattern

Using Model/View/Controller

garth.gilmour@instil.co

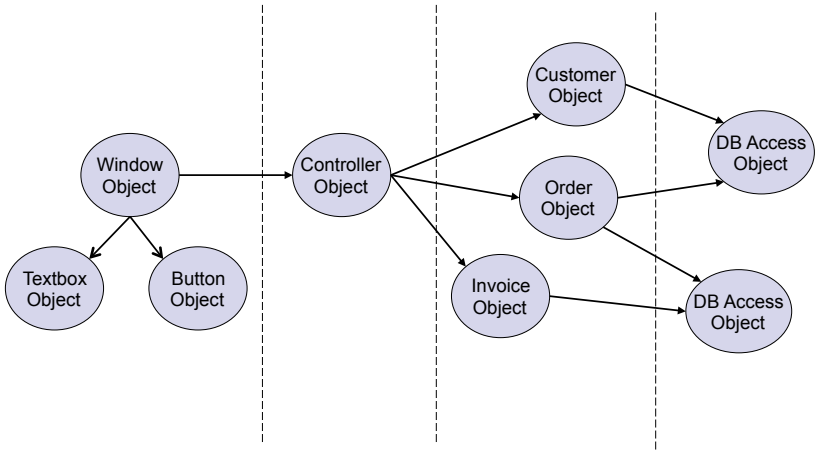
Discussion: Is this a good idea?



Model View Controller

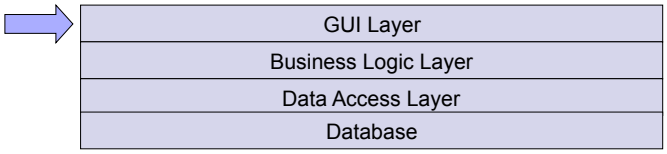
- MVC is the most pervasive pattern of all
  - It is used in every OO framework and software design
- The Problem
  - You want to modify as little code as possible when the requirements for your system change
- The Solution
  - Divide your classes into three types:
    - Model classes represent things (invoices, orders, claims etc…)
    - View classes talk to outside systems (users, networks etc…)
    - Control classes manage the flow of information
  - A class is either a Model, a View or a Controller (hence M-V-C)

Model View Controller



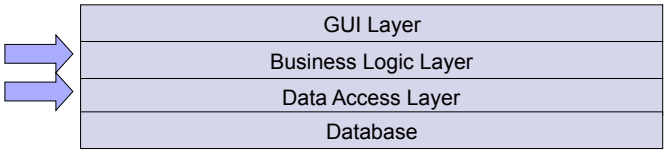
MVC & Architectural Layers

- MVC is used to create a layered architecture
  - Which neatly divides our system up into 'Tiers'
- The 'GUI Layer' contains the View classes that make up the User Interface (often called controls)
  - These controls may be provided for us or we may have created our own (called custom controls)



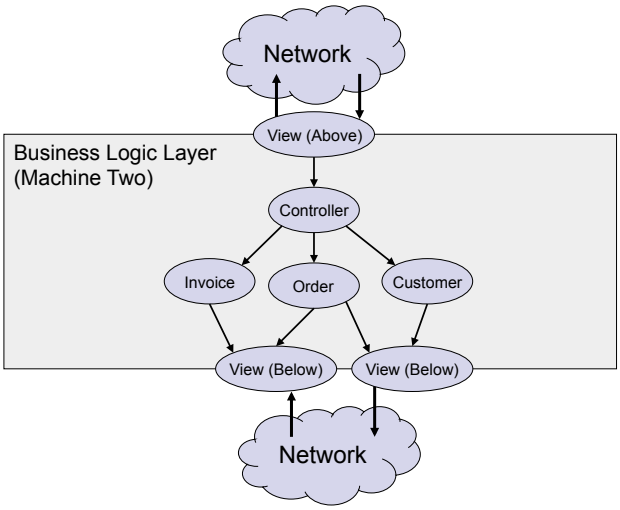
MVC & Architectural Layers

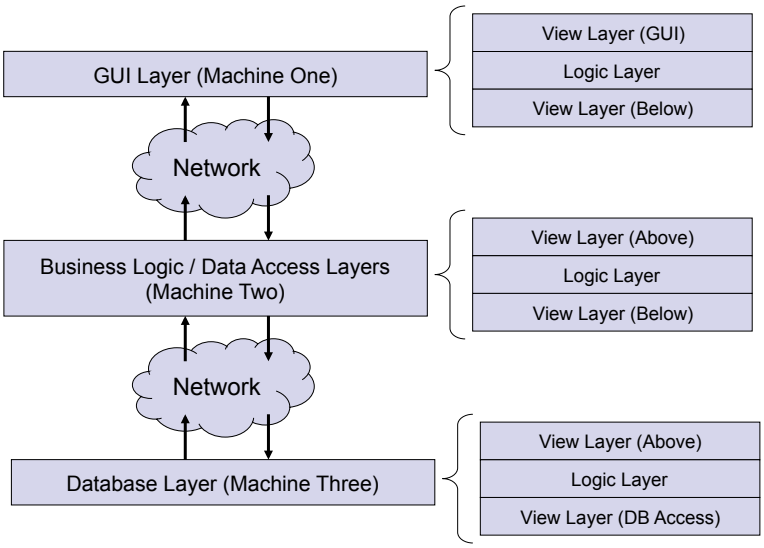
- The 'Business Logic Layer' is mostly made up of Controller and Model classes
  - These will have been written from scratch
  - Note that they have no platform or DB dependencies
- The 'Data Access Layer' holds more View classes
  - This time they are 'back end' classes that talk to the DB

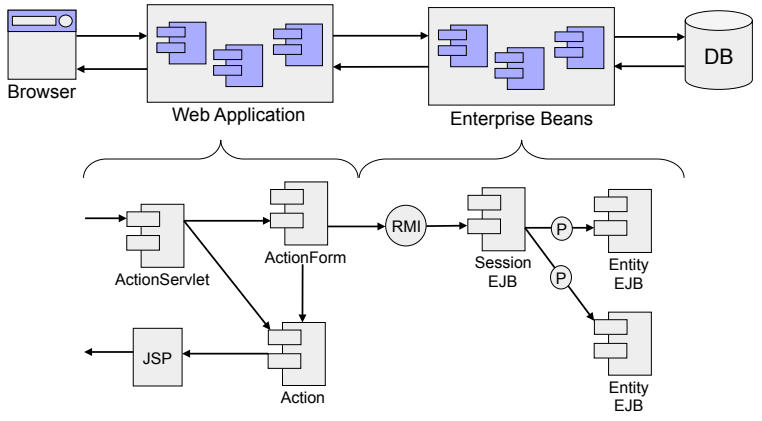


MVC and Distribution

- The importance of layers depends on deployment:
  - If deploying to a single machine layers are useful for organizing and maintaining code but have no runtime existence
  - If deploying to multiple machines layers are critical
- If we distribute the layers then each needs:
  - View classes to talk to the layers above and below
  - Controllers to manage events on the current machine
  - Data classes to pass to and fro with the layer beneath
- So each layer has its own internal M-V-C relationship
  - This is sometimes called hierarchical MVC







Little Language

Creating Your Own Language

garth.gilmour@instil.co

The Little Language Pattern

- This pattern requires considerable effort
  - But it can produce major productivity benefits for developers
- Little Language is used when:
  - You have one or more well defined data sources
  - Developers frequently write code to search, navigate, read and/ or modify these data sources
  - The code that needs to be written is lengthy, repetitive, tedious to write and hard to understand
- The solution is to create a new query language
  - So developers can write expressions in a dedicated mini-language that can automatically be turned into code

Examples of Little Language

- SQL and Regular Expressions are little languages
  - They are far more convenient than writing code to read database tables or parsing text in search of patterns
- XML has the 'XPath' query language
  - This has been built into Java since version 5
- HTML has 'CSS Selectors'
  - This was originally developed for stylesheets but can be run from JavaScript via frameworks like Dojo and JQuery
- JEE has evolved many query languages
  - Such as JPA-EL, JSP-EL, OGNL and Spring-EL

The CSS Selector Syntax in HTML

CSS Selector	Description
p	All paragraph elements in the document
p:first-child p:last-child	All paragraph elements which are the first (or last) children of their parent. Note type does not matter.
p:first-of-type p:last-of-type	All paragraph elements which are the first (or last) children of type paragraph of their parent.
p:nth-child(2)	All paragraphs which are the second child of their parent
body p	All paragraphs which are a descendent of a body element
body > p	All paragraphs which are children of a body element (not grandchildren, great-grandchildren etc…)
body div p	Paragraphs descended from a div descended from a body
div#myDivTwo p	Paragraphs inside a div which has a id of 'myDivTwo'
p[style]	Any paragraph which has a 'style' attribute

CSS Selectors in Dojo and JQuery

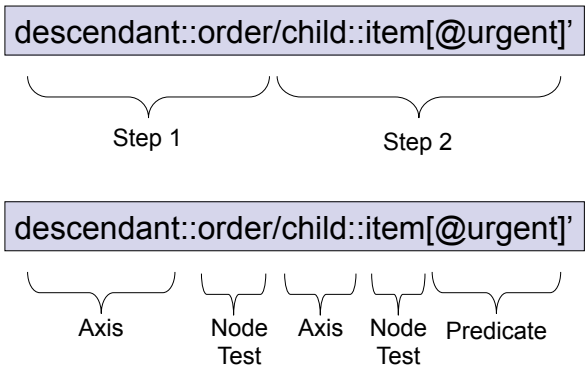
```
function runQuery(queryStr) {
    var result = "" + queryStr + "' gives: ";
    $(queryStr).each(function(index,node) {
        result += $.trim($(node).html());
        result += " ";
    });
    console.log(result);
}
```

```
function runQuery(queryStr) {
    var result = "" + queryStr + "' gives: ";
    dojo.query(queryStr).forEach(function(node) {
                                                result += dojo.trim(node.innerHTML);
                                                result += " ";
                                            });
    console.log(result);
}
```


The XPath Query Language

- XPath is a query language for XML node trees
  - It reduces the need to write code in SAX, DOM etc…
  - A query is made up of one or more steps
    - Each contains an axis, a node test and optional predicates
- An example is 'descendant::order/child::item[@urgent]'
  - The first step selects all elements named 'order' on the descendant axis and stores them in a node set
  - The second step examines each 'order' element in the first node set for children which are elements called 'item'
  - Finally the predicate at the end of the second step filters out all the nodes which do not have an 'urgent' attribute

XPath Expressions



Understanding JPA-QL

- Both the JPA and the (deprecated) Entity EJB type require a mini-language for running queries
  - This allows a developer to find multiple records without needing to know SQL, the mappings or the underlying DB schema
  - This was originally called EJB-QL but evolved into JPA-QL
- A JPA-QL query mixes relational and OO concepts
  - The syntax will be familiar to anyone who has seen SQL
  - However navigation is performed using OO associations
- Queries can be placed in XML or annotations
  - The 'EntityManager' interface contains methods for creating and running Query objects, with support for parameters and paging

```
@NamedQueries({
    @NamedQuery(name="basicQuery", query="SELECT c FROM Course c"),
    @NamedQuery(name="positionalParameterQuery",
                query="SELECT c FROM Course c WHERE c.number = ?"),
    @NamedQuery(name="namedParameterQuery",
                query="SELECT c FROM Course c WHERE c.number = :number"),
    @NamedQuery(name="returningFieldsQuery", query="SELECT c.number FROM Course c"),
    @NamedQuery(name="joiningQuery", query="SELECT DISTINCT d.course FROM Delivery d ")
})
```

```
Query query = manager.createNamedQuery("basicQuery");
List<Course> results = query.getResultList();
```

```
Query query = manager.createNamedQuery("positionalParameterQuery");
query.setParameter(1, "AB12");
Object result = query.getSingleResult();
```

```
Query query = manager.createNamedQuery("namedParameterQuery");
query.setParameter("number", "AB12");
Object result = query.getSingleResult();
```


The Criteria API in JPA2

- JPA2 adds a Criteria API
  - This is a curiosity for some but essential to others
- A Criteria API lets you build up JPA-QL on the fly
  - As might be required if your application lets users assemble their own customized reports
  - You can try to accomplish this via ‘StringBuilder’
    - But the code will be fiddly and open to abuse
- The API is designed as a ‘fluent interface’
  - You chain method calls in a (vaguely) readable way
  - Factory methods are used to generate objects that correspond to the types and expressions in JPA-QL

```
public static void showCriteriaQuery() throws Exception {
    EntityManager manager = factory.createEntityManager();
    manager.getTransaction().begin();

    CriteriaBuilder builder = manager.getCriteriaBuilder();
    CriteriaQuery criteriaQuery = builder.createQuery();

    Root<Course> root = criteriaQuery.from(Course.class);

    criteriaQuery.select(root.get("title"))
                .where(builder.equal(root.get("type"), "Beginners"));

    Query query = manager.createQuery(criteriaQuery);
    List<String> results = query.getResultList();

    for(String title : results) {
        System.out.printf("\t%s\n",title);
    }
    manager.getTransaction().commit();
    manager.close();
}
```


Expression Language in JSP 2.0

- JSP 2.0 added an Expression Language
  - Based on the syntax developed in JSTL 1.0
  - Support for EL is found in 'javax.servlet.jsp.e'
  - EL lets you draw info from the standard sources Instead of chains of method calls in scriptlets
- Consider '${sessionScope.basket.items[0]}'
  - The identifier 'sessionScope' is an *implicit object*
  - The compiler knows 'basket' must be an attribute
  - Objects are assumed to be beans so 'getItems' is called
  - Finally the square brackets index into an array or collection

Building Expressions

```
${sessionScope.basket.items[0]}
```

```
HttpSession session = request.getSession();
ShoppingBasket basket = (ShoppingBasket)session.getAttribute("basket");
Items[] items = basket.getItems();
out.write(items[0]);
```


EL Implicit Objects

Implicit Object	Description
pageContext	The PageContext object of the current JSP
pageScope	The PageContext attributes table as a Map
requestScope	The Request attributes table as a Map
sessionScope	The Session attributes table as a Map
applicationScope	The ServletContext attributes table as a Map
param	A Map of parameter names to strings
paramValues	A Map of parameter names to string arrays
header	A Map of header names to strings
headerValues	A Map of header names to string arrays
cookie	A Map of cookie names to Cookie objects
initParam	A Map of web.xml parameter names to strings

Little Language Pattern & Scripting

- Java has traditionally been a single language platform
  - Despite the fact that bytecode is language neutral
  - The complexity of the language has made it unappealing for certain types of development (e.g. QA and Sys Admin)
- Recently there has been lots of interest in developing scripting languages compatible with the JSE
  - For writing test scripts, server pages, utility programs etc...
- A dominant Little Language has yet to emerge
  - Jython, Ruby and Javascript / Rhino are ports of existing languages whilst Groovy and BeanShell are bespoke

Groovy - a Little Language for Java

```
def myList = [ new Person('Joe',27), new Person('Jane',28),
               new Person('Bob',29), new Person('Betty',30) ]

myList.each {
    println("\t $it")
}
myList.each {
    println("\t $it.name aged $it.age")
}
def result = myList.findAll { p ->
    p.age > 28
}
result.each {
    println("\t $it")
}
result = myList.collect { p ->
    return new Person(p.name,p.age + 1)
}
result.each {
    println("\t $it")
}
```

Domain Specific Languages

- DSL's are the re-emergence of an old idea
  - In the 1980's there was a trend for companies to create their own single-purpose languages for internal use
  - These had the benefit of simplicity but forced the companies to become compiler and IDE vendors
- Virtual Machines have given the idea new impetus
  - Architects can graphically set out the rules of a language
  - A bytecode compiler and IDE is automatically generated
  - The output is compatible with Java libraries and other DSL's
  - Both Java and .NET tool vendors are working on DSL tools

Factories and Patterns

Constructing Complex Objects

garth.gilmour@instil.co

Factories and Patterns

- Factories are essential to Object Orientation
  - They decouple a client from responsibility for creating an object
- Factories return base class or interface references
  - The actual type of the object is hidden from the client
  - The type may change as the system is maintained or enhanced
- Factories are essential when choosing between vendors
  - Such as JDBC Drivers, XML Parsers or Transaction Managers
- The idea of a factory is too basic to be a pattern
  - But there are two patterns based on the idea of factories

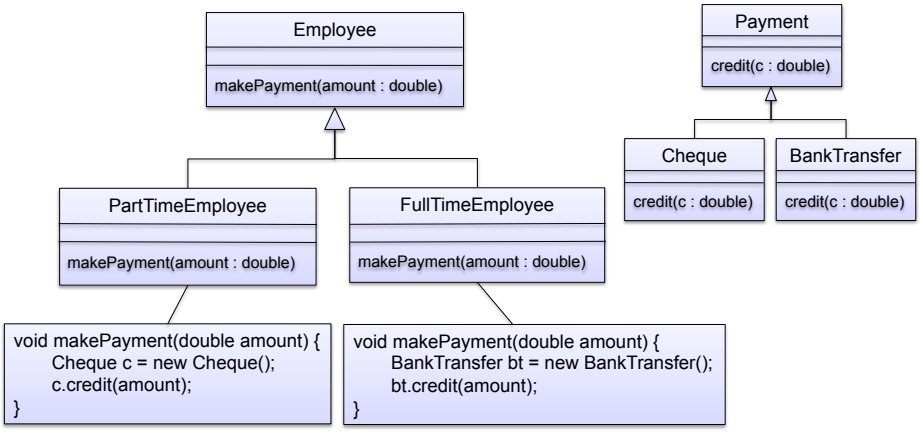
Object Factories in JCA / JCE

```
public void doSymmetricEncryption() {
	KeyGenerator generator = KeyGenerator.getInstance("DES");
	SecretKey key = generator.generateKey();

	Cipher cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
	cipher.init(Cipher.ENCRYPT_MODE,key);
	IvParameterSpec iv = new IvParameterSpec(cipher.getIV());

	RandomAccessFile input = new RandomAccessFile("data/plaintext.txt","r");
	byte[] clearText = new byte[(int)input.length()];
	input.readFully(clearText);
	printData("Original Text", clearText);

	byte[] cypherText = cipher.doFinal(clearText);
	printData("Encrypted Text", cypherText);

	cipher.init(Cipher.DECRYPT_MODE,key,iv);
	byte[] decryptedText = cipher.doFinal(cypherText);
	printData("Decrypted Text", decryptedText);
}
```

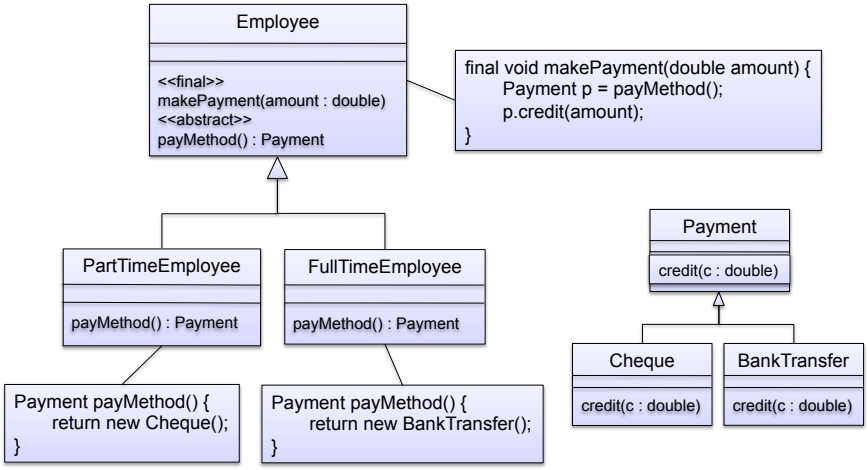
The Factory Patterns

- 'Factory Method' is a straightforward pattern
  - In class 'A' all creation of 'B' objects is done via a factory method
  - Classes inheriting from 'A' can override this as required
    - So if a method of 'A' is called as part of a derived object then the overridden version of the factory method is triggered
    - Hence the derived layer can choose the type of the dependency
- 'Abstract Factory' is more complex
  - Define an abstract base class for an object factory
    - This will hold several abstract factory methods
  - Create a derived factory class for each platform, vendor etc…
  - Create a factory method that builds the correct factory object

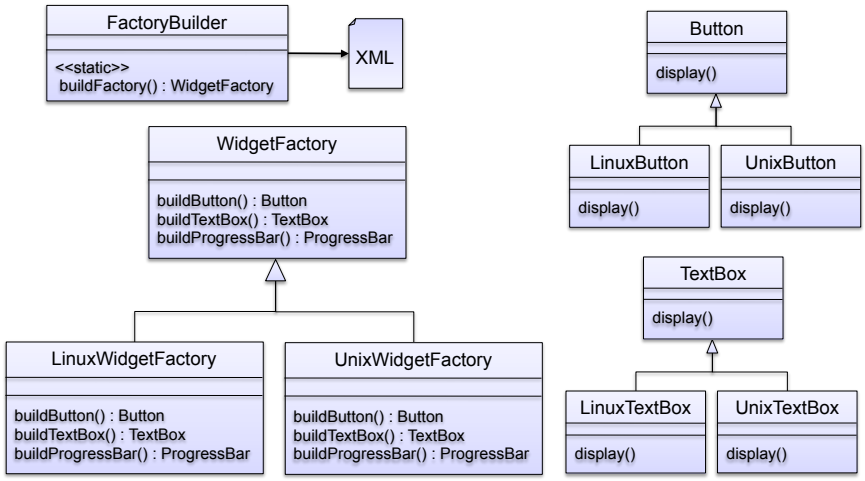
Factory Method Pattern - Before



Factory Method Pattern - After



The Abstract Factory Pattern



Abstract Factory As Used in JAXP

```
public void buildDocument() throws Exception  {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();
      Document doc = builder.newDocument();

      doc.appendChild(doc.createComment("This document describes a customer"));
      Element docElement = doc.createElement("Customer");
      doc.appendChild(docElement);

      Element name = doc.createElement("Name");
      name.setAttribute("title","Mr");
      docElement.appendChild(name);

      Element forename = doc.createElement("Forename");
      forename.appendChild(doc.createTextNode("Joe"));
      Element surname = doc.createElement("Surname");
      surname.appendChild(doc.createTextNode("Bloggs"));
      name.appendChild(forename);
      name.appendChild(surname);
}
```


Singleton Pattern

Creating Only One Instance

garth.gilmour@instil.co

The Singleton Pattern

- Singleton is the easiest pattern to understand
  - But the implementation can become very complex
- Singleton is used when:
  - There should only be a single instance of a class
  - The instance needs to be easily accessible to clients
  - You (usually) want to defer creating the instance
- Examples usually fall into two categories
  - Classes that parse and represent static resources
    - Such as configuration files and access control lists
  - Classes that communicate with external systems
  - Classes that manage collections of objects
    - For example within a persistence framework

The Singleton Pattern

- Singletons are a better choice than utility classes
  - Classes which only contain static methods
  - A Singleton can perform initialisation and cleanup tasks, retain state and be extended via inheritance
- A Singletons uniqueness needs to be protected
  - The instance should be obtained indirectly
  - Constructors should be inaccessible to clients
- Error conditions need special attention
  - The Singleton object must be impossible to crash
  - The Phoenix Singleton pattern extends Singleton by allowing an instance to crash and then be replaced by another

Implementing Singleton

- To create a Singleton class
  - Declare all the constructors as private
  - Add a private static field and a static method which returns the content of the field, initializing it if required
    - The method is usually called ‘instance’ or ‘getInstance’
- Usually thread safety is a concern
  - If ‘getInstance’ can be called from multiple threads then it must be synchronized to avoid race conditions
- The code is simpler if you don’t need deferred creation
  - The field can be made read-only and initialized on startup
  - E.g. ‘private static readonly S instance = new S();’

Implementing Singleton

```
class Singleton {
    private Singleton() {
        //initialization goes here
    }
    public static synchronized Singleton getInstance() {
        if(null == instance) {
            // if the method was not synchronized two threads
            //  could simultaneously see that ‘null == instance’
            //  and hence the line below would be run twice
            instance = new Singleton();
        }
        // always return the same instance
        return instance;
    }
    private static Singleton instance;
}
```

Singleton and Thread Safety

- 'Double Checked Locking' is an idiom used when implementing the Singleton pattern
  - It aims to remove the overhead incurred by acquiring a lock every time the 'getInstance' method is called
  - We perform an unlocked check first and only try to acquire a lock if we need to recheck and initialize the field
- The idiom should not be used in Java pre V5
  - Because it was possible for the field to be set to point to the instance before its construction was complete
    - From Java 5 this can be avoided by declaring the field 'volatile'
  - Even when it works DCL should be avoided
    - Due to the low overhead of uncontested locking in modern VM's

Double Checked Locking

```
class Singleton {
    private Singleton() {
    }
    public static Singleton getInstance() {
        if(null == instance) {
            // First call but with the potential of a race condition
            synchronized(Singleton.class) {
                if(null == instance) {
                    // First call without the possibility of race conditions
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
    private static volatile Singleton instance;
}
```

Singleton in JEE / .NET

- The common implementation of Singleton is based on referencing the instance via a static field
  - There is one copy of the field inside the virtual machine
- This assumes your application runs inside a single VM
  - JEE applications may run inside a server that is clustered onto multiple VM's spread across a local network
  - This may not be the case at present but you don't want to introduce a design feature that will limit scalability
- Creating distributed Singleton objects is very difficult
  - Fortunately you usually want a local Singleton in each VM
  - There are numerous ways to do this without static fields

Singleton and DI Containers

- Singletons in enterprise applications have been replaced by Dependency Injection containers
  - A DI container creates components on demand
    - Usually based on a name specified in a config file
  - The creation of the component my require that it be 'injected' with any number of dependencies
    - This process can be repeated to any depth
- There are several forms of DI in JEE
  - Specialized frameworks such as Spring and Guice
  - JEE5 included support for injecting built in types
  - JEE6 introduces support for injecting arbitrary types

Spring and Dependency Injection

- Spring acts as a universal object factory
  - It reads bean definitions from XML files and instantiates them
  - Both constructor and property based injection is possible
    - You can use whatever combination works best

```
public static void main(String[] args) throws Exception {
    BeanFactory factory = new XmlBeanFactory(new FileSystemResource ("config.xml"));
    Shop shop = (Shop)factory.getBean("shopWithMocks");
    if(shop.makePurchase("AB123", 20, "DEF456GHI78")) {
        System.out.println("Purchase Succeeded");
    } else {
        System.out.println("Purchase Failed");
    }
}
```

NB From Spring 3 on use generics rather than casting:

```
Shop shop =
    f.getBean(name,Shop.class);
```



```
<bean id="stockMockOne"
        class="demos.v2.StockCheckEngineMock“/>

<bean id="paymentMockOne"
        class="demos.v2.PaymentEngineMock“/>

<bean id="pricingMockOne"
        class="demos.v2.PricingEngineMock">
    <constructor-arg index="0">
        <value>500</value>
    </constructor-arg>
    <constructor-arg index="1">
        <value>10</value>
    </constructor-arg>
</bean>
```

Dependency injection using arguments to constructors

```
<bean id="shopWithMocks" class="demos.v2.Shop">
    <constructor-arg index="2">
        <ref bean="paymentMockOne"/>
    </constructor-arg>
    <constructor-arg index="1">
        <ref bean="stockMockOne"/>
    </constructor-arg>
    <constructor-arg index="0">
        <ref bean="pricingMockOne"/>
    </constructor-arg>
</bean>
```

```
<bean id="stockMockOne"
        class="demos.v3.StockCheckEngineMock“/>
</bean>
<bean id="paymentMockOne"
        class="demos.v3.PaymentEngineMock“/>

<bean id="pricingMockOne"
        class="demos.v3.PricingEngineMock">
    <property name="minimumDiscountAmount">
        <value>500</value>
    </property>
    <property name="percentageDiscount">
        <value>10</value>
    </property>
</bean>
```

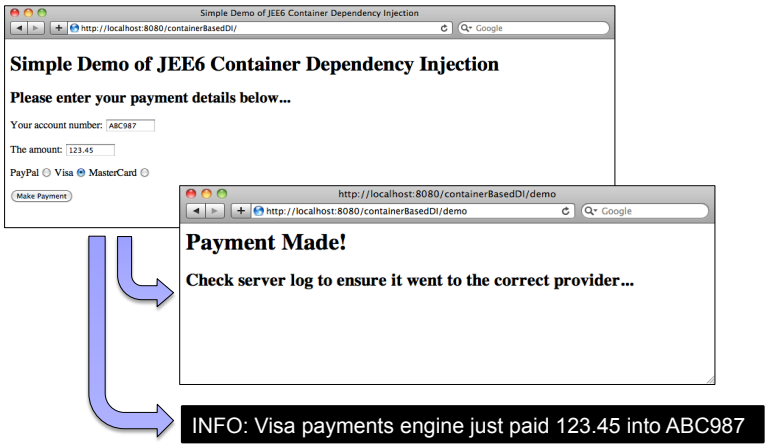
Dependency injection using properties (e.g. 'setPricingEngine')

```
<bean id="shopWithMocks"
        class="demos.v3.Shop">
    <property name="paymentEngine">
        <ref bean="paymentMockOne"/>
    </property>
    <property name="stockCheckEngine">
        <ref bean="stockMockOne"/>
    </property>
    <property name="pricingEngine">
        <ref bean="pricingMockOne"/>
    </property>
</bean>
```


Dependency Injection in JEE5

```
@Stateless
public class SomeBean implements BI {
    //Injecting container specific resources
    @Resource SessionContext myContext;
    @Resource UserTransaction myTransaction;

    // Injecting JPA resources
    private @PersistenceContext EntityManager entityManager;
    private @PersistenceUnit EntityManagerFactory entityManagerFactory;

    //Injecting JDBC and JMS resources
    private @Resource(name=“…”) DataSource database;
    private @Resource(name=“…”) TopicConnectionFactory topicFactory;
    private @Resource(name=“…”) QueueConnectionFactory queueFactory;

    //Injecting links to other EJB's ('Shop' is a business interface)
    @EJB Shop otherBean;
}
```

Dependency Injection in JEE6




```
public interface PaymentsEngine {
    public void makePayment(String account, double amount);
}
```

```
@Visa
public class VisaPaymentsEngine implements PaymentsEngine {
    public void makePayment(String account, double amount) {
        System.out.printf("Visa payments engine just paid %.2f into %s\n", amount, account);
    }
}
```

```
@PayPal
public class PaypalPaymentsEngine implements PaymentsEngine {
    public void makePayment(String account, double amount) {
        System.out.printf("Paypal payments engine just paid %.2f into %s\n", amount, account);
    }
}
```

```
@MasterCard
public class MasterCardPaymentsEngine implements PaymentsEngine {
    public void makePayment(String account, double amount) {
        System.out.printf("MasterCard payments engine just paid %.2f into %s\n", amount, account);
    }
}
```

```
@Target({FIELD,METHOD,PARAMETER,TYPE})
@Retention(RUNTIME)
@Qualifier
public @interface PayPal {
}
```

```
@Target({FIELD,METHOD,PARAMETER,TYPE})
@Retention(RUNTIME)
@Qualifier
public @interface Visa {
}
```

```
@Target({FIELD,METHOD,PARAMETER,TYPE})
@Retention(RUNTIME)
@Qualifier
public @interface MasterCard {
}
```



```
@WebServlet(name = "DemoServlet", urlPatterns = {"/demo"})
public class DemoServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
                                                                    throws IOException {
        String account = request.getParameter("accountNumber");
        double amount = Double.parseDouble(request.getParameter("amount").trim());
        String provider = request.getParameter("provider");

        makePayment(provider, account, amount);

        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writeResponse(writer);
    }
    private void writeResponse(PrintWriter writer) {
        String msg1 = "Payment Made!";
        String msg2 = "Check server log to ensure it went to the correct provider...";

        writer.write(String.format("<html><body><h1>%s</h1><h2>%s</h2></body></html>",
                                   msg1,msg2));
    }
```

```
    private void makePayment(String provider, String account, double amount) {
        if(provider.equals("paypal")) {
            paypalPaymentsEngine.makePayment(account, amount);
        } else if(provider.equals("visa")) {
            visaPaymentsEngine.makePayment(account, amount);
        } else if(provider.equals("mastercard")) {
            mastercardPaymentsEngine.makePayment(account, amount);
        }
    }
    @Inject @PayPal
    private PaymentsEngine paypalPaymentsEngine;
    @Inject @Visa
    private PaymentsEngine visaPaymentsEngine;
    @Inject @MasterCard
    private PaymentsEngine mastercardPaymentsEngine;
}
```


Bean Scopes in JEE6 CDI

- When you create a bean to be used with CDI you can control if and when instances are reused
  - CDI defines five scopes in which a bean can live
    - By default beans have what Spring calls ‘prototype scope’
  - There is an annotation for each scope

Scope Annotation	Description
@Dependent	The default. A new instance is created each time
@RequestScoped	Beans are shared across the same HTTP request, EJB remote method call, web service call or JMS delivery
@ConversationScoped	Only relevant to JSF based web controls
@SessionScoped	Beans are shared across the HTTP session
@ApplicationScoped	Beans are shared for the lifetime of the application

Observer Pattern

Simplifying Notification

garth.gilmour@instil.co

Observer

- Observer is one of the most popular patterns
  - It creates a loosely coupled design
    - Often called a publish-subscribe or multicast architecture
- Observer should be applied when:
  - An object is going to change state intermittently
  - An unspecified number of other objects need to be notified when the object undergoes a state change
  - These observers need to be added and removed dynamically
  - The observed object should not know the type of the observers
    - It should only be aware of a common method to call

Implementing Observer

- The observed object maintains a list of observers
  - Its interface has methods to add and remove observers
  - The observers can be of any type
    - But all must implement a common interface
    - Usually containing only a single 'notify' method
- When a state change occurs observers are notified
  - The list is iterated and the notify methods called
- A system based on observer is very flexible
  - The list of observers can grow and shrink
  - New observer classes can be created as required

Observer and GUI Event Handling

- Event handling in Java and JEE uses interfaces
  - An interface is created for each event type
  - Methods are added for each event
- A component has methods to register event listeners
  - Instances of classes which implement event handler interfaces
- Listener objects are held in collections
  - When an event occurs the component walks over the related collection and calls the appropriate method
- The AWT and Swing libraries define many listeners
  - The most commonly used is probably 'ActionListener'

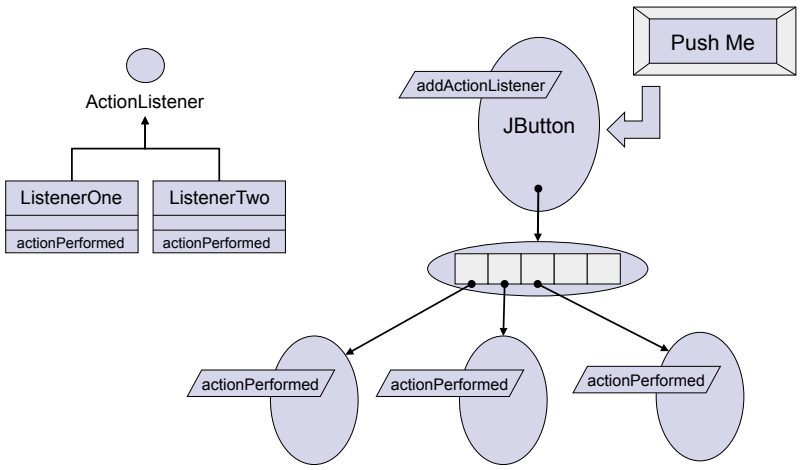
Example Listener Interfaces

```
public interface ActionListener extends EventListener {
      public void actionPerformed(ActionEvent e);
}
```

```
public interface MouseListener extends EventListener {
      public void mouseClicked(MouseEvent e);
      public void mousePressed(MouseEvent e);
      public void mouseReleased(MouseEvent e);
      public void mouseEntered(MouseEvent e);
      public void mouseExited(MouseEvent e);
}
```

```
public interface KeyListener extends EventListener {
      public void keyTyped(KeyEvent e);
      public void keyPressed(KeyEvent e);
      public void keyReleased(KeyEvent e);
}
```


Listening to Events In Swing



Event Handling

- Every event handler method takes an event object
  - This holds a reference to the component that generated the event and details of the context in which the event occurred
- A single listener can register with many components
  - The event object is used to identify the event source
- Adapter classes simplify writing listeners
  - These implement an interface and 'stub out' all the methods
  - Allowing you to inherit from the adapter and only override the methods that are of interest

Using Adapter Classes

```
public abstract class MouseAdapter implements MouseListener {
      public void mouseClicked(MouseEvent e) {}
      public void mousePressed(MouseEvent e) {}
      public void mouseReleased(MouseEvent e) {}
      public void mouseEntered(MouseEvent e) {}
      public void mouseExited(MouseEvent e) {}
}
```



```
public class MyMouseListener extends MouseAdapter {
      // we only need to override methods that interest us…
      public void mouseClicked(MouseEvent e) {
            // event handling code goes here
      }
}
```

Problems with Observer

- Intensive use of Observer can cause problems
  - At any one time who are the registered observers?
  - In what order will they be called?
- Observer gets in the way of debugging
  - Instead of the familiar call stack of nested invocations you have to understand how the observer objects are being notified
  - The order in which observers are notified should never matter
- The pattern is also complicated by threading
  - The notify method is called in a separate thread from others that may be using the observer object
  - What happens if an observer is added or removed while the notification process is underway?

Command Pattern

Simplifying Event Handling

garth.gilmour@instil.co

The Command Pattern

- Event management needs to be centralized
  - Typically a single object has responsibility for receiving application events and triggering the associated logic
- Centralized event handling tends to degenerate
  - The event manager contains a lengthy conditional statement with one branch for each event that may arrive
  - This conditional is hard to maintain and easy to break
- The Command Pattern addresses this problem
  - Event handlers are encapsulated into objects, stored in a table and indexed by the type code of the corresponding event

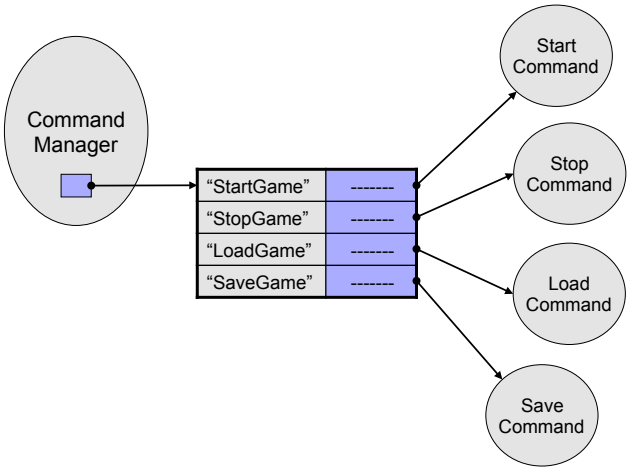
The Motivation For Command

```
private void doCommand(ActionEvent e) {
    Object source = e.getSource();

    // TODO: For every new button added to
    //    the GUI we need a new branch below
    if(source == startButton) {
        startGame();
    } else if(source == stopButton) {
        stopGame();
    } else if(source == resetButton) {
        resetGame();
    } else if(source == loadButton) {
        loadGame();
    } else if( … ) {
        // other command methods…
    }
}
```

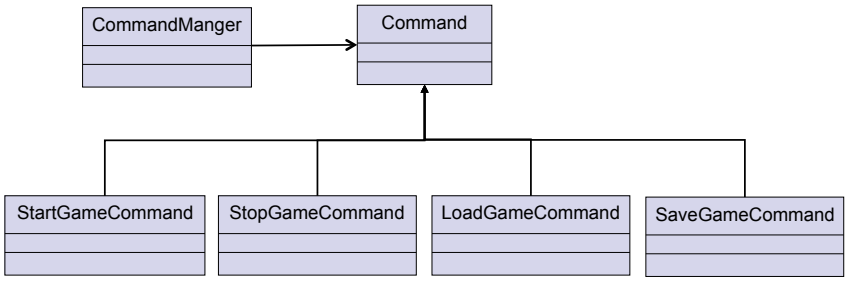
Implementing Command

- To implement the Command Pattern:
  - Create an abstract base class for commands
    - Containing an abstract method such as 'execute'
  - Create a derived class for each command type
    - Overriding the 'execute' method to hold business logic
    - If a command requires extra information this can be passed as parameters in the constructor
  - Create a 'CommandManager' class which manages a set of associations between events and command objects
    - Normally via one of the 'java.util.Map' implementations
  - When an event occurs use it as the key to find the command

Applying the Command Pattern



A Hierarchy Of Command Classes



Applying the Command Pattern

```
private void initCommands(CommandManager m) {
      m.addCommand(saveButton.getActionCommand(), new SaveGameCommand());
      m.addCommand(loadButton.getActionCommand(), new LoadGameCommand());
      m.addCommand(startButton.getActionCommand(), new StartGameCommand());
      m.addCommand(stopButton.getActionCommand(), new StopGameCommand());
}
```

```
private void doCommand(ActionEvent e) {
      String msg = e.getActionCommand();
      Command cmd = commandManager.getCommand(msg);
      cmd.execute();
}
```

Trade Offs In Applying Command

- Command has many advantages:
  - Command objects can manage as much state as required
  - Support for ‘undo’ and ‘redo’ can be added if needed
  - Logging and exception handing are simplified
  - Mock objects can be written as commands
- Command does have drawbacks
  - It is most appropriate when you have an evolving application with many disparate events that trigger logic in different objects
    - Otherwise the complexity may not be justified
  - It can lead to bidirectional associations
    - Where event generators also contain business logic

Template Method

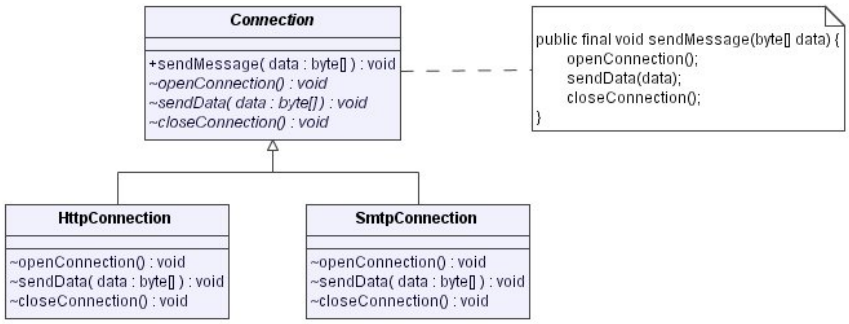
Specifying Generic Algorithms

garth.gilmour@instil.co

The Template Method Pattern

- Template Method is used when:
  - The steps of an algorithm are well defined
  - The individual steps have varying implementations
- To implement the pattern
  - A class hierarchy is created with a generic base class and subclasses for each implementation
  - The base class has a concrete method which calls the steps of the algorithm in the correct sequence
  - Each step is represented by a polymorphic method
    - This method should be abstract if there is no sensible default
  - The derived classes override the methods for steps as required

Implementing A Template Method



Examples of Template Method

- 'Thread' is a simple example of Template Method
  - You extend the base class and override the 'run' method
- Rendering in Swing uses a template method
  - The 'paint' method of 'JComponent' calls 'paintBorder', 'paintComponent' and 'paintChildren'
  - To implement custom rendering override 'paintComponent'
  - The drawing you do is buffered to reduce flicker
- Calls to Servlets also pass through a template
  - The 'service' method of 'HttpServlet' detects the type of the request and triggers the matching method
  - You derive class and implement 'doGet', 'doPost', 'doHead', 'doOptions', 'doTrace' or 'doDelete' as required

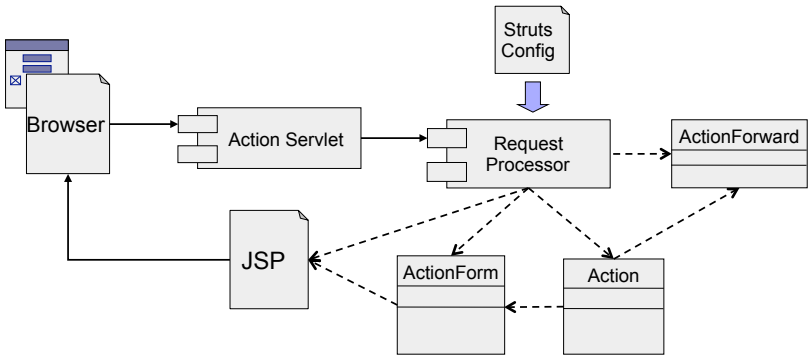
Template Method & Class Loaders

- Sometimes all the steps in the algorithm will have a default implementation in the base class
  - The pattern is used to customize the implementation
  - Normally you will only replace a single step
- This version is used to develop custom Class Loaders
  - The steps are implemented in the base class
  - Except for the 'findClass' method which is overridden to locate the specified class file and generate a Class object
    - The 'defineClass' method builds a Class object from a byte array

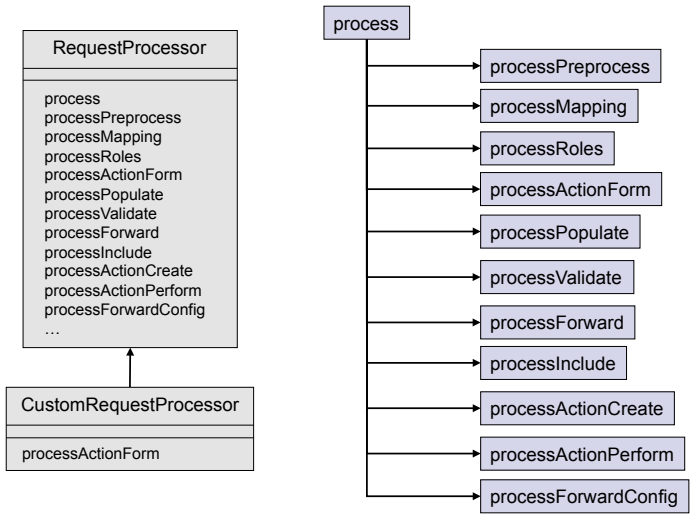
Default Template Steps and Struts

- Struts is based around Template Method
  - In V1.0 the 'ActionServlet' implemented the lifecycle
  - This made customizing single steps very difficult
- V1.1 introduced the 'RequestProcessor' class
  - The 'process' method manages the lifecycle
  - For each step there is a 'processXXX' method
    - E.g. 'processPopulate' takes parameters from an HTTP request and uses them to set the properties of an 'ActionForm' object
- Any step can be overridden to customize the lifecycle
  - The new processor is registered in the configuration file

The Struts MVC Architecture



Template Methods in Struts



Visitor Pattern

Simplifying Class Hierarchies

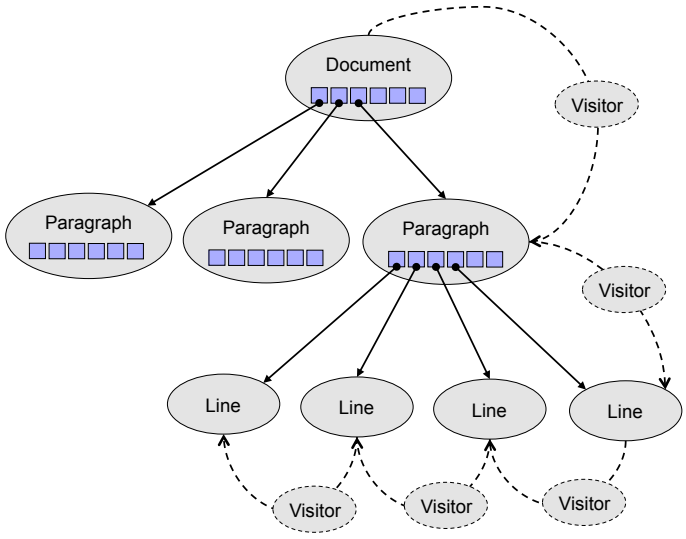
garth.gilmour@instil.co

The Visitor Pattern

- Visitor is used to add extra functionality to an established structure of collaborating objects
  - Such as a Document Object Model (DOM)
- Visitor should be applied when:
  - The object structure is stable and will not be modified
  - There is a need to add arbitrary functionality that requires access to the tree of objects in the structure
  - The extra functionality cannot be placed in methods because:
    - Adding all the functionality as methods would confuse the code
    - Different clients will need different subsets of the functionality
    - We must support adding functionality at any future time

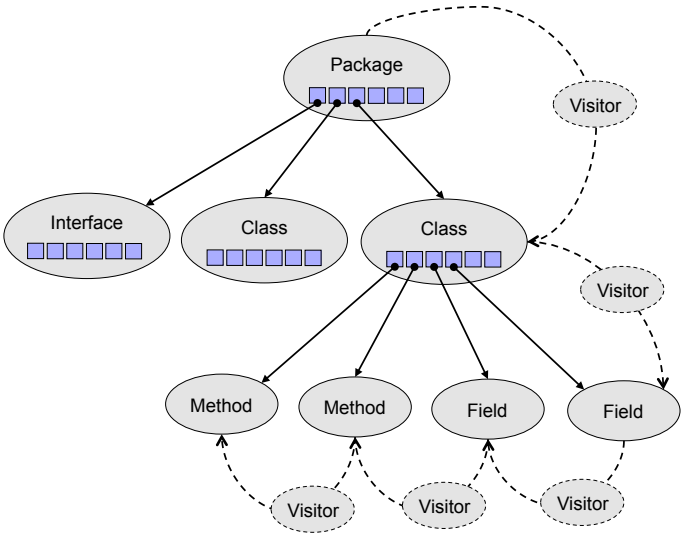
Implementing Visitor

- To implement Visitor
  - Create a hierarchy of visitors that model possible tasks
  - Create methods in the main classes to accept and use visitors
- Visitor can be difficult to implement
  - Functionality must be distributed between two classes
  - The visitor needs to access the state of the main class
  - Visitors often need a mechanism to iterate over and visit a number of classes, such as to compile a report
- The Visitor pattern is rarely used
  - But where it can be applied it is very useful



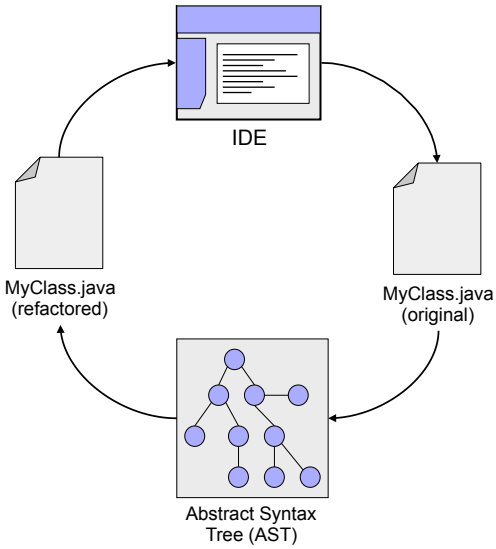
ASM Library and the Visitor Pattern

- ASM is a bytecode manipulation framework
  - It lets you process the contents of a class file
- The library is based around the Visitor Pattern
  - The ‘ClassVisitor’, ‘FieldVisitor’ and ‘MethodVisitor’ interfaces define the methods for visiting class members and bytecode
  - You could use these to map dependencies between classes, generate code quality metrics, find symbols etc…
  - If you were creating a compiler you could use it to write unit tests which validated the generated bytecode instructions
- The library also uses the Visitor Pattern in reverse
  - Unusually you can call the ‘visit’ methods in order to generate new classes and populate their methods with bytecode



Eclipse and the Visitor Pattern

- The Eclipse IDE has its own library for representing code
  - The classes are found in 'org.eclipse.jdt.core'
  - The 'ICompilationUnit' interface represents a '.java' file
- Changes can be made on a 'working copy' of the code
  - Calling 'getWorkingCopy' on a compilation unit makes the copy
  - This can be merged with the original by calling 'reconcile''
- Visitor can be used to implement a complex refactoring
  - By passing a visitor object around the Abstract Syntax Tree
  - The 'ASTParser' class converts the source into an AST
    - Which can then be navigated and manipulated




```
public class MyClass {
    public MyClass(int val) {
        this.val = val;
    }
    public void setVal(int param) {
        val = param;
    }
    public int getVal() {
        return val;
    }
    private int val;
}
```

```
public class MyClass {
    public MyClass(int val) {
        this._val = val;
    }
    public void setVal(int param) {
        _val = param;
    }
    public int getVal() {
        return _val;
    }
    private int _val;
}
```

```
public class FieldModifierASTVisitor extends ASTVisitor {
    public boolean visit(VariableDeclarationFragment node) {
        if(node.getParent() instanceof FieldDeclaration) {
            renameNode(node);
        }
        return true;
    }
    public boolean visit(FieldAccess node) {
        renameNode(node);
        return true;
    }
    private void renameNode(FieldAccess node) {
        AST ast = node.getAST();
        SimpleName oldName = node.getName();
        SimpleName newName = ast.newSimpleName("_" + oldName.getIdentifier());
        node.setName(newName);
    }
    private void renameNode(VariableDeclarationFragment node) {
        AST ast = node.getAST();
        SimpleName oldName = node.getName();
        SimpleName newName = ast.newSimpleName("_" + oldName.getIdentifier());
        node.setName(newName);
    }
}
```


Decorator Pattern

Wrapping Objects To Add Extra Functionality

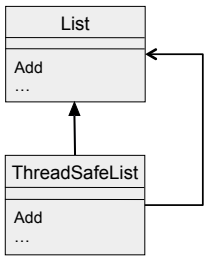
garth.gilmour@instil.co

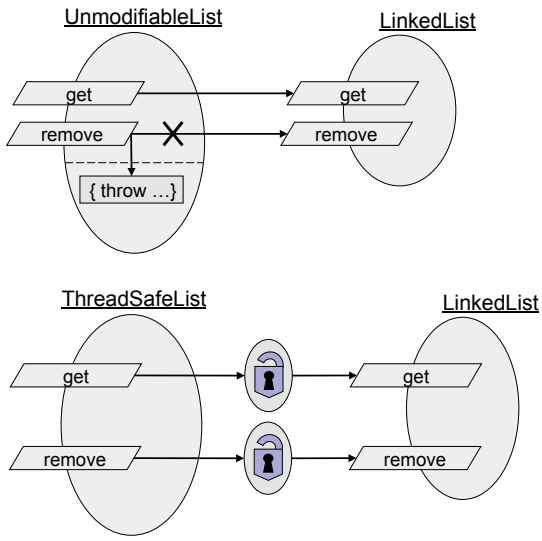
Decorator

- Decorator is a simple useful pattern
  - Decorator classes are often called ‘wrappers’
  - Both the original and the wrapper share an interface
    - So that client code cannot tell which it is using
- Decorator should be applied when:
  - You need to add extra functionality to a class
    - Such as to make a collection thread-safe or read only
  - The functionality cannot be added directly because:
    - There are reasons why the class cannot change
    - The extra functionality may need to be withdrawn
    - The extra functions are only needed in a particular context

An Example of Decorator

- Consider a requirement for a thread-safe list
  - We do not want to re-implement the container
  - Instead we want to wrap it and add locking
- So our ‘ThreadSafeList’ will:
  - Implement the ‘List’ interface
  - Contain a field of type ‘List’





Implementing Decorator

- A class which acts as a decorator:
  - Shares an interface or base class with the one to be decorated
  - Can be created out of an instance of the class to be decorated
    - In C++ this is known as a ‘conversion constructor’
- The decorator is used as if it were the original
  - The use of the decorator is transparent to client code
- Many invocations are passed to the original object
  - But some are intercepted and redirected
- Decorator has many uses
  - To make objects thread safe or constant
  - To make GUI’s scrollable, moveable etc…

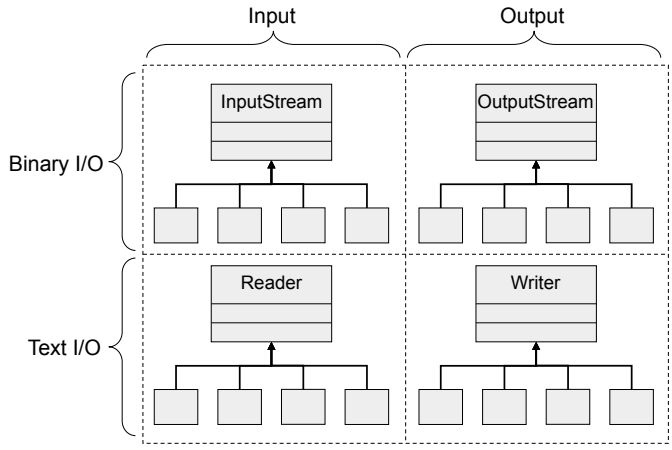
The Decorator Pattern & Streams

- Java I/O is based around the idea of streams
  - An input stream reads data from a source
  - An output stream writes data to a sink
- The decorator pattern is used to add functionality
  - Low level streams attach directly to a source or sink
  - High level (wrapper) streams add extra functionality
    - They wrap around an existing stream object
    - Examples include buffering, tokenizing and encrypting
    - Wrapper streams don't care about the underlying implementation, meaning they can be used with any data source/sink

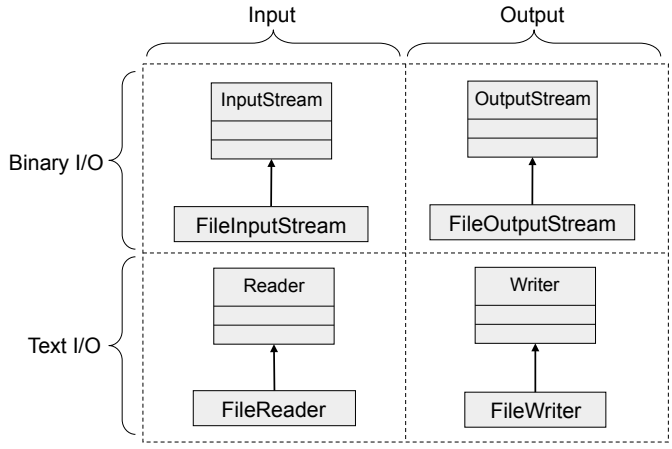
Binary and Character Streams

- Originally only input and output streams existed
  - They supported both byte and char based I/O
  - However Java 1.0 users found them too complex
- Two new hierarchies were introduced in Java 1.1
  - To simplify textual I/O and better support Unicode
  - 'Reader' is the base class for character based input
  - 'Writer' is the base class for character based output
- So 'java.io' contains four separate class hierarchies
  - Accounting for the large number of classes in the package
  - Similar classes can be found in each hierarchy
  - Always prefer readers and writers for character based I/O

Stream Hierarchies



Stream Hierarchies



File Streams

- To do binary I/O to a file
  - Create a File object and check the files permissions
  - Create a 'FileInputStream' or 'FileOutputStream' object
    - Passing the file object to the constructor
  - Wrap the object in a buffering stream
    - A 'BufferedInputStream' or 'BufferedOutputStream' object
    - This avoids the overhead of processing one byte at a time
- To do character based I/O to a file
  - Create a File object as before
  - Create a 'FileReader' or 'FileWriter', passing in the File object
  - Wrap the stream class in a 'BufferedReader' or 'BufferedWriter'
    - To avoid processing the file one character at a time

File Streams

```
public static void main(String[] args) {
    File f = new File(args[0]);
    if(!f.canRead()) {
        System.out.println("Cannot read from file!");
        System.exit(0);
    }
    try {
        BufferedReader br =
            new BufferedReader(new FileReader(f));
        String currentLine;
        int lineNum = 0;
        while(null != (currentLine = br.readLine())) {
            String msg = "Line " + ++lineNum + " is " + currentLine;
            System.out.println(msg);
        }
    } catch(FileNotFoundException ex) {
        System.out.println ("Cannot read file");
    } catch(IOException ex) {
        System.out.println ("IO Problem: " + ex);
    }
}
```

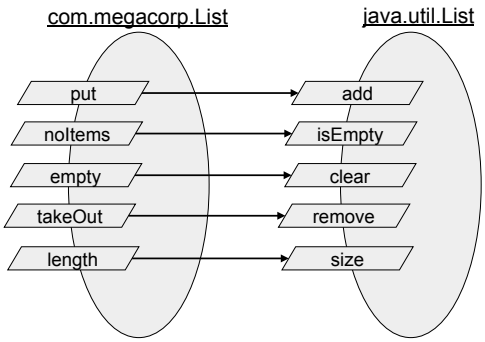
Adapter Pattern

Converting Between Interfaces

garth.gilmour@instil.co

The Adapter Pattern

- Adapter is similar in structure to Decorator
  - But the underlying intention is very different
- Again one object 'wraps up' another
  - But instead of adding a new service the wrapper converts calls from one interface to another
- Consider the following problem
  - You have your own 'java.util.List' implementation
  - You need to support the 'com.megacorp.List' interface
    - Which has the same methods but with different names
  - The solution is to apply Adapter



The Adapter Pattern and Streams

- Adapter classes are used in the streams library
  - They create character streams from binary ones
- An ‘InputStreamReader’ is a type of reader
  - But it is constructed from an input stream
- An ‘OutputStreamWriter’ is a type of writer
  - But it is constructed from an output stream
- These are essential for complex tasks
  - Such as decrypting data and then reading it as text
  - You can specify which character set is to be used for the converting bytes into characters and vice versa
    - Character conversion issues can be very hard to solve

Additional Examples of Adapters

- Java itself minimises the need for Adapter
  - The class file format allows any class to implement extra interfaces without existing clients needing to recompile
- Consider the evolution of the collections library
  - When collections were introduced the 'Vector' class was retrofitted to implement the 'List' interface
  - When thread friendly collections were added in 1.5 the 'LinkedList' class was retrofitted to implement 'Queue'
- There is an example in the XML libraries
  - 'XMLReaderAdapter' enables a SAX2 implementation to be used by clients expecting a SAX1 parser

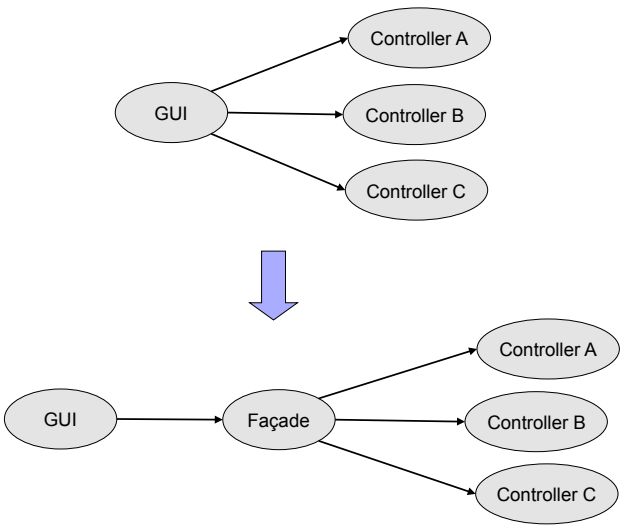
Façade Pattern

Hiding Complexity

garth.gilmour@instil.co

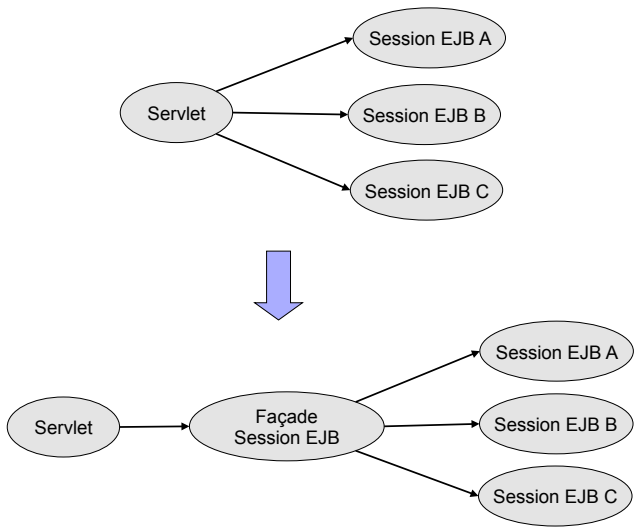
The Façade Pattern

- Façade is a very straightforward pattern
  - Although it can be tricky to apply correctly
  - It can be confused with Decorator and Adapter
- A façade class hides subsystem(s) from clients
  - It talks to the subsystem on the clients behalf but only exposes those details that clients need to know
  - Usually the client is some kind of GUI
- The pattern has several advantages
  - The client code only sees a single subsystem
    - Making it easier to write and test
  - The façade hides the 'dialog' with other subsystems



The Façade Pattern in JEE

- Façade is encouraged in JEE for two reasons
  - Because J2EE components can be divided into the simple (Servlets/JSP) and complex (EJB)
  - To minimise the number of calls made over the network
    - These are expensive and error prone
- A Stateless Session EJB is the Façade
  - It is used by the web application and communicates with other business components on its behalf
- Message Driven Beans can also be used
  - These have the advantage of adding making the business logic asynchronous so that the browser does not time out



Strategy Pattern

Encapsulating Algorithms

garth.gilmour@instil.co

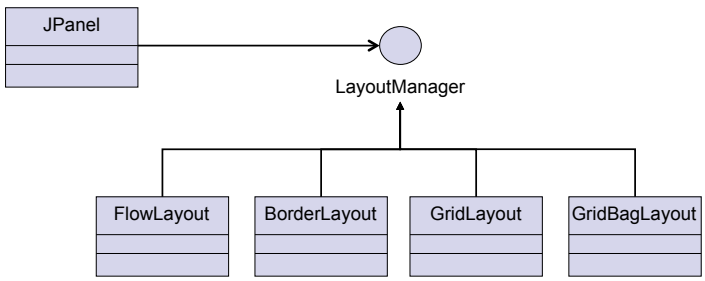
Strategy

- Strategy is a rarely used pattern
  - It separates an algorithm from the class which uses it
- Strategy is used when:
  - A class provides a service which can be implemented using a variety of different algorithms (or strategies)
  - You want clients to be able to select an algorithm
  - You want clients to be able to change the algorithm
  - Clients should not care how the algorithms work
    - Only about the benefits and drawbacks of each one
  - Bundling the algorithms within the class would not work
    - It would cause code bloat and prohibit future extensions

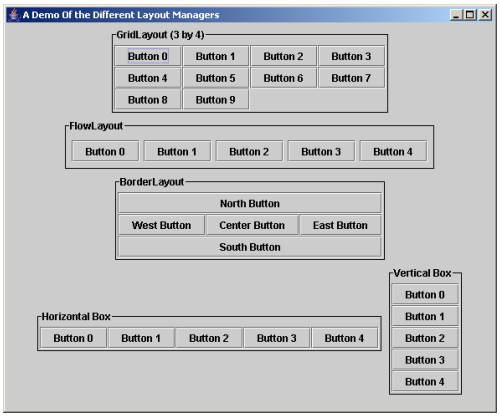
Implementing Strategy

- To implement strategy:
  - Create a class hierarchy to model possible algorithms
    - Each class in the hierarchy overrides an 'apply' method
  - The main class has a reference to one of the algorithms
    - There can be a default algorithm or selecting one can be required
- Strategy is used in cross-platform GUI frameworks
  - Absolute positioning cannot be relied on
  - A range of positioning algorithms must be available
- In Swing 'LayoutManager' objects control positioning
  - Containers are passed a manager via the 'setLayout' method
  - You can use the built in managers or create your own

Layout Managers In Swing



Layout Algorithms In Swing



A C# Demo of the Strategy Pattern

- Consider the design of a mock objects generator
  - This will build a mock based on an interface or abstract class
  - Generators are easy to create in C# because of the CodeDOM library that ships with the .NET framework
- We will need to support many kinds of mock object
  - E.g. mock objects that always throw exceptions, that set a boolean flag when called or that keep a log of all calls
- A design based around the strategy pattern is ideal
  - The generator object can create a shell for the mock object
  - Strategy objects take the shell and add the extra functionality

```
public abstract class Math {
    public abstract int add(int no1, int no2);
    public abstract int multiply(int no1, int no2);
    public abstract double divide(int no1, int no2);
    public abstract void accumulate(int num);
    public abstract int accumulateTotal();
}
```

```
public class MathMock : Math {

    public int add(int no1, int no2) {
        throw new System.NotSupportedException();
    }
    public int multiply(int no1, int no2) {
        throw new System.NotSupportedException();
    }
    public double divide(int no1, int no2) {
        throw new System.NotSupportedException();
    }
    public void accumulate(int num) {
        throw new System.NotSupportedException();
    }
    public int accumulateTotal() {
        throw new System.NotSupportedException();
    }
}
```

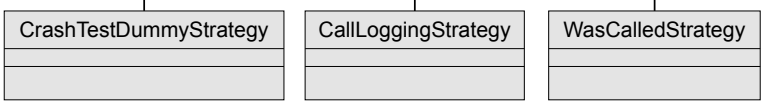


```
public class MathMock : Math {
    public bool addWasCalled;
    public bool multiplyWasCalled;
    public bool divideWasCalled;
    public bool accumulateWasCalled;
    public bool accumulateTotalWasCalled;
    public int add(int no1, int no2) {
        this.addWasCalled = true;
        return 0;
    }
    public int multiply(int no1, int no2) {
        this.multiplyWasCalled = true;
        return 0;
    }
    public double divide(int no1, int no2) {
        this.divideWasCalled = true;
        return 0;
    }
    public void accumulate(int num) {
        this.accumulateWasCalled = true;
    }
    public int accumulateTotal() {
        this.accumulateTotalWasCalled = true;
        return 0;
    }
}
```

```
public class MathMock : Math {
    public StringBuilder callLog;

    public MathMock() {
        this.callLog = new StringBuilder();
    }
    public int add(int no1, int no2) {
        this.callLog.Append("#add");
        return 0;
    }
    public int multiply(int no1, int no2) {
        this.callLog.Append("#multiply");
        return 0;
    }
    public double divide(int no1, int no2) {
        this.callLog.Append("#divide");
        return 0;
    }
    public void accumulate(int num) {
        this.callLog.Append("#accumulate");
    }
    public int accumulateTotal() {
        this.callLog.Append("#accumulateTotal");
        return 0;
    }
}
```

```
public void buildMockClass(string targetClassName,
                           string mockClassName,
                           IList strategies) {
    //Create the mock object class…

    foreach(Strategy strategy in strategies) {
        strategy.apply(klass,targetClass);
    }
}
```

```
public abstract class Strategy {
    public abstract void apply(CodeTypeDeclaration mock, Type target);
}
```




```
public class CrashTestDummyStrategy : Strategy {
    public override void apply(CodeTypeDeclaration mock, Type target) {
        CodeTypeMemberCollection mockMethods = mock.Members;
        foreach(CodeTypeMember member in mockMethods) {
            if(member is CodeMemberMethod) {
                CodeMemberMethod method = (CodeMemberMethod)member;
                CodeStatementCollection statements =  method.Statements;
                if(statements.Count > 0) {
                    CodeStatement lastStatement = statements[statements.Count - 1];
                    if(lastStatement is CodeMethodReturnStatement) {
                        statements.Remove(lastStatement);
                    }
                }
                CodeTypeReference typeToThrow =
                        new CodeTypeReference(typeof(System.NotSupportedException));
                CodeObjectCreateExpression createExpression =
                        new CodeObjectCreateExpression(typeToThrow,new CodeExpression[] {});
                CodeThrowExceptionStatement throwStatement =
                        new CodeThrowExceptionStatement(createExpression);
                statements.Add(throwStatement);
            }
        }
    }
}
```

```
public class WasCalledStrategy : Strategy {
    public override void apply(CodeTypeDeclaration mock, Type target) {
        MethodInfo[] methods = target.GetMethods();
        foreach(MethodInfo m in methods) {
            addWasCalledFieldForMethod(m,mock);
        }
        CodeTypeMemberCollection mockMethods = mock.Members;
        foreach(CodeTypeMember member in mockMethods) {
            if(member is CodeMemberMethod) {
                CodeMemberMethod method = (CodeMemberMethod)member;
                method.Statements.Insert(0,buildStatementToSetWasCalledField(method.Name));
            }
        }
    }
    private void addWasCalledFieldForMethod(MethodInfo method, CodeTypeDeclaration klass) {
        string name = method.Name + "WasCalled";
        CodeMemberField wasCalled = new CodeMemberField(typeof(bool), name);
        wasCalled.Attributes = MemberAttributes.Public;
        klass.Members.Add(wasCalled);
    }
    private CodeAssignStatement buildStatementToSetWasCalledField(string methodName) {
        CodeFieldReferenceExpression fieldToSet = new CodeFieldReferenceExpression(
                                    new CodeThisReferenceExpression(), methodName + "WasCalled");
        CodePrimitiveExpression valueToSet = new CodePrimitiveExpression(true);
        return new CodeAssignStatement(fieldToSet,valueToSet);
    }
}
```



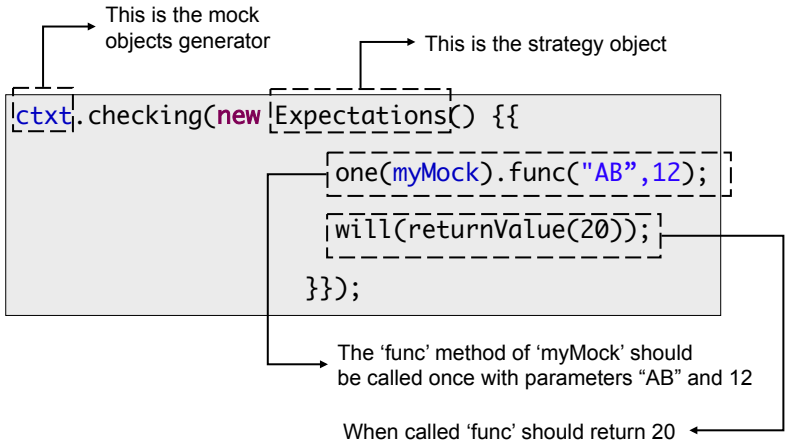
```
public class CallLoggingStrategy : Strategy {
    public override void apply(CodeTypeDeclaration mock, Type target) {
        addCallLogField(mock);
        CodeTypeMemberCollection mockMethods = mock.Members;
        foreach(CodeTypeMember member in mockMethods) {
            if(member is CodeMemberMethod) {
                CodeMemberMethod method = (CodeMemberMethod)member;
                CodeMethodInvokeExpression invocation =
                                    buildExpressionToUpdateCallLog("#" + method.Name);
                method.Statements.Insert(0,new CodeExpressionStatement(invocation));
            }
        }
        addDefaultConstructor(mock);
    }

    private void addCallLogField(CodeTypeDeclaration klass) {
        CodeMemberField logField =
                new CodeMemberField(typeof(System.Text.StringBuilder), "callLog");
        logField.Attributes = MemberAttributes.Public;
        klass.Members.Add(logField);
    }
```

```
    private void addDefaultConstructor(CodeTypeDeclaration klass) {
        CodeConstructor constructor = new CodeConstructor();
        constructor.Attributes = MemberAttributes.Public;
        CodeFieldReferenceExpression targetField =
            new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), "callLog");
        CodeObjectCreateExpression ctorCall =
            new CodeObjectCreateExpression(typeof(System.Text.StringBuilder),
                                           new CodeExpression[]{});
        constructor.Statements.Add(new CodeAssignStatement(targetField,ctorCall));
        klass.Members.Add(constructor);
    }

    private CodeMethodInvokeExpression buildExpressionToUpdateCallLog(string msg) {
        CodeFieldReferenceExpression targetField =
            new CodeFieldReferenceExpression(new CodeThisReferenceExpression(), "callLog");
        CodeExpression[] parameters =
            new CodeExpression[] { new CodePrimitiveExpression(msg) };
        return new CodeMethodInvokeExpression(targetField,"Append",parameters);
    }
}
```


The Strategy Pattern and JMock

- JMock is a popular Java mocking tool
  - It also uses the strategy pattern to control behavior
  - It takes the idea further than our C# example
- When you have created a mock object you pass it an instance of the ‘Expectations class’
  - This determines how the mock object will behave
- Expectations are placed in an initializer block
  - You inherit methods to control how many times a method of the mock should be called, what you want it to return etc…
  - This is an example of the ‘Little Language’ pattern

The Strategy Pattern and JMock




```
public class Shop {
   public Shop(PricingEngine pricingEngine,
               StockCheckEngine stockCheckEngine,
               PaymentEngine paymentEngine) {
      this.pricingEngine = pricingEngine;
      this.stockCheckEngine = stockCheckEngine;
      this.paymentEngine = paymentEngine;
   }
   public boolean makePurchase(String itemNo, int quantity,
                               String cardNo) {
      if(stockCheckEngine.check(itemNo) >= quantity) {
         double charge = pricingEngine.price(itemNo, quantity);
         if(paymentEngine.authorize(cardNo, charge)) {
               return true;
         }
      }
      return false;
   }
   private PricingEngine pricingEngine;
   private StockCheckEngine stockCheckEngine;
   private PaymentEngine paymentEngine;
}
```

```
public interface PaymentEngine {
   boolean authorize(String cardNo, double amount);
}
```

```
public interface PricingEngine {
   double price(String itemNo, int quantity);
}
```

```
public interface StockCheckEngine {
   int check(String itemNo);
}
```



```
@RunWith(JMock.class)
public class ShopTest {
    @Before
    public void start() {
        pricingEngine = context.mock(PricingEngine.class);
        paymentEngine = context.mock(PaymentEngine.class);
        stockCheckEngine = context.mock(StockCheckEngine.class);
    }
    @Test
    public void makePurchaseWorksForValidQuantity() {
        context.checking(new Expectations() {{
                            one(stockCheckEngine).check("AB12");
                            will(returnValue(20));
                        }});
        context.checking(new Expectations() {{
                            one(pricingEngine).price("AB12",19);
                            will(returnValue(27.30));
                        }});
        context.checking(new Expectations() {{
                            one(paymentEngine).authorize("010203XYZ",27.30);
                            will(returnValue(true));
                        }});
        Shop s = new Shop(pricingEngine,stockCheckEngine,paymentEngine);
        assertTrue(s.makePurchase("AB12", 19, "010203XYZ"));
    }
```

```
    @Test
    public void makePurchaseFailsForInvalidQuantity() {
        context.checking(new Expectations() {{
                          one(stockCheckEngine).check("AB12");
                          will(returnValue(20));
                        }});
        context.checking(new Expectations() {{
                          never(pricingEngine).price("",0);
                        }});
        context.checking(new Expectations() {{
                          never(paymentEngine).authorize("",0);
                        }});
        Shop s = new Shop(pricingEngine,stockCheckEngine,paymentEngine);
        assertFalse(s.makePurchase("AB12", 21, "010203XYZ"));
    }

    private PricingEngine pricingEngine;
    private PaymentEngine paymentEngine;
    private StockCheckEngine stockCheckEngine;
    private Mockery context = new JUnit4Mockery();
}
```


State Pattern

Simplifying Behaviour

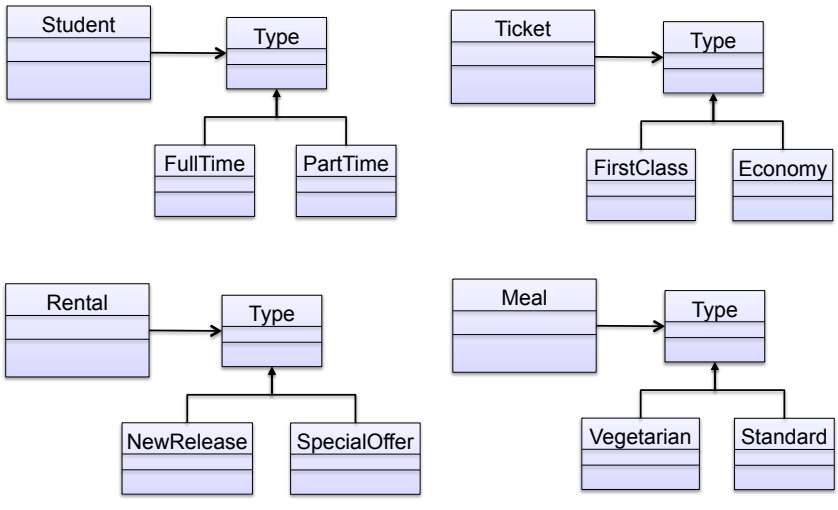
garth.gilmour@instil.co

The State Pattern

- State is a complex, infrequently used pattern
  - It applies to objects with complex internals
- The state pattern is used when:
  - You have a large complex class to maintain
  - The complexity is due to state specific behaviour:
    - The class has a number of possible states
    - You frequently need to switch based on the state
  - Conventional inheritance is not a good solution
    - Because the object can change its state whilst in use
- E.g. consider a 'Student' class:
  - Behaviour depends on whether he/she is full or part time
  - A student can change types during a semester

The State Pattern

- To use the state pattern:
  - Clearly define all the possible states of the original class
    - Perhaps by using UML state charts and interaction diagrams
  - Create a hierarchy of classes to model possible configurations
  - Move all state specific behaviour into the new classes
  - Add code to create an initial state object and use it
  - Add methods to change the type of the state object as the original class moves through its configurations
- The state pattern still uses inheritance and overriding
  - Except that the class hierarchy is moved 'sideways'
  - The client code is unaware that the hierarchy exists



The State Pattern

- There are several variants of the state pattern
  - Depending on where to place the most functionality
- The state object can expose query methods
  - Such as 'isConnected()' or 'isAllowed()'
  - The original object uses these to make decisions
  - This is the simplest implementation
- State objects can have lifecycle methods
  - Such as 'onEnteringState()' and 'onExitingState()'
    - A objects can be responsible for selecting the next state
  - The original object can pass itself to the state object
    - The state object changes the fields of the enclosing class
    - Making the state objects inner classes facilitates this

The Proxy Pattern

Creating Surrogate Objects

garth.gilmour@instil.co

The Proxy Pattern

- A proxy is a placeholder for a ‘real’ object
  - The client is frequently unaware it is using a proxy
- There are three common reasons for using a proxy
  - To restrict the clients access to the implementation
  - To transfer method calls to another machine (a Remote Proxy)
  - To defer creating the actual object (a Virtual Proxy)
- Proxy is frequently confused with Decorator
  - The difference is that a Decorator class layers on additional operations whereas a Proxy class manages access rights

Creating Proxies Using Reflection

- The Reflection library has built in support for proxies
  - A call to 'Proxy.newProxyInstance' creates a proxy object
    - This implements one or more interfaces provided by you
  - You specify a list of interfaces and a handler object
    - Plus the classloader via which you want the proxy loaded
- The generated class is public, final and extends 'Proxy'
  - You can test if a class is a proxy via 'Proxy.isProxyClass'
- Method calls are automatically passed on to the handler
  - It must implement the 'InvocationHandler' interface
  - The 'invoke' method is always the one called


```
public class Tester {
	public static void main(String[] args) {
		Object proxy = buildProxy();

		Maths maths = (Maths)proxy;
		System.out.println(maths.add(2,3));
		System.out.println(maths.multiply(2,3));
		System.out.println(maths.subtract(2,3));
		System.out.println(maths.divide(2,3));

		Finance finance = (Finance)proxy;
		System.out.println(finance.calculateVat(200));
		System.out.println(finance.calculateTax(200));
	}
	private static Object buildProxy() {
		ClassLoader loader = Maths.class.getClassLoader();
		Class[ ] interfaces = new Class[] {Maths.class,Finance.class};
		ProxyImpl implementation = new ProxyImpl();
		return Proxy.newProxyInstance(loader,interfaces,implementation);
	}
}
```

```
public interface Finance {
    double calculateVat(double amount);
    double calculateTax(double amount);
}
```

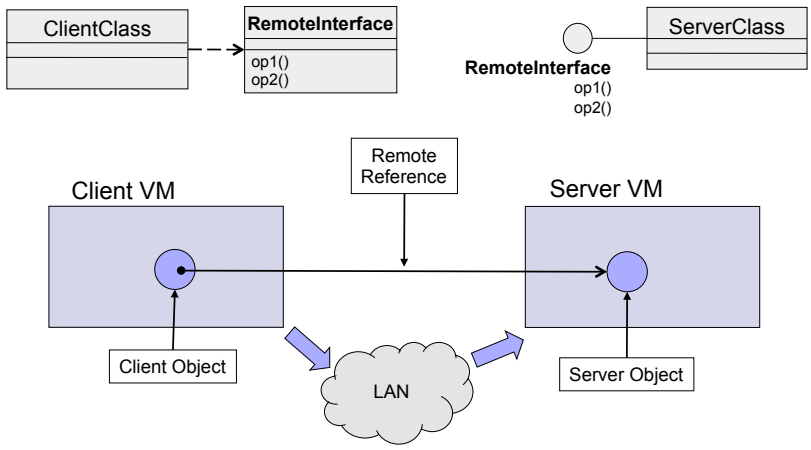
```
public interface Maths {
    public double add(double p1, double p2);
    public double subtract(double p1, double p2);
    public double multiply(double p1, double p2);
    public double divide(double p1, double p2);
}
```

```
public class ProxyImpl implements InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        String name = method.getName();
        if(method.getDeclaringClass() == Finance.class) {
            return processFinanceMethod(args, name);
        } else {
            return processMathMethod(args, name);
        }
    }
    private Object processMathMethod(Object[] args, String name) {
        // Implementation omitted
    }
    private Object processFinanceMethod(Object[] args, String name) {
        // Implementation omitted
    }
}
```

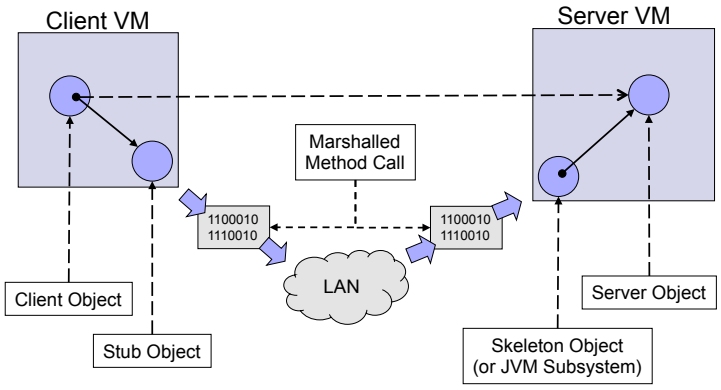

The Proxy Pattern in JEE

- Client side proxies are used to implement RMI
  - The ability to call methods in objects on remote VM's
  - A client-side proxy marshals requests to the server
- Server side proxies are added in Enterprise JavaBeans
  - A server-side proxy prevents a client directly using an EJB
  - This enables a container to implement services
    - Such as security checks and transaction management
- Proxies are used in several other places in JEE
  - 'Connection' and 'UserTransaction' objects are lightweight proxies that represent the resource whilst the client owns it
  - The JPA uses proxies to implement lazy loading

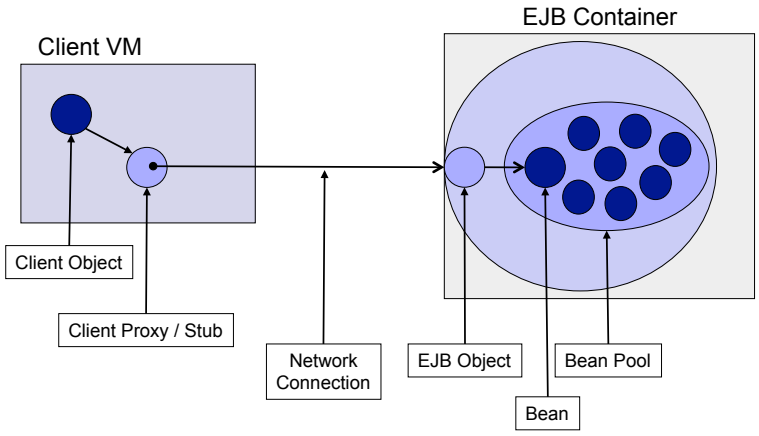
The Logical View of Java Remoting



The Architecture of Remoting



The Architecture of EJB's



Functional Patterns

Patterns Used Scala, F# etc…

garth.gilmour@instil.co

What is Functional Programming?

- FP is a separate style of programming
  - One which has been explored in academia for over 30 years and is now coming into the mainstream
  - The motivation for this is the advent of multi-core computing and the part FP played in Ruby's success
- The key characteristics of FP are:
  - Referential transparency
  - First class functions
  - Higher order functions
    - Currying and partial invocation

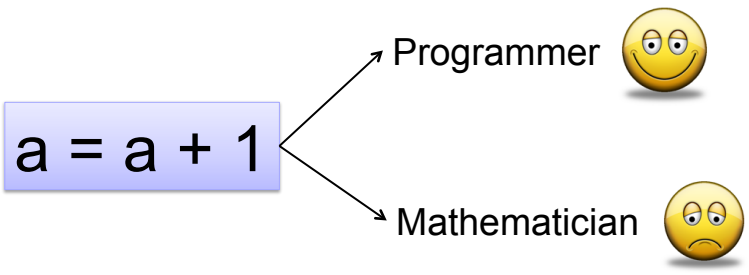
A Great Summary...

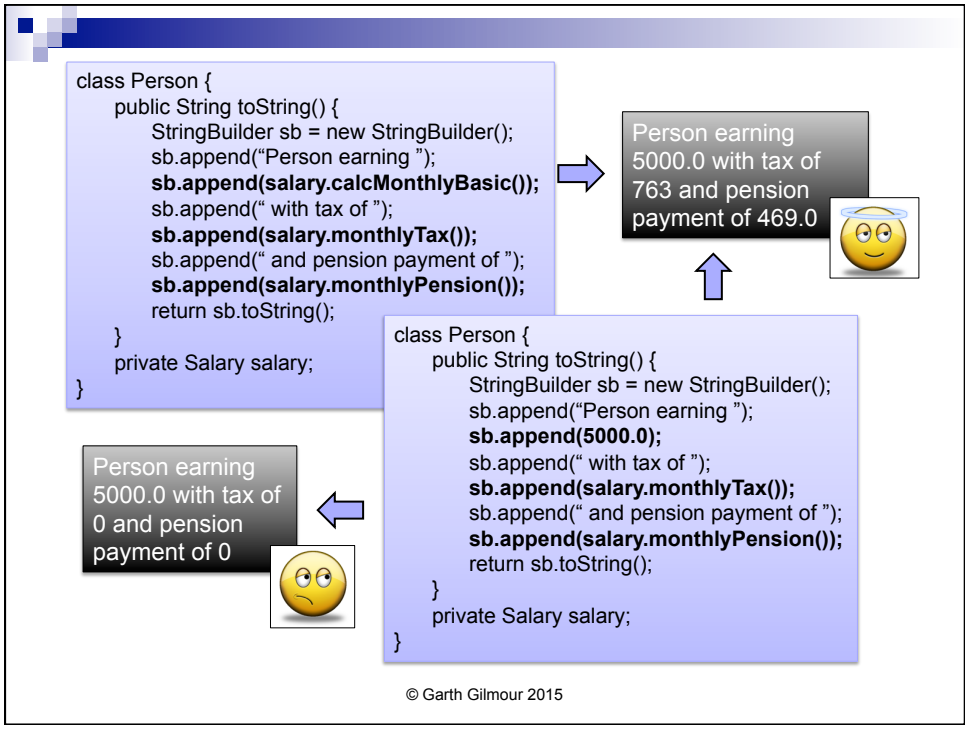
> OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.
>
> Michael Feathers

Referential Transparency

- In a purely functional language a function does nothing more than calculate outputs based on inputs
  - No matter how many times the function is called the same inputs always yield the same outputs
  - The function has no side effects that 'change the world'
- Such functions are said to be 'referentially transparent'
  - They make concurrent programming less daunting
- The application cannot be 'referentially transparent'
  - Otherwise it can do nothing but 'heat up the CPU' ☺
  - But individual components can and should be
    - An XSLT engine is one commonly used example

Referential Transparency

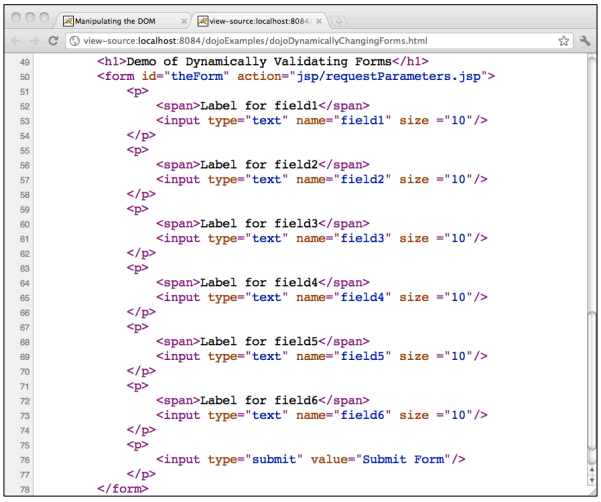




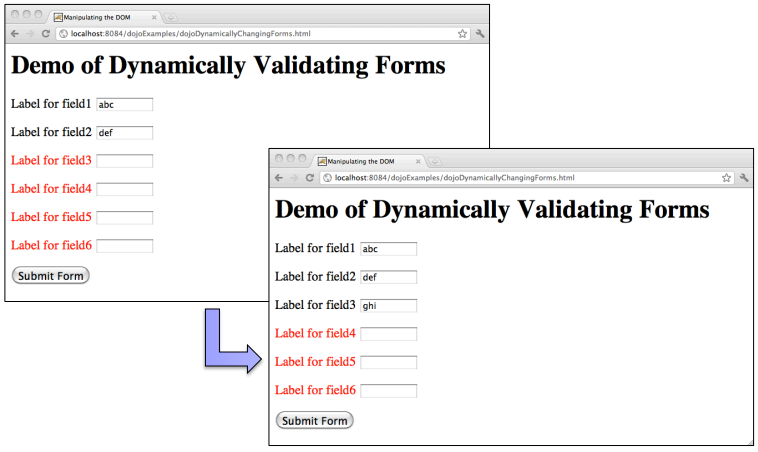
First Class Functions

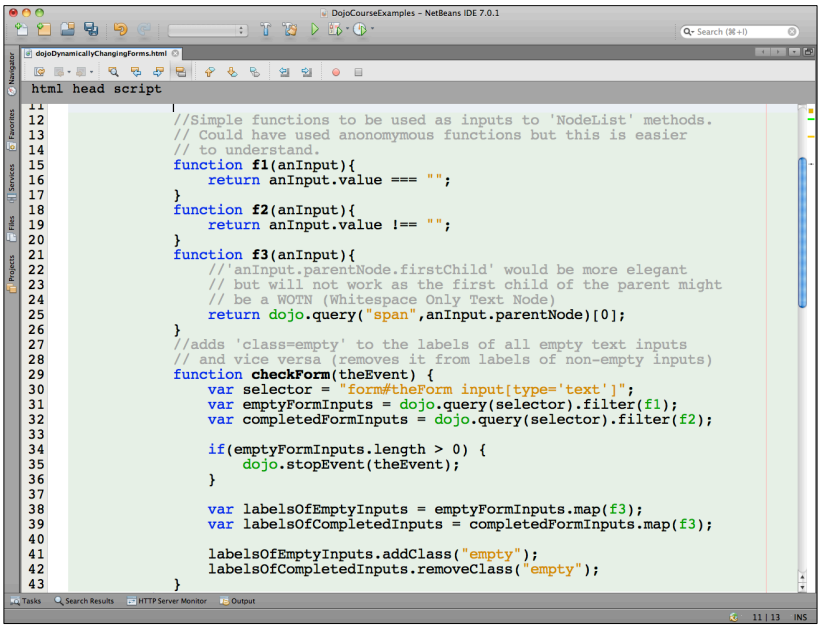
- Functions are the main building block in FP
  - They are linked together to accomplish useful tasks
- This means they need to be '1st class citizens'
  - Functions can be passed as parameters into other functions and built and returned as required
- Combining functions takes a while to get used to
  - Given that it is the opposite of how OO works
- One immediate benefit is 'internal iteration'
  - A collection can loop internally over its elements, with the supplied function applying a predicate or transformation

1st Class Functions in JavaScript



1st Class Functions in JavaScript






```
function f1(anInput){
    return anInput.value === "";
}
function f2(anInput){
    return anInput.value !== "";
}
function f3(anInput){
    return dojo.query("span",anInput.parentNode)[0];
}
function checkForm(theEvent) {
    var selector = "form#theForm input[type='text']";
    var emptyFormInputs = dojo.query(selector).filter(f1);
    var completedFormInputs = dojo.query(selector).filter(f2);

    if(emptyFormInputs.length > 0) {
        dojo.stopEvent(theEvent);
    }

    var labelsOfEmptyInputs = emptyFormInputs.map(f3);
    var labelsOfCompletedInputs = completedFormInputs.map(f3);

    labelsOfEmptyInputs.addClass("empty");
    labelsOfCompletedInputs.removeClass("empty");
}
```

First Class Functions and Lambdas

- Passing functions by name is convenient
  - But it is annoying to have to think up names for single line functions that are only going to be used once
    - Typically you will write dozens of these in user interfaces
  - They take up space and are a maintenance burden
- Single line functions can be replaced with lambdas
  - A lambda can be viewed as an anonymous function
    - Which can be inlined into the call to the main function
  - They have a simplified syntax to ease readability
    - In C# '(a,b) => { return a + b; }' is the basic syntax but this can be shortened to '(a,b) => a + b' for single statements
    - In Scala this can be further shortened to '_ + _'

Lambdas in C# (via LINQ)

```
void runQueryUsingLinqSyntax(IEnumerable<Employee> testData) {
   var results =  from e in testData
                  where e.Dept == "IT"
                  orderby e.Age
                  select new { e.Name, e.Age };

   Console.WriteLine("----- Query Via LINQ -----");
   foreach (var x in results) {
      Console.WriteLine("{0} of age {1}", x.Name, x.Age);
   }
}
```

Lambdas in C# (via LINQ)

```
void runQueryUsingLambdas(IEnumerable<Employee> testData) {
   var results = testData.Where(e => e.Dept == "IT")
                         .Select(e => new { e.Name, e.Age })
                         .OrderBy(e => e.Age);

   Console.WriteLine("----- Query Via Lambdas -----");
   foreach (var x in results) {
      Console.WriteLine("{0} of age {1}", x.Name, x.Age);
   }
}
```


Lambdas in Scala

- Scala lambdas have a similar syntax to C#
  - This being a parameter list, an arrow and a body
  - If the body contains multiple expressions it needs braces
- Lambdas generally represent jobs of work
  - They tend to be very short e.g. obj.foo((x) => bar(x))
- Scala lambdas have one special feature
  - The parameter list can be omitted in favor of underscores
  - The first '_' is replaced with the first parameter, the second replaces the second parameter and so on…
    - So 'obj.foo(bar(_, _, _))' instead of 'obj.foo((x,y,z) => bar(x,y,z))'
    - Once you get used to this it is a wonderful feature

Lambdas in Scala

```
object Program {
  def main(args : Array[String]) {
    println("---Demo 1---")
    val limit = 6
    (1 to limit).foreach(num => println("Scala Rocks! " + num))

    println("\n---Demo 2---")
    val data = List(10,11,12,13,14,15)
    val result = data.foldLeft("Data is:")((a,b) => a + " " + b)
    println(result)

    println("\n---Demo 3---")
    val text = "abc123def456ghi789"
    val regex = "[a-z]{3}".r
    val newText = regex.replaceAllIn(text, m => m.group(0).toUpperCase())
    println(newText)
  }
}
```

```
---Demo 1---
Scala Rocks! 1
Scala Rocks! 2
Scala Rocks! 3
Scala Rocks! 4
Scala Rocks! 5
Scala Rocks! 6

---Demo 2---
Data is: 10 11 12 13 14 15

---Demo 3---
ABC123DEF456GHI789
```


Lambdas in Scala

```
object PyramidTwo {
  def main(args : Array[String]) {
    println("Enter the height of the pyramid...")
    val heightString = readLine()
    val height = Integer.parseInt(heightString)

    (1 to height).foreach(i => println(buildRow(i, height)))
  }
  def buildRow(rowNum : Int, height : Int) = {
    val numSpaces = height - rowNum
    val numHashes = rowNum * 2 - 1

    (" " * numSpaces) + ("#" * numHashes)
  }
}
```

```
Enter the height of the pyramid...
6
     #
    ###
   #####
  #######
 #########
###########
```

Lambdas Plus Collections in Scala

- The most obvious use of lambdas is with collections
  - Lambdas as used to perform internal iteration
  - E.g. instead of manually taking items from a list in a 'while' we pass in a lambda and have the iteration done for us
    - Besides shortening our code this abstracts the mechanism that is used to traverse the collection, allowing it to be refined
- Common methods are 'foreach', 'filter' and 'reduceLeft'
  - 'reduceLeft' accepts a lambda which takes two parameters
  - The lambda is applied to the first two parameters, and then again to the result and the third parameter and so on…

Lambdas Plus Collections in Scala

```
val myList = List(1,2,3,4,5)
myList.reduceLeft((no1, no2) => no1 + no2)
myList.reduceLeft(_+_)
myList.reduceLeft((no1, no2) => no1 * no2)
myList.reduceLeft(_*_)
myList.foldLeft(8)((no1, no2) => no1 + no2)
myList.foldLeft(8)(_+_)
myList.foldLeft(10)((no1, no2) => no1 * no2)
myList.foldLeft(10)(_*_)
```

```
15
15
120
120
23
23
1200
1200
```

Concurrency and Scala Collections

- Scala has an actor-based library for concurrency
  - Each actor object has a mailbox into which messages can be delivered and then processed asynchronously
  - Actor objects are mounted onto threads from a pool
    - You should not need to care about the details of the mapping
- This library will be replaced by the Akka 2.0
  - Akka is an alternative Actor library which can be used from both Java and Scala and ships with the Typesafe Stack
  - Akka supports Software Transactional Memory (STM), distributed actors, fault tolerance and non-blocking I/O

Concurrency and Scala Collections

- Scala 2.9 introduced parallel collections
  - This provides a simple way of taking advantage of multi-core machines via the Java 7 fork-join framework
  - To switch to the parallel versions simply add ‘par’
    - E.g. ‘myList.par.foreach(x => func(x))’
  - The tasks you run in parallel need to be side effect free
    - Otherwise adding concurrency will result in ‘hiesenbugs’
- The SCALA research group at EPFL have won a five year European Research Grant on parallelism
  - The idea is to use Scala to create DSL’s that will take advantage of massively parallel systems (1000’s of cores)

```
class Result (val value : String, val threadID : Long) {}

object Program {
  def main(args : Array[String]) {
    val originalData = List("ab", "cd", "ef", "gh", "ij", "kl")

    val newData1 = originalData.map(
                          (str) => new Result(str + "yy", Thread.currentThread.getId))
    val newData2 = originalData.par.map(
                          (str) => new Result(str + "zz", Thread.currentThread.getId))

    for( r <- newData1) {
      printf("%s from thread %d\n", r.value, r.threadID)
    }
    println("----------------------------")

    for( r <- newData2) {
      printf("%s from thread %d\n", r.value, r.threadID)
    }
  }
}
```

```
abyy from thread 1
cdyy from thread 1
efyy from thread 1
ghyy from thread 1
ijyy from thread 1
klyy from thread 1
----------------------------
abzz from thread 12
ghzz from thread 11
cdzz from thread 12
ijzz from thread 11
efzz from thread 12
klzz from thread 11
```


Lambdas in Java 8

- Java was supposed to gain lambdas in version 7
  - Several proposals were submitted with prototypes
  - For various reasons none of them made it in
- They have finally appeared in Java 8
  - Which will hopefully simplify many popular API's

```
private void demoTemplatesAndCallbacks(HibernateTemplate template) {
    String query = "select delivery.course.title from Delivery delivery";
    List titles = template.executeFind(new QueryCallback(query));
    System.out.println("Courses being delivered are:");
    for(Object title : titles) {
        System.out.printf("\t%s\n",title);
    }
}
```

```
public class QueryCallback implements HibernateCallback {
    public QueryCallback(String query) {
        this.queryStr = query;
    }
    public Object doInHibernate(Session session)
                    throws HibernateException, SQLException {
        return session.createQuery(queryStr).list();
    }
    public String queryStr;
}
```

```
private void demoTemplatesAndCallbacks(HibernateTemplate template) {
    String query = "select delivery.course.title from Delivery delivery";
    List titles = template.executeFind((s) -> s.createQuery(query).list());

    System.out.println("Courses being delivered are:");
    titles.forEach((t) -> System.out.printf("\t%s\n",title));
}
```


```
SwingUtilities.invokeLater(new Runnable() {
                                public void run() {
                                    textField.setText(theResult);
                                }
                            });
```

```
SwingUtilities.invokeLater(() -> textField.setText(theResult));
```

Higher Order Functions

- Since functions are first class citizens they can be built within, and returned from, other functions
  - This enables two closely related FP techniques, known as 'currying' and 'partial invocation'
- Currying is when a function accepts 'n' parameters and returns a function that accepts 'n - 1'
  - The first parameters is stored in the returned function
- Partial invocation is when a function is invoked with some of its parameters missing
  - Scala functions can have multiple parameter lists, if one of these is omitted a function is returned that expects the rest


```
object CurryingDemoOne {
  def addNormally(no1: Int, no2: Int) = {
    no1 + no2
  }

  def addV1(no1: Int): (Int) => Int = {
    return (no2: Int) => no1 + no2
  }

  def addV2(no1: Int) = {
    (no2: Int) => no1 + no2
  }

  def addV3(no1: Int) = (no2: Int) => no1 + no2

  def addV4(no1: Int)(no2: Int) = no1 + no2
```

```
  //All the invocations below print 19
  def main(args: Array[String]) {
    println(addNormally(12, 7))
    println(addV1(12)(7))
    println(addV2(12)(7))
    println(addV3(12)(7))
    println(addV4(12)(7))

    val func1 = addV1(12)
    val func2 = addV2(12)
    val func3 = addV3(12)
    val func4 = addV4(12)_

    println(func1(7))
    println(func2(7))
    println(func3(7))
    println(func4(7))
  }
}
```

```
object CurryingDemoTwo  {
  def findMatchesV1(regex : String): (String) => (Array[String]) = {
     return (data : String) => {
       regex.r.findAllIn(data).toArray
     }
  }
  def findMatchesV2(regex : String) = {
     (data : String) => {
       regex.r.findAllIn(data).toArray
     }
  }
  def findMatchesV3(regex : String) = (data : String) => {
       regex.r.findAllIn(data).toArray
  }
  def findMatchesV4(regex : String) (data : String) = {
       regex.r.findAllIn(data).toArray
  }
```



```
def demoV1() {
  val data = "ab CD ef GH ij KL mn OP qr ST"
  val matchTwoUppercase = findMatchesV1("[A-Z]{2}")
  val matchTwoLowercase = findMatchesV1("[a-z]{2}")

  println("Version One Results:")
  for (m <- matchTwoUppercase(data)) {
    println("\t" + m)
  }
  println("-------------")
  for (m <- matchTwoLowercase(data)) {
    println("\t" + m)
  }
}
```

```
Version One Results:
	CD
	GH
	KL
	OP
	ST
-------------
	ab
	ef
	ij
	mn
	qr
```

```
def demoV2() {
  val data = "ab CD ef GH ij KL mn OP qr ST"
  val matchTwoUppercase = findMatchesV2("[A-Z]{2}")
  val matchTwoLowercase = findMatchesV2("[a-z]{2}")

  println("Version Two Results:")
  for (m <- matchTwoUppercase(data)) {
    println("\t" + m)
  }
  println("-------------")
  for (m <- matchTwoLowercase(data)) {
    println("\t" + m)
  }
}
```

```
Version Two Results:
        CD
        GH
        KL
        OP
        ST
-------------
        ab
        ef
        ij
        mn
        qr
```



```
def demoV3() {
  val data = "ab CD ef GH ij KL mn OP qr ST"
  val matchTwoUppercase = findMatchesV3("[A-Z]{2}")
  val matchTwoLowercase = findMatchesV3("[a-z]{2}")

  println("Version Three Results:")
  for (m <- matchTwoUppercase(data)) {
    println("\t" + m)
  }
  println("-------------")
  for (m <- matchTwoLowercase(data)) {
    println("\t" + m)
  }
}
```

```
Version Three Results:
        CD
        GH
        KL
        OP
        ST
-------------
        ab
        ef
        ij
        mn
        qr
```

```
def demoV4() {
  val data = "ab CD ef GH ij KL mn OP qr ST"
  val matchTwoUppercase = findMatchesV4("[A-Z]{2}")_
  val matchTwoLowercase = findMatchesV4("[a-z]{2}")_

  println("Version Four Results:")
  for (m <- matchTwoUppercase(data)) {
    println("\t" + m)
  }
  println("-------------")
  for (m <- matchTwoLowercase(data)) {
    println("\t" + m)
  }
}
```

```
Version Four Results:
	CD
	GH
	KL
	OP
	ST
-------------
	ab
	ef
	ij
	mn
	qr
```



```
object CurryingDemoThree {
  def joinUp(separator : Char)(data : Array[String]) = {
      val sb = new StringBuilder
      for(str <- data) {
        sb.append(str)
        sb.append(separator)
      }
      sb.substring(0,sb.length - 1)
  }

  def joinUpWithHashes = joinUp('#') _
  def joinUpWithHyphens = joinUp('-') _

  def main(args : Array[String]) {
      val testData = Array("abc","def","ghi","jkl","mno")
      println(joinUp(',')(testData))
      println(joinUpWithHashes(testData))
      println(joinUpWithHyphens(testData))
  }
}
```

```
abc,def,ghi,jkl,mno
abc#def#ghi#jkl#mno
abc-def-ghi-jkl-mno
```

The Monad Pattern

- Monadic Composition is a design pattern
  - Common to all functional programming languages
  - Since it is an FP pattern it wont make much sense until you are familiar with the FP programming toolkit
    - In the same way that OO patterns make no sense till you understand the limitations of the OO toolkit
- Monads are used with 'amplified types'
  - In Java amplified versions of type 'T' would include a 'List<T>', a 'Class<T>' and an 'EntityManager<T>'
  - The monad pattern enables amplified types to be created from values (and vice versa) automatically

Options And Monadic Composition

> Monads turn control flow into data flow, where it can be constrained by the type system.
>
> Oleg Kiselyov

Options and Monadic Composition

- The monad pattern can be used in any language
  - But it is only practical where the language has a built in syntax for performing monadic compositions
  - In Scala the 'for' expression can be used
- The only example of monads you will frequently encounter in normal Scala is the 'Option Monad'
  - 'Option' has two derived types, called 'Some' and 'None'
  - Instead of returning 'null' a method should:
    - Be declared as returning 'Option[T]'
    - When data is available return a 'Some[T]'
    - When data is unavailable return 'None'

Handling Errors Without Option

```
void printPostcode(Person person) {
    Address address = person.findAddress();
    if(address != null) {
        Postcode postcode = address.findPostcode()
        if(postcode != null) {
            String code = postcode.findValue();
            if(code != null) {
                System.out.println(code);
            }
        }
    }
}
```

Handling Errors With Option

```
def printPostcodeIfExists(person : Person) {
    println("Working with " + person.name)
    for (
        place   <- person findAddress;
        code    <- place findPostcode;
        result  <- code findValue
       ) println(result)
}
```

The Scala Cake Pattern

- The Cake Pattern is specific to Scala
  - It enables you to perform a kind of dependency injection
  - The DI is implemented at the compiler level
    - You cannot rewire the application by altering an external configuration file but you can specify combinations of components in your code and switch between them
- To implement the pattern we:
  - Place multiple implementations of services inside traits, along with an abstract field of their common type
  - Declare an object which mixes-in the traits and provides a definition for each of the abstract fields

Applying The Cake Pattern

```
trait PricingEngine {
  def price(productId: String, quantity: Int): Double
}

trait PaymentsEngine {
  def makePayment(cardNumber: String, amount: Double): Boolean
}

trait StockCheckEngine {
  def quantityInStock(productId: String): Int
}
```


Applying The Cake Pattern

```
trait PricingEngineComponent {
  val pricingEngine: PricingEngine

  class PricingEngineStub extends PricingEngine {
    def price(productId: String, quantity: Int) = {
      println("Stub pricing engine called")
      4.50
    }
  }
  class PricingEngineReal extends PricingEngine {
    def price(productId: String, quantity: Int) = {
      println("Real pricing engine called")
      5.60
    }
  }
}
```

Applying The Cake Pattern

```
trait PaymentsEngineComponent {
  val paymentsEngine: PaymentsEngine

  class PaymentsEngineStub extends PaymentsEngine {
    def makePayment(cardNumber: String, amount: Double) = {
      println("Stub payments engine called")
      true
    }
  }
  class PaymentsEngineReal extends PaymentsEngine {
    def makePayment(cardNumber: String, amount: Double) = {
      println("Real payments engine called")
      true
    }
  }
}
```


Applying The Cake Pattern

```
trait StockCheckEngineComponent {
  val stockCheckEngine: StockCheckEngine

  class StockCheckEngineStub extends StockCheckEngine {
    def quantityInStock(productId: String) = {
      println("Stub stock check engine called")
      12
    }
  }
  class StockCheckEngineReal extends StockCheckEngine {
    def quantityInStock(productId: String) = {
      println("Real stock check engine called")
      24
    }
  }
}
```

Applying The Cake Pattern

```
trait ShopComponent {
  this: PricingEngineComponent with PaymentsEngineComponent
        with StockCheckEngineComponent =>

  val shop = new Shop

  class Shop {
    def makePurchase(productId: String, quantity: Int, cardNumber: String): Boolean = {
      if (stockCheckEngine.quantityInStock(productId) >= quantity) {
        val price = pricingEngine.price(productId, quantity)
        paymentsEngine.makePayment(cardNumber, price)
      }
      false
    }
  }
}
```


Applying The Cake Pattern

```
object UnitTestComponentRepository extends ShopComponent
                                      with PricingEngineComponent
                                      with PaymentsEngineComponent
                                      with StockCheckEngineComponent {
  val pricingEngine = new PricingEngineStub
  val paymentsEngine = new PaymentsEngineStub
  val stockCheckEngine = new StockCheckEngineStub
}
object IntegrationTestComponentRepository extends ShopComponent
                                             with PricingEngineComponent
                                             with PaymentsEngineComponent
                                             with StockCheckEngineComponent {
  val pricingEngine = new PricingEngineReal
  val paymentsEngine = new PaymentsEngineReal
  val stockCheckEngine = new StockCheckEngineStub
}
```

Applying The Cake Pattern

```
object ProductionComponentRepository extends ShopComponent
                                        with PricingEngineComponent
                                        with PaymentsEngineComponent
                                        with StockCheckEngineComponent {
  val pricingEngine = new PricingEngineReal
  val paymentsEngine = new PaymentsEngineReal
  val stockCheckEngine = new StockCheckEngineReal
}
```


Applying The Cake Pattern

```
object Program {
  def main(args : Array[String]) {
    println("----- Making Purchase Using Unit Test Repository -----")
    val shopWithStubs = UnitTestComponentRepository.shop
    shopWithStubs.makePurchase("AB12", 8, "XYZ987FGR6")

    println("----- Making Purchase Using Integration Test Repository -----")
    val shopWithSomeStubs = IntegrationTestComponentRepository.shop
    shopWithSomeStubs.makePurchase("AB12", 8, "XYZ987FGR6")

    println("----- Making Purchase Using Production Repository -----")
    val shopWithImpls = ProductionComponentRepository.shop
    shopWithImpls.makePurchase("AB12", 8, "XYZ987FGR6")
  }
}
```

Applying The Cake Pattern

```
----- Making Purchase Using Unit Test Repository -----
Stub stock check engine called
Stub pricing engine called
Stub payments engine called
----- Making Purchase Using Integration Test Repository -----
Stub stock check engine called
Real pricing engine called
Real payments engine called
----- Making Purchase Using Production Repository -----
Real stock check engine called
Real pricing engine called
Real payments engine called
```


Appendix One

A Reminder of UML Syntax

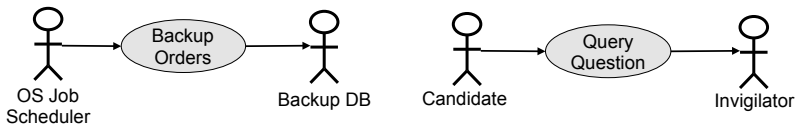
garth.gilmour@instil.co

Unified Modeling Language (UML)

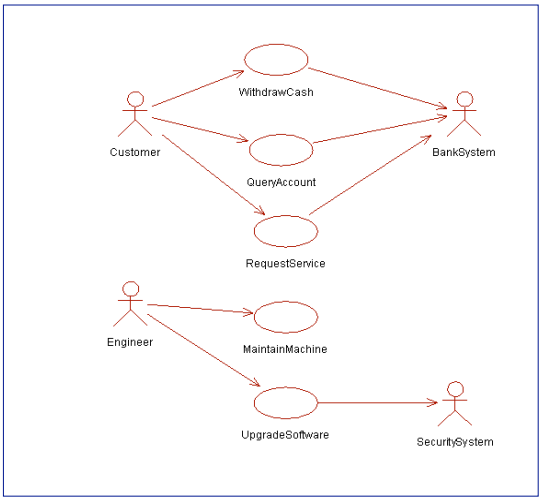
- UML is the industry standard way of visually describing the different aspects of an object-oriented design
  - Development tools increasingly allow round trip engineering between UML diagrams and code
- Patterns are best visualized using UML diagrams
  - With class and method names based on the abstract roles
- The UML is made up of nine diagrams
  - Each describing the system from a different perspective
  - In practice three diagrams predominate
    - Use Case Diagrams (actors and requirements)
    - Sequence Diagrams (interactions between objects)
    - Class Diagrams (classes and their relationships)

Use Case Diagrams

- Normally a Use Case:
  - Is initiated by a human actor (Customer, Employee etc…)
  - Makes use of external system actors (Web Services etc…)
- Other combinations are possible
  - The scenario could be started by an external system
  - Human actors could be consulted as the scenario progresses



Use Case Diagram (Requirements)

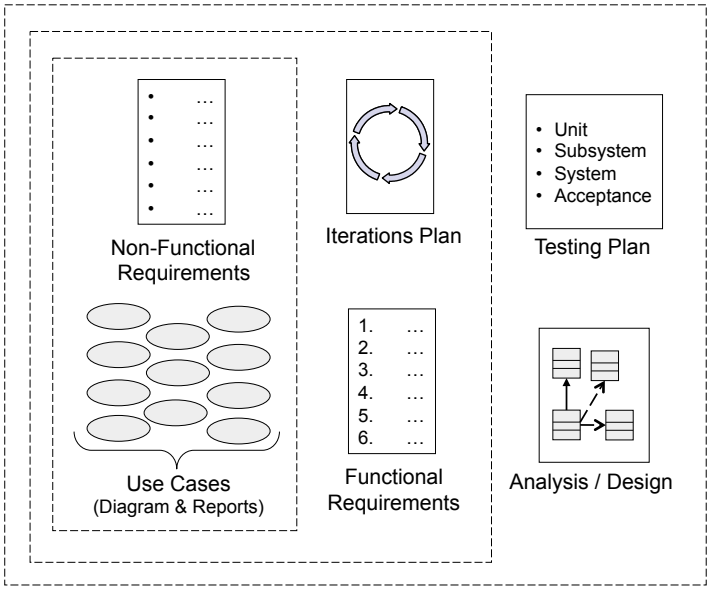


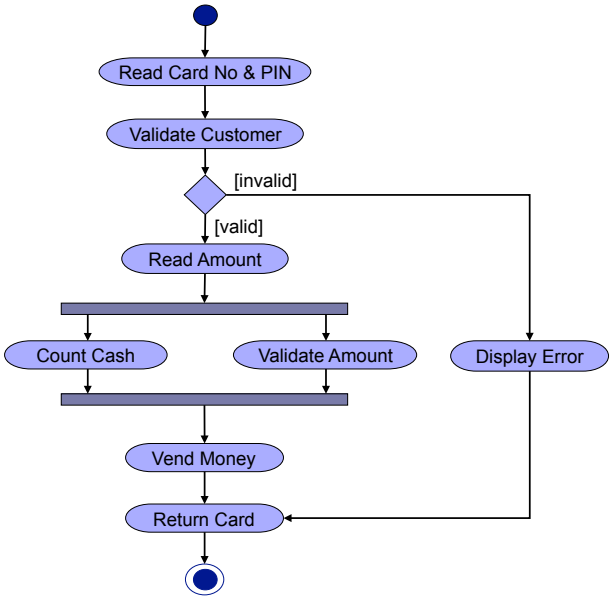
Withdraw Cash Use Case Report

Preconditions:
The ATM is running and has cash available

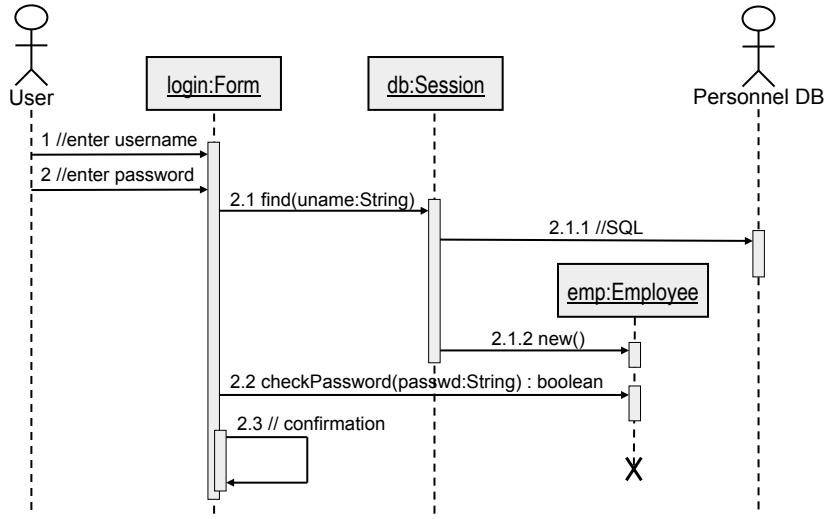
Postconditions:
The ATM has dispensed cash, the users balance has been reduced

Flows Of Events:
Basic Flow (The Happy Path)
1) User enters their card and pin number
2) System verifies the users identity and displays withdrawal amounts
3) User selects an amount to withdraw
4) System verifies amount and returns card
5) User removes card
6) System dispenses cash
7) User removes cash
Alternative Flows
User specifies the amount to withdraw
User has to re-enter password
Exceptional flows
Card jams in reader
Bank System unavailable

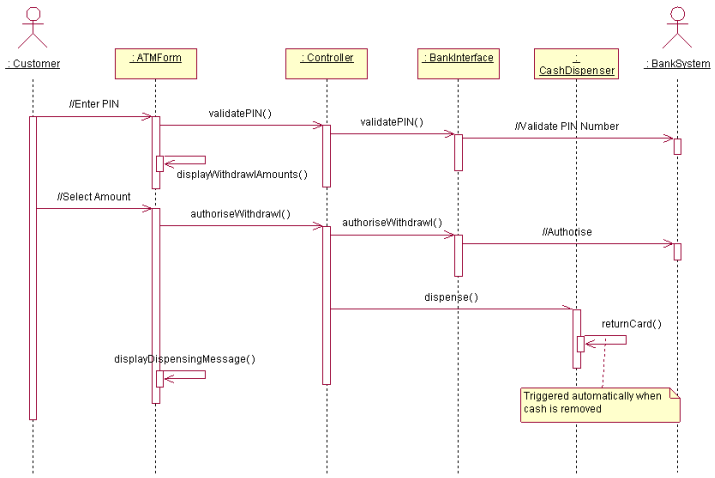




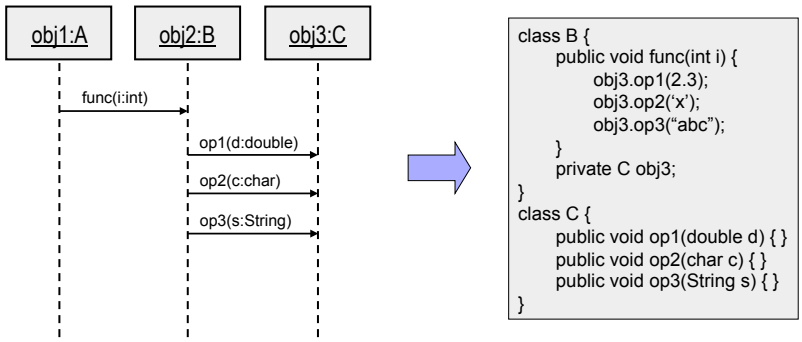
Sequence Diagrams (Design View)



Sequence Diagrams (Design View)

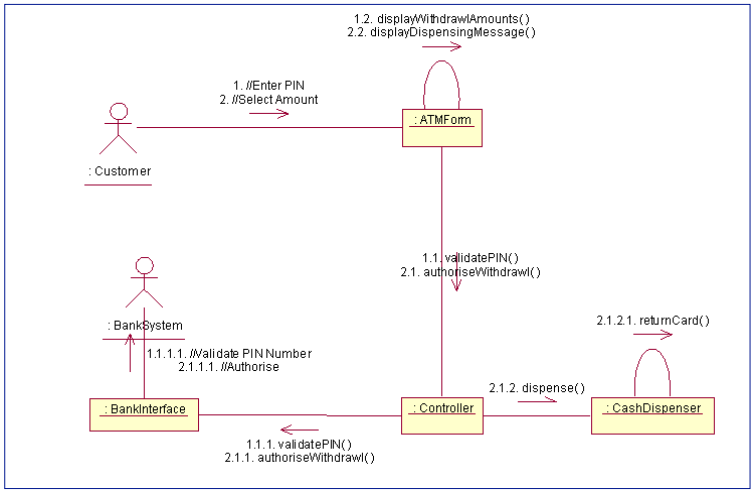


Sequence Diagrams (Design View)




```
class B {
      public void func(int i) {
            obj3.op1(2.3);
            obj3.op2('x');
            obj3.op3("abc");
      }
      private C obj3;
}
class C {
      public void op1(double d) { }
      public void op2(char c) { }
      public void op3(String s) { }
}
```

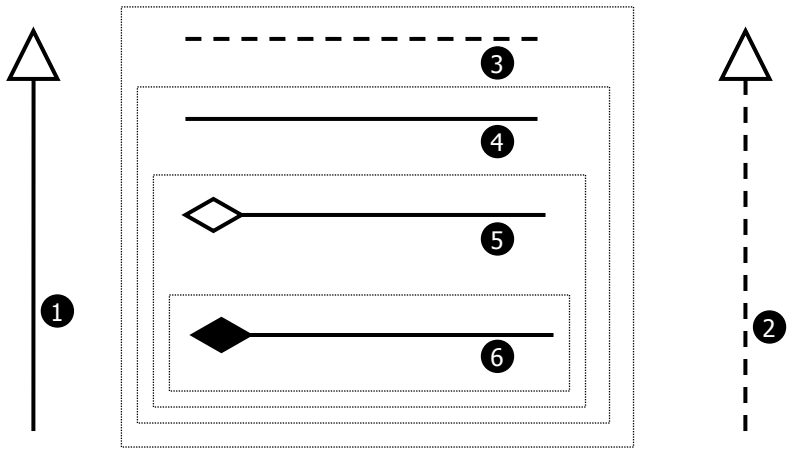

Collaboration Diagrams (Coupling)



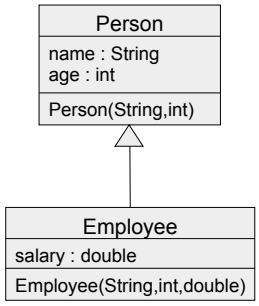
Structural Relationships and UML

- UML defines 6 kinds of relationships
  - These mainly exist between classes but also between packages, actors and use cases
  - Note that not all OO programming languages offer each of these relationships and every language uses them differently
- The relationships are shown on the next slide:
  1. Generalization
  2. Realization
  3. Dependency
  4. Association
  5. Aggregation
  6. Composition

Structural Relationships and UML



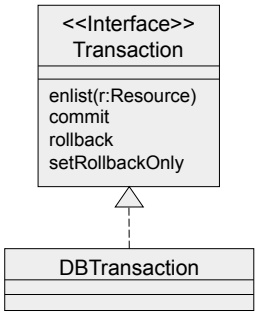
Generalization In Classes

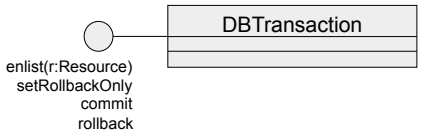



```
public class Person {
   private String name;
   private int age;
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

```
public class Employee extends Person {
   private double salary;
   public Employee(String name, int age, double salary) {
      super(name,age);
      this.salary = salary;
   }
}
```


Realization

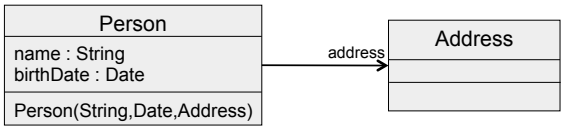





```
public interface Transaction {
   public void enlist(Resource r);
   public void commit();
   public void rolback();
   public void setRollbackOnly();
}
```

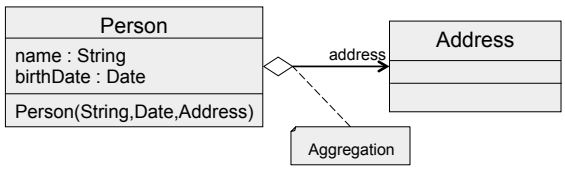
```
public class DBTransaction
                implements Transaction {
   // Code omitted
}
```

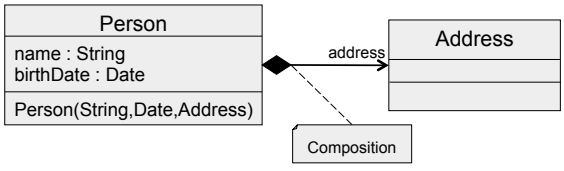
Association



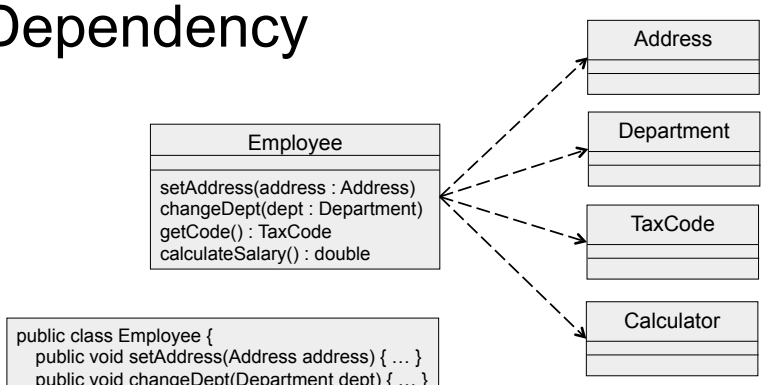

```
public class Person {
   private String name;
   private Date birthDate;
   private Address address;
   public Person(String name, Date birthDate,
                 Address address) {
      this.name = name;
      this.birthDate = birthDate;
      this.address = address;
   }
}
```


Aggregation and Composition



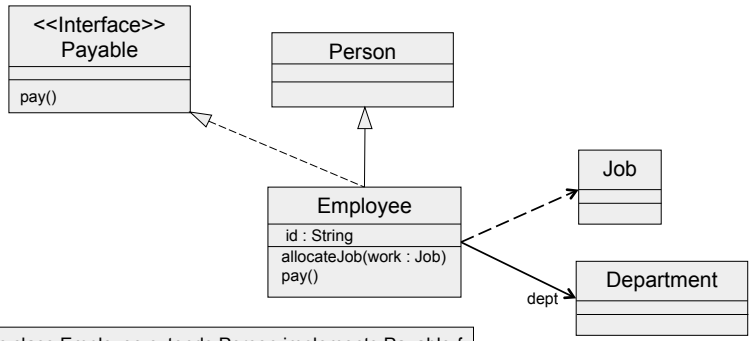


Dependency




```
public class Employee {
   public void setAddress(Address address) { … }
   public void changeDept(Department dept) { … }
   public TaxCode getCode() { … }
   public double calculateSalary() {
      Calculator calc = new Calculator();
      …
   }
}
```

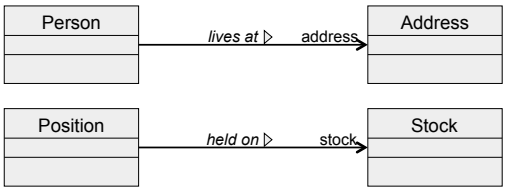

Relationships Summary



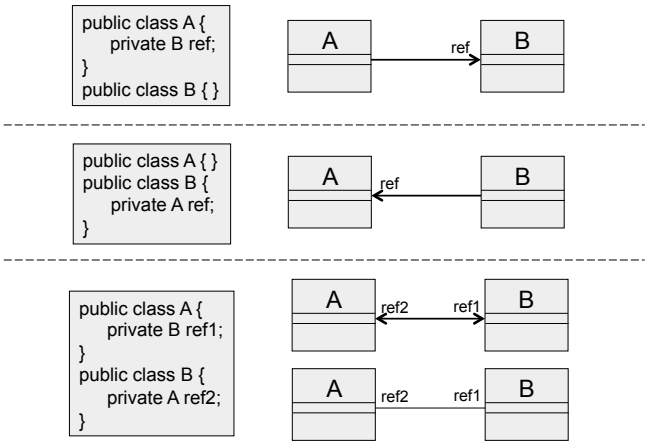

```
public class Employee extends Person implements Payable {
   private Department dept;
   private String id;
   public void allocateJob(Job work) { … }
   public void pay() { … }
}
```

Naming Relationships

- Relationships can be named
  - This has no effect on the generated code but greatly increases the readability of the diagram
- An arrowhead indicates how the name should be read
  - For straightforward relationships this is not required
  - But for industry specific terms it is very important

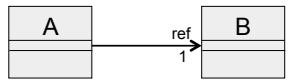


Navigating Relationships and Roles

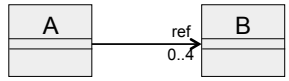


Navigation and Multiplicity

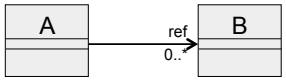
```
public class A {
    private B ref;
}
public class B { }
```



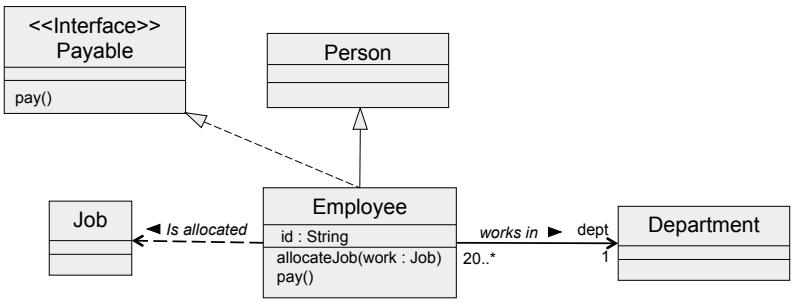

```
public class A {
    private B [] ref = new B[4];
}
public class B { }
```




```
public class A {
    //Holds B objects
    private List ref = new ArrayList();
}
public class B { }
```



Detailed Relationships Summary




```
public class Employee extends Person implements Payable {
   private Department dept;
   private String id;
   public void allocateJob(Job work) { … }
   public void pay() { … }
}
```

Appendix Two

Patterns Specific to JEE

garth.gilmour@instil.co

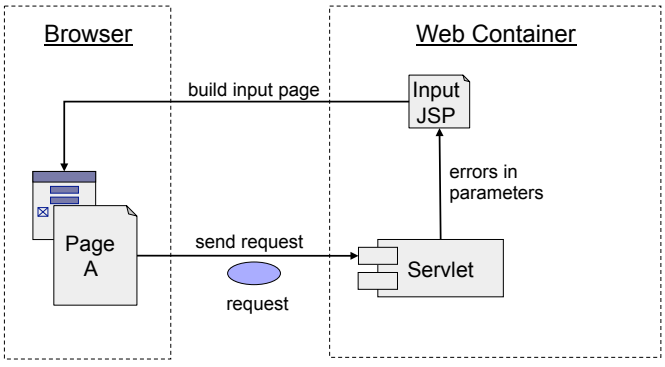
Limitations of the Core JEE Types

- The Web Container provides a limited set of components
  - Servlets, JSP, Filters, Listeners, Tag Libraries etc…
  - All of these are concerned with presentation
- If you want to build an application entirely within this Container you need to create extra components
  - Controller components to hold business logic
  - Value objects to hold typed and validated input data
- In effect you need to create your own mini-architecture
  - Most open source JEE web frameworks evolved in this way
  - Today there are many different frameworks to choose between

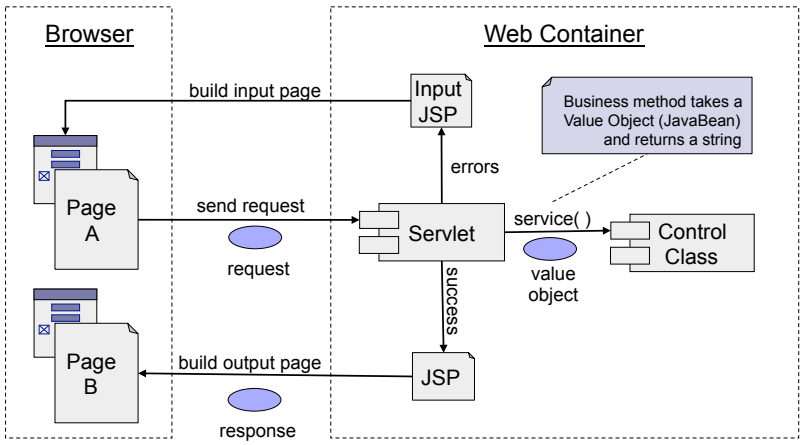
The Model 2 Web App Architecture

- Sun has defined a 'Model 2 Architecture'
  - Which describes how Servlets and JSP's can best be combined
  - Most JEE Web Frameworks are based on this architecture
- In a Model 2 based Web Application:
  - A Servlet receives the request and generates a JavaBean
    - Validating and converting the parameters as it does so
  - The JavaBean is passed to a controller class
    - This could be an EJB or a plain Java object (POJO)
  - The controllers output is passed with the JavaBean to a JSP
    - This takes care of generating the response for the client
    - There may be one JSP for each type of controller response
    - The original JSP may be called if the page contained errors

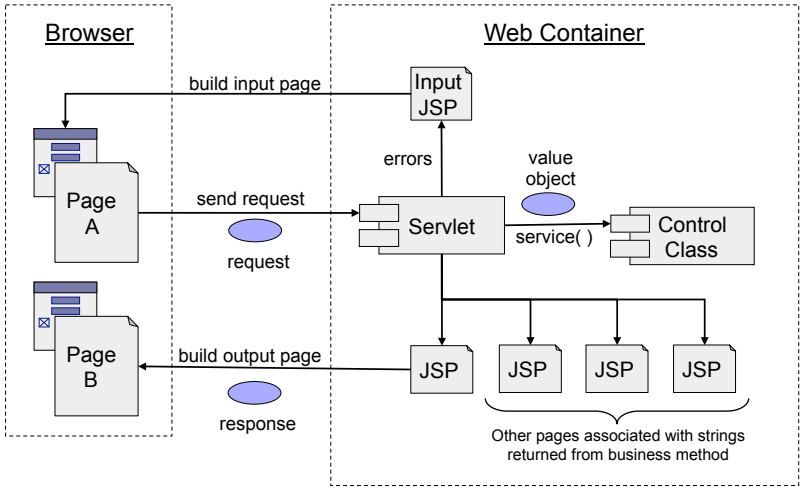
Model 2 Design Part 1



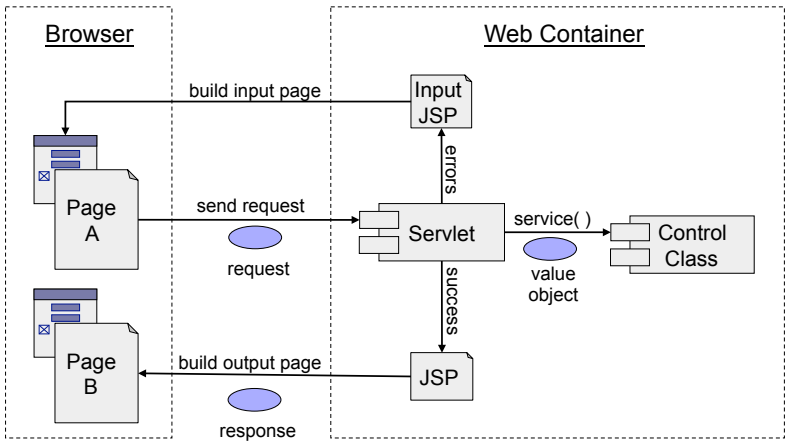
Model 2 Design Part 2



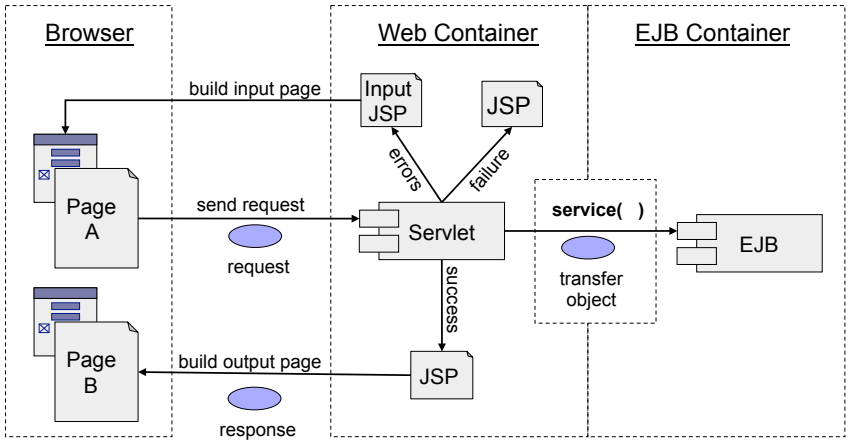
Model 2 Design Part 3



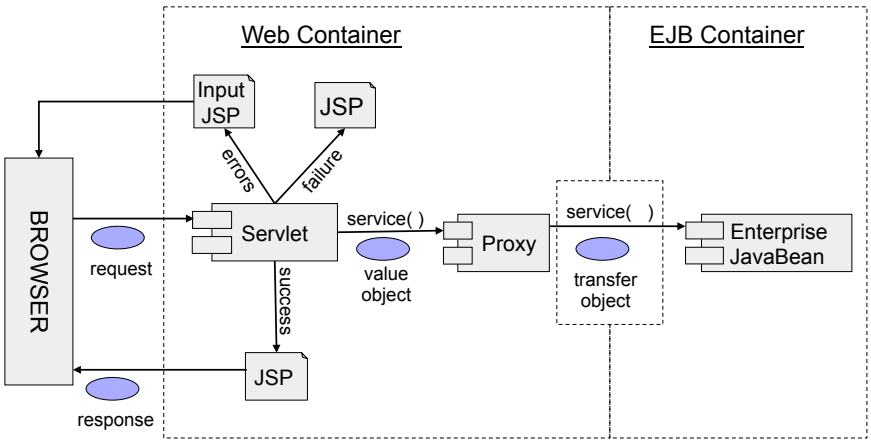
Model 2 Design (POJO Controller)



Model 2 Design (EJB as Controller)



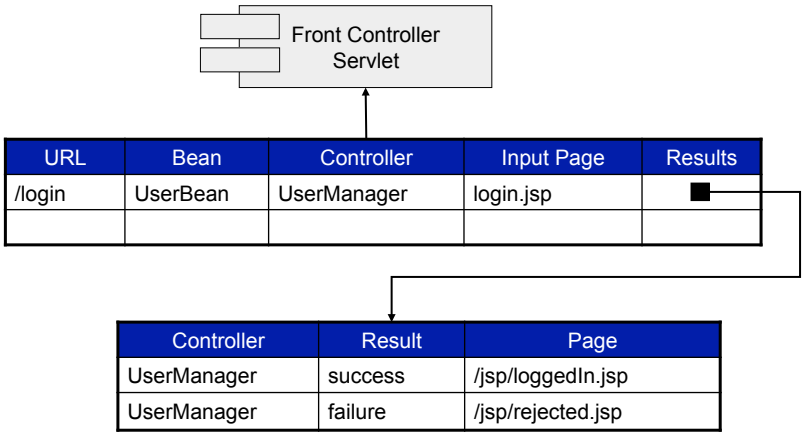
Modified Design (Dual Controllers)



The Front Controller Pattern

- Following Model 2 manually quickly becomes tedious
  - You are continually writing Servlets to route requests
  - For each request all that changes is:
    - The names of the parameters in the request
    - The name of the JavaBean to be created
    - The name of the controller to pass the bean to
    - The names of the JSP's that build the response
- However in reality only a single Servlet is needed
  - For each request this 'Master Servlet' reads the name of the JavaBean, Controller etc… from its own special configuration file
- This is known as a 'Front Controller' Servlet
  - All web frameworks are built around one

The Front Controller Pattern



URL	Bean	Controller	Input Page	Results
/login	UserBean	UserManager	login.jsp	■

Controller	Result	Page
UserManager	success	/jsp/loggedIn.jsp
UserManager	failure	/jsp/rejected.jsp


```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is the configuration file for an imaginary Model2 (MVC) framework -->
<FrontControllerConfig>
    <ValueObjectList> <!-- Value Objects -->
        <ValueObject name="flightVO" class="flights.FlightSearchValueObject"/>
        <ValueObject name="bookingVO" class="flights.BookingValueObject"/>
    </ValueObjectList>
    <ControllerList> <!-- Controllers -->
        <Controller name="flightC" class="flights.FlightSearchController"/>
        <Controller name="bookingC" class="flights.BookingController"/>
    </ControllerList>
    <UrlMappingsList> <!-- Url Mappings -->
        <UrlMapping url="/do/travel" controller="flightC" valueObject="flightVO"
                    valueObjectScope="session" inputPage="/jsp/travelform.jsp"/>
        <UrlMapping url="/do/booking" controller="bookingC" valueObject="bookingVO"
                    valueObjectScope="session" inputPage="/jsp/flightselection.jsp"/>
    </UrlMappingsList>
    <ResultsMappingsList> <!-- Results Mappings -->
        <ResultMapping url="/do/travel" result="success" page="/jsp/searchOK.jsp">
        <ResultMapping url="/do/travel" result="fail" page="/jsp/searchFailed.jsp">
    </ResultsMappingsList>
</FrontControllerConfig>
```

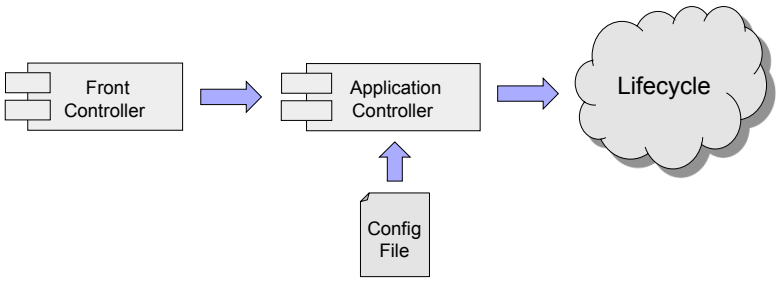
The Front Controller Pattern

- Adopting a 'Front Controller' Servlet requires some extra modifications to your design
  - The Servlet cannot hold the code to extract parameters by name from the request and perform the necessary validations
    - Instead it obtains a list of all the parameter names and uses reflection to call all the matching 'setXXX' methods
  - The JavaBeans must validate their own data
    - So each bean needs to have a 'validate' method
  - Controllers must have the same business method
    - Usually they share a base class or interface
  - The business method returns a link to the output JSP
    - Usually this is a string that is mapped to a URL in the XML file

The Application Controller Pattern

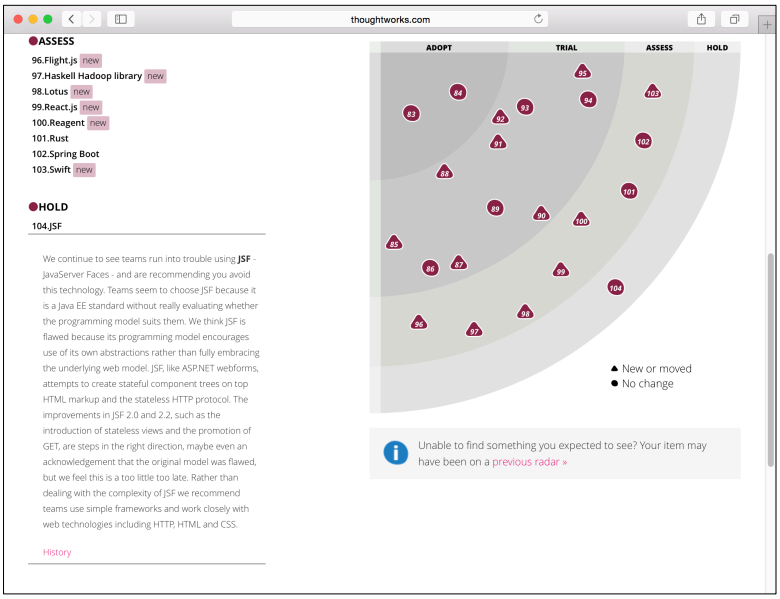
- The 'Application Controller' pattern is a commonly used modification to the 'Front Controller' pattern
  - Instead of the Servlet doing all the work it delegates all functionality to a normal Java class (the Application Controller)
  - The Front Controller acts as a shell
    - It passes requests directly to the Application Controller
    - This reads from the XML configuration file and routes the request from JavaBean to Controller to output JSP
- Frameworks usually adopt this pattern for extensibility
  - Users of the framework can replace the controller with their own
  - Usually the controller provides a non-final method for each stage of the lifecycle, so you can inherit and override as required

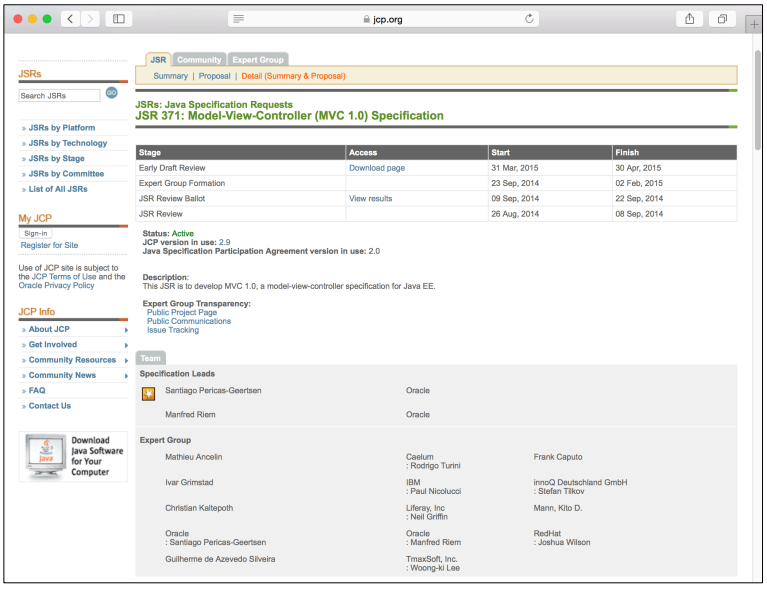
The Application Controller Pattern



The Upcoming Framework in JEE8

- There are **many** JEE MVC frameworks
  - Struts, Spring MVC and have been the most used
  - Dozens of others were written over the years
    - Including in-house ones that were never publicised
- Sun / Oracle never formally endorsed any of them
  - Instead choosing to back the JSF standard, which was based on the idea of server side web controls
  - This has never proved popular, and is virtually redundant given the improvements in modern browsers
- An official JEE MVC spec has finally been proposed
  - This will hopefully become part of JEE8





The Data Transfer Object Pattern

- In a multi-tier architecture there will be remote calls
  - Originally this would be from a client to an EJB
  - Today it would be to a Web Service (SOAPy or RESTful)
- Remote calls are expensive and risky
  - Risky because networking is inherently unpredictable
  - Expensive because of the (un)marshalling required
- Hence we want a small number of calls
  - With a (relatively) large amount of data sent in each call
  - This is the reverse of the best practises within the JVM

The Data Transfer Object Pattern

- A Data Transfer Object is used in remote calls
  - It contains a large number of fields that aggregate data
  - Unlike model or entity objects it has no behaviour
  - Fields are accessed via JavaBean properties
- Transfer and value objects are commonly confused
  - The term ‘value object’ originally referred to an object with a single field representing a date, direction, payment etc…
  - In JEE parlance it often refers to a JavaBean used to transfer data between components **in the same tier**

Appendix 3

Functional Features in Java 8

garth.gilmour@instil.co

Functional Features of Java 8

- In version eight Java becomes a functional language
  - This is essential to enable parallel programming
  - It is also desirable to keep pace with language evolution
    - The case for FP in industry has been proved by Ruby and Scala
    - Functional API's are now the norm (e.g. in 'Big Data')
- The key features to know are:
  - Support for 'method handles' in the language
  - The introduction of a syntax for lambda expressions
  - The 'Optional' type for eliminating null pointer errors
  - Support for the key functional methods in collections
  - Extension of the collections API to support parallelism

Introducing Method References

- Java 7 introduced the idea of 'Method Handles'
  - This provides a way to refer to an individual method, constructor or field and invoke / access it
  - Along with *invokedynamic* method handles enable scripting languages to run efficiently on the JVM
- Method references provide the same functionality within the core syntax of the Java language
  - You can store a link to a method or pass it to another
- You can link to a method reference via an interface
  - We will examine the different types of interface later

Types of Method Reference

- There are four types of method reference
  - All are expressed using the syntax 'foo::bar'
  - Where 'foo' is a class or object and 'bar' a member

Name / Description	Example
Reference to a static method	Math::random
Reference to an instance method (without specifying the object)	StringBuilder::append
Reference to an isntance method (specifying the object)	StringBuilder sb = new StringBuilder() sb::append
Reference to a constructor	StringBuilder::new

```
private static void demoStaticMethodRefs() {
        Supplier<Double> ref1 = Math::random;
        Supplier<Thread> ref2 = Thread::currentThread;
        DoubleUnaryOperator ref3 = Math::sqrt;
        UnaryOperator<String> ref4 = System::getProperty;

        System.out.println(ref1.get());
        System.out.println(ref2.get().getId());
        System.out.println(ref3.applyAsDouble(16.0));
        System.out.println(ref4.apply("java.vm.vendor"));
}
```

```
0.8673006807854204
1
4.0
Oracle Corporation
```



```
private static void demoInstanceMethodRefs () throws Exception {
        StringBuilder builder = new StringBuilder();
        InetAddress address = InetAddress.getByName("localhost");
        File currentDir = new File(".");

        Function<String,StringBuilder> ref1 = builder::append;
        Supplier<String> ref2 = address::getCanonicalHostName;
        Supplier<String> ref3 = currentDir::getAbsolutePath;
        Consumer<Object> out = System.out::println;

        out.accept(ref1.apply("def"));
        out.accept(ref2.get());
        out.accept(ref3.get());
}
```

```
def
localhost
/Users/ggilmour/Development/Java/JavaEightDemos/.
```

```
private static void demoConstructorRefs() throws Exception {
    Function<String,File> ref1 = File::new;
    Function<URI,File> ref2 = File::new;

    File f1 = ref1.apply(".");
    File f2 = ref2.apply(new URI("file:///bin"));

    System.out.println(f1.getAbsolutePath());
    System.out.println(f2.getAbsolutePath());
}
```

```
/Users/ggilmour/Development/Java/JavaEightDemos/./bin
```

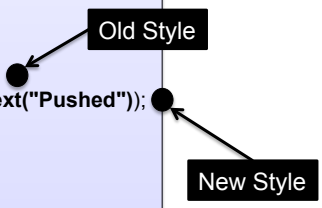

Introducing Lambdas

- A lambda is a self-contained block of code that can be stored, passed as a parameter or returned
  - The terms ‘lambda’, ‘closure’ and ‘code block’ are closely related but have slightly different meanings
  - But are used interchangeably by many developers
- Lambdas are not an Object Oriented concept
  - In OO methods (‘verbs’) never leave the objects (‘nouns’)
    - See the famous ‘Kingdom on Nouns’ blog post
  - The ideal of lambdas comes from Functional Programming
    - Ruby was the first language to popularize them in industry
    - Although anonymous functions in JavaScript are closely related

Lambdas and the Listener Pattern

- OO has a problem with event-driven coding
  - The standard convention of attaching handler functions to events does not fit into the object model
- The Listener Pattern tries to reconcile OO and events
  - But causes code bloat when you create an (anonymous) inner class in order to override a single callback method
  - C# went a different route by introducing delegates
- Lambdas are a more natural fit than listeners
  - Before: 'bt.addActionListener(new ActionListener() { … })'
  - After: 'bt.addActionListener(e -> println(e.getWhen()))'

```
public class MyGui extends JFrame {
   private class MyListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         textField.setText("Pushed");
      }
   }
   public MyGui() {
      super("My First Swing GUI");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      textField = new JTextField(10);
      button = new JButton("Push Me");

      button.addActionListener(new MyListener());
      button.addActionListener(e -> textField.setText("Pushed"));

      setLayout(new FlowLayout());
      add(button);
      add(textField);
   }
   private JButton button;
   private JTextField textField;
}
```



The Road Not Taken…

The newest version of the Microsoft Visual J++ development environment supports a language construct called *delegates* or *bound method references*…

It is unlikely that the Java programming language will ever include this construct. Sun already carefully considered adopting it in 1996, to the extent of building and discarding working prototypes. Our conclusion was that bound method references are unnecessary and detrimental to the language…

We believe bound method references are *unnecessary* because another design alternative, *inner classes*, provides equal or superior functionality. In particular, inner classes fully support the requirements of user-interface event handling, and have been used to implement a user-interface API at least as comprehensive as the Windows Foundation Classes.

We believe bound method references are *harmful* because they detract from the simplicity of the Java programming language and the pervasively object-oriented character of the APIs….

Extracts From: About Microsoft's "Delegates"
Whitepaper by the Java language team at JavaSoft

Lambdas and Internal Iteration

- The second use of lambdas is as 'jobs of work'
  - When a method generates a container we can use a lambda to specify how items should be processed
- This 'internal iteration' shortens your code
  - As long as there are one or more functional methods available that can match up against your intent
  - When this is not the case use an explicit loop
- Internal iteration has one massive benefit
  - A library can introduce parallelism 'under the hood'
  - Users of he library don't need to understand threading
    - But they must ensure their lambdas are side-effect free

The Syntax of Lambdas

- The full syntax of a Lambda is:
  - Zero or more parameter names in parenthesis
    - The types of the parameters are inferred from usage
  - The arrow '->' symbol
    - Note this is not the '=>' (aka rocket) used in C# / Scala
  - One or more lines of code within braces
- The following shortcuts can (and should) be used
  - A single parameter does not need parenthesis
  - A single statement body does not need braces
    - The return from the statement is returned from the lambda

```
@FunctionalInterface
public interface Converter<T,U> {
    public U convert(T input);
}
```

```
public class Program {
    public static void main(String [] args) {
        Supplier<String> ref1 = () -> "Scooby";

        Consumer<String> ref2 = s -> System.out.println(s);

        Function<String,Integer> ref3 = s -> s.length();

        IntToDoubleFunction ref4 = num -> num * 1.0;

        Predicate<String> ref5 = s -> s.length() == 5;

        UnaryOperator<String> ref6 = s -> s + "wobble";

        BinaryOperator<String> ref7 = (s1,s2) -> s1 + s2;

        Converter<String,Double> ref8 = s -> Double.parseDouble(s);
```



```
        //Never write multi-line lambdas!
        Consumer<String> ref9 = (s) -> {
                                    System.out.println(ref1.get());
                                    ref2.accept(s);
                                    System.out.println(ref3.apply(s));
                                    System.out.println(ref4.applyAsDouble(123));
                                    System.out.println(ref5.test(s));
                                    System.out.println(ref6.apply(s));
                                    System.out.println(ref7.apply(s, "wobble"));
                                    System.out.println(ref8.convert("1234.567"));
                                };
        ref9.accept("wibble");
    }
}
```

```
Scooby
wibble
6
123.0
false
wibblewobble
wibblewobble
1234.567
```

References to Lambdas

- There is no single type for 'reference to lambda'
  - C# does this via a series of generic 'Func' types
  - Java cannot do this because generics are not reified
- Instead 'Functional Interfaces' are used
  - A functional interface declares a single abstract method
  - Any other methods must have default implementations
  - The '@FunctionalInterface' annotation tells the compiler you intend the current interface to be used for this purpose
- Over forty functional interfaces are provided for you
  - They all reside within the package 'java.util.function'

Functional Interfaces

- Most of the functional interfaces have default methods
  - E.g. 'Consumer' has 'andThen' as well as 'accept'
- Extra versions of the interfaces exist for primitive types
  - E.g. 'IntConsumer', 'LongConsumer' & 'DoubleConsumer'

Name	Description
Consumer<T>	Accepts a T and performs operation(s) on it
Supplier<T>	Generates a T
Predicate<T>	Accepts a T and returns a boolean
UnaryOperator<T>	Both takes and returns a T
Function<T,U>	Accepts a T and returns a U
BiFunction<T, U, V>	Takes a T and a U and returns a V

Best Practices for Lambdas

- Never write multi-line lambdas
  - The resulting code is hard to read tortuous to debug
- Avoid nesting lambdas
  - Although this can lead to very concise code it can quickly become too dense for others to understand
- Prefer method references to lambdas
  - It is easier to read 'obj::func' than 'x -> obj.func(x)'
  - Refactor a multi-line lambda by:
    - Extracting the code into a private helper method
    - Referring to the extracted method via a reference

```
'myList.stream().forEach(x -> System.out.println(x))'
'myList.stream().forEach(System.out::println)'
```


Using Lambdas as Return Values

- Lambdas can be used as return types
  - This allows you to create functions that return functions
  - A hard concept to get your head round at first…
- Consider wrapping text within HTML tags…
  - You write functions like 'wrapInH1(*text*)' for every tag
    - But this would take time and produce 'code bloat'
  - You could write a single function 'wrapIn(*tagName,text*)'
    - But this would be harder to read and force the user to enter the tag name on ever call, inevitably resulting in silly mistakes
  - The best solution is to write a function that build a function
    - So a call to 'build("H1")' returns a wrapper function for headers
    - The generated function could itself take a function as input

```
public class Program {
    private static UnaryOperator<String> buildWrapperV1(String tagName) {
        final String openingTag = String.format("<%s>",tagName);
        final String closingTag = String.format("</%s>",tagName);

        return s -> openingTag + s + closingTag;
    }
    public static void main(String [] args) {
        UnaryOperator<String> wrapInH1 = buildWrapperV1("h1");
        UnaryOperator<String> wrapInH2 = buildWrapperV1("h2");
        UnaryOperator<String> wrapInEm = buildWrapperV1("em");

        System.out.println(wrapInH1.apply("foo"));
        System.out.println(wrapInH2.apply("bar"));
        System.out.println(wrapInEm.apply("zed"));
    }
}
```

```
<h1>foo</h1>
<h2>bar</h2>
<em>zed</em>
```



```
public class Program {
    private static String getVendorName() {
        return System.getProperty("java.vendor");
    }
    private static String getOSName() {
        return System.getProperty("os.name");
    }
    private static String getInstallDir() {
        return System.getProperty("java.home");
    }
    private static void printDivider() {
        System.out.println("----------------");
    }
    private static Function<Supplier<String>,String> buildWrapperV2(String tagName) {
        final String openingTag = String.format("<%s>",tagName);
        final String closingTag = String.format("</%s>",tagName);

        return f -> openingTag + f.get() + closingTag;
    }
```

```
    public static void main(String [] args) {
        Function<Supplier<String>,String> wrapInH1 = buildWrapperV2("h1");
        Function<Supplier<String>,String> wrapInH2 = buildWrapperV2("h2");
        Function<Supplier<String>,String> wrapInEm = buildWrapperV2("em");

        System.out.println(wrapInH1.apply(() -> System.getProperty("java.vendor")));
        System.out.println(wrapInH2.apply(() -> System.getProperty("os.name")));
        System.out.println(wrapInEm.apply(() -> System.getProperty("java.home")));
        printDivider();
        System.out.println(wrapInH1.apply(Program::getVendorName));
        System.out.println(wrapInH2.apply(Program::getOSName));
        System.out.println(wrapInEm.apply(Program::getInstallDir));
    }
}
```

```
<h1>Oracle Corporation</h1>
<h2>Mac OS X</h2>
<em>/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/jre</em>
----------------
<h1>Oracle Corporation</h1>
<h2>Mac OS X</h2>
<em>/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/jre</em>
```


Introducing Optional

- 'Optional' prevents null pointer exceptions
  - Optional is an abstract base class with two derived types
    - By convention these are called 'Some' and 'None'
  - Think of 'Some' as a set of 1 and 'None' as a set of 0
- Instead of returning 'null' we return an 'Optional<T>'
  - The caller does not check the type of the returned object
  - Instead they use methods like 'orElse' to specify the value to be use if the desired value could not be found
  - Doing this consistently will remove memory errors
- Optional is one example of a Monad
  - Sadly Java 8 does not support Monadic Composition

```
public class Program {
   private static Optional<String> fetchSystemProperty(String name) {
      String result = System.getProperty(name);
      if(result == null) {
         return Optional.empty();
      } else {
         return Optional.of(result);
      }
   }
   public static void main(String [] args) {
      Optional<String> result1 = fetchSystemProperty("java.version");
      Optional<String> result2 = fetchSystemProperty("wibble");

      result1.ifPresent(s -> System.out.println(s));
      result2.ifPresent(s -> System.out.println(s));
      result1.ifPresent(System.out::println);
      result2.ifPresent(System.out::println);
      System.out.println(result1.orElse("No such property!"));
      System.out.println(result2.orElse("No such property!"));
   }
}
```

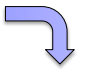
```
1.8.0
1.8.0
1.8.0
No such property!
```


The Functional Toolkit

- Coders in functional languages expect libraries to make good use of the features we have seen
  - Most importantly the data structures should be able to accomplish most tasks using internal iteration
  - Method refs or lambdas specifying the work to be done
- This can be thought of as the ‘functional toolkit’
  - It is provided in C# by the LINQ technology
  - It is slowly being baked into JavaScript
    - But available today through libraries like ‘Underscore.js’
- The toolkit is added to the collection types in Java 8
  - But you have to build a ‘stream’ from the array or collection

Streams and Interoperability

```
public class Program {
   public static void main(String [] args) {
      int [] myarray = {10,25,40,55,70,85};

      DoubleStream ds = Arrays.stream(myarray).mapToDouble(i -> i * 2.0);
      ds.forEach(d -> System.out.println(d));

      System.out.println("----");

      Stream<String> ss = Arrays.stream(myarray).mapToObj(String::valueOf);
      ss.filter(s -> s.endsWith("5")).forEach(System.out::println);
   }
}
```



```
20.0
50.0
80.0
110.0
140.0
170.0
----
25
55
85
```


Using the Functional Toolkit

- Using the FP toolkit is a lot like the UNIX shell
  - Each method performs a single specific task
  - You chain the methods to produce the effect you need
- The most commonly used methods are:
  - **For Each** - to perform an operation on each item
  - **Filter** - to remove items that don't meet a predicate
  - **Map** - to apply a transformation to a list (e.g. int to string)
  - **Flat Map** - to transform a list and join up the results
  - **Reduce** - to produce a single value from the list content

```
public class Program {
   public static void main(String [] args) {
      List<String> data = Arrays.asList("ab","cde","fg","hij","kl","mno","pq","rst","uv","wxyz");

      demoInternalIteration(data);
      demoTesting(data);
      demoFiltering(data);
      demoMapping(data);
      demoFlatMapping(data);
      demoReducing(data);
   }
   private static void demoInternalIteration(List<String> data) {
      System.out.printf("Internal iteration produced:\n\t");
      data.stream().forEach(s -> System.out.printf("%s ",s));
      System.out.println();
   }
   private static void demoFiltering(List<String> data) {
      Stream<String> results =  data.stream().filter(s -> s.length() == 3);
      printResults("Filtering produced:", results);
   }
```



```
   private static void demoMapping(List<String> data) {
      Stream<Integer> results1 = data.stream().map(s -> s.length());
      printResults("Mapping produced:", results1);

      IntStream results2 = data.stream().mapToInt(s -> s.length());
      printResults("Mapping to int produced:", results2);

      LongStream results3 = data.stream().mapToLong(s -> s.length());
      printResults("Mapping to long produced:", results3);

      DoubleStream results4 = data.stream().mapToDouble(s -> s.length() * 1.0);
      printResults("Mapping to double produced:", results4);
   }

   private static void demoFlatMapping(List<String> data) {
      Stream<Character> results = data.stream().flatMap(Utils::toStream);
      Utils.printResults("Flat mapping produced:", results);
   }
```

```
private static void demoReducing(List<String> data) {
    Optional<String> result1 = data.stream()
                                    .reduce((s1,s2) -> s1 + "-" + s2);

    System.out.printf("First reduction produced:\n\t%s\n", result1.orElse("nothing"));

    StringBuilder result2 = data.stream()
                                .reduce(new StringBuilder(),
                                        (sb,s) -> sb.insert(0,s),
                                        (sb1,sb2) -> sb1);
    System.out.printf("Second reduction produced:\n\t%s\n", result2);

    int result3 = data.stream()
                      .reduce(0,
                              (total,s) -> total + s.codePointAt(0),
                              (a,b) -> a + b );
    System.out.printf("Third reduction produced:\n\t%s\n", result3);
}
```



```
private static void demoTesting(List<String> data) {
    System.out.println("Testing produced:");
    System.out.println("\t" + data.stream().findAny().orElse("Empty!"));
    if(data.stream().allMatch(s -> s.length() > 1)) {
        System.out.println("\tAll strings longer than 1");
    }
    if(data.stream().anyMatch(s -> s.length() > 3)) {
        System.out.println("\tAt least one string longer than 3");
    }
    if(data.stream().noneMatch(s -> s.length() > 4)) {
        System.out.println("\tNo strings longer than 4");
    }
  }
}
```

```
Results from mappings:
        10 Arcatia Road London 11 Arcatia Road London 12 Arcatia Road London
20 University Street Belfast 21 University Street Belfast 22 University Street Belfast
Results of first grouping:
        In IT: Dave Mary
        In Sales: Fred Evie
        In HR: Jane Pete
Results of second grouping:
        Band 1: Dave Fred Pete
        Band 2: Jane Mary Evie
Generated stream content:
        0.10970352229879887 0.8627696120950432 0.6503816300966332
0.9255841284326823 0.39599056685464395
```


Running Operations in Parallel

- The main motivation for adding functional support to Java was to enable parallel programming
  - Most developers do not have an in depth knowledge of the memory model, threading API and underlying hardware
  - Yet the ‘multicore apocalypse’ makes the ability to scale tasks over multiple cores an essential coding skill
- One example of the is the parallel collections
  - Adding a call to ‘parallel()’ in any of the preceding examples causes the task to be spread over multiple cores
  - The stream is divided into sub-streams, which are then assigned to threads via the fork-join framework

Running Operations in Parallel

```
public class Program {
    public static void main(String [] args) {
        List<Integer> input = Arrays.asList(30,26,22,18,14,10);
        input.stream()
                .parallel()
                .map(Program::waitAndReturn)
                .forEach(System.out::println);
    }
    private static String waitAndReturn(Integer sleep) {
        goToSleep(sleep);
        return buildMessage(sleep);
    }
```


Running Operations in Parallel

```
    private static void goToSleep(Integer sleep) {
        try {
            Thread.sleep(sleep * 1000);
        } catch (InterruptedException ex) {
            System.out.println("BANG! " + ex.getMessage());
        }
    }
    private static String buildMessage(Integer sleep) {
        String msg = "Thread %d just returned after waiting %d secs";
        long threadId = Thread.currentThread().getId();

        return String.format(msg, threadId, sleep);
    }
}
```

Running Operations in Parallel

```
Thread 11 just returned after waiting 10 secs
Thread 13 just returned after waiting 14 secs
Thread 1 just returned after waiting 18 secs
Thread 14 just returned after waiting 22 secs
Thread 10 just returned after waiting 26 secs
Thread 12 just returned after waiting 30 secs
```


Running Operations in Parallel

```
public class Program {
    public static void main(String [] args) {
        List<Integer> input = Arrays.asList(30,26,22,18,14,10);
        input.stream()
                    .parallel()
                    .map(Program::waitAndReturn)
                    .forEach(System.out::println);
    }
```

```
Thread 1 just returned after waiting 30 secs
Thread 1 just returned after waiting 26 secs
Thread 1 just returned after waiting 22 secs
Thread 1 just returned after waiting 18 secs
Thread 1 just returned after waiting 14 secs
Thread 1 just returned after waiting 10 secs
```