301AA - Advanced Programming [AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Assistant: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-11: Frameworks and Inversion of Control
Frameworks and Inversion of Control

- Recap: JavaBeans as Components
- Frameworks, Component Frameworks and their features
- Frameworks vs IDEs
- Inversion of Control and Containers
- Frameworks vs Libraries
- Decoupling Components
- Dependency Injection
- IoC Containers in Spring
Components: a recap

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party. Clemens Szyperski, ECOOP 1996

• Example: *Java Beans*
• *Contractually specified interfaces*: events, methods and properties
• *Explicit context dependencies*: serializable, constructor with no argument
• *Subject to composition*: connection to other beans
  – Using connection oriented programming (event source and listeners)
Towards Component Frameworks

- **Software Framework**: A collection of *common code* providing *generic functionality* that can be *selectively overridden or specialized* by user code providing *specific functionality*

- **Application Framework**: A software framework used to implement the *standard* structure of an application for a *specific* development environment

- **Web (Application) Framework**: as above, for Web Applications
  - **ASP.NET** by Microsoft for web sites, web applications and web services
  - **GWT** - Google Web Toolkit (GWT)
  - **Rails** - Written in Ruby - Provides default structures for databases, web services and web pages.
  - **Spring** - for Java-based enterprise web applications
Examples of Frameworks

Web Application Frameworks

GUI Toolkits
Examples of Frameworks

• General software frameworks
  – **.NET** – Windows platform. Provides language interoperability
  – **Android SDK** – Supports development of apps in Java (but does not use a JVM!)
  – **Cocoa** – Apple’s native OO API for macOS. Includes C standard library and the Objective-C runtime.
  – **Eclipse** – Cross-platform, easily extensible IDE with plugins
Examples of Frameworks

• Frameworks for Application with GUI
  – **Gnome** – Written in C; mainly for Linux
  – **Qt** - Cross-platform; written in C++
Examples of Frameworks

• Web Application Frameworks [based on Model-View-Controller design pattern]
  – **ASP.NET** by Microsoft for web sites, web applications and web services
  – **GWT** - Google Web Toolkit (GWT)
  – **Rails** - Written in Ruby - Provides default structures for databases, web services and web pages.
Examples of Frameworks

• Concurrency
  – **Hadoop Map/Reduce** - software framework for applications which process big amounts of data in parallel on large clusters (thousands of nodes) in a fault-tolerant manner.

    • **Map**: Takes input data and converts it into a set of tuples (key/value pairs).
    • **Reduce**: Takes the output from Map and combines the data tuples into a smaller set of tuples.
Features of Frameworks

• A framework embodies some abstract design, with more behavior built in.

• In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes.

• The framework’s code then calls your code at these points.

• A very general concept, emphasizing inversion of control: as opposed to libraries is the code of the framework that calls the code.
Component Frameworks

- Frameworks that support development, deployment, composition and execution of components designed according to a given Component Model.
- Support the development of individual components, enforcing the design of precise interfaces.
- Support the composition/connection of components according to the mechanisms provided by the Component Model.
- Allows instances of these components to be “plugged” into the component framework itself.
- Provide prebuilt functionalities, such as useful components or automated assembly functions that automatically instantiate and compose components to perform common tasks.
- The component framework establishes environmental conditions for the component instances and regulates the interaction between component instances.
Frameworks vs IDEs

- Orthogonal concepts
- A framework can be supported by several IDEs
  - Eg: Spring supported by Spring Tool Suite (based on Eclipse), NetBeans, IntelliJ IDEA, Eclipse, ...
- An IDE can support several frameworks
  - Eg: NetBeans supports JavaBeans, Spring, J2EE, Maven, Hibernate, JavaServer Faces, Struts, Qt, ...
- Interesting issue: compositional approach to frameworks
Frameworks Features

• Consist of **parts** that are found in many apps of that type
  – **Libraries** with APIs (classes with methods etc.)
  – Ready-made extensible programs (**"engines"**) 
  – Sometimes also **tools** (e.g. for development, configuration, content)

• Frameworks, like software libraries, provide **reusable abstractions** of code wrapped in a well-defined API

• But: **Inversion of control**
  – unlike in libraries, the overall program's flow of control is not dictated by the caller, but by the framework

• Helps solving recurring design problems
  – Providing a default behavior
  – Dictating how to fill-in-the-blanks

• Non-modifiable framework code
  – Extensibility: usually by selective overriding
Extensibility

• All frameworks can be extended to cater for app-specific functionality.
  – A framework is intended to be extended to meet the needs of a particular application
• Common ways to extend a framework:
  – Extension is carried out by sub-classing, overriding methods, and implementing interfaces
  – Plug-ins: framework can load certain extra code in a specific format
  – Within the framework language:
    • Subclassing & overriding methods
    • Implementing interfaces
    • Registering event handlers
Two selected topics

We give a closer look to two general topics related to frameworks:

• Inversion of control
• Mastering dependencies among components
Inversion of Control (IoC) in GUIs

- In text-based interaction, the order of interactions and of invocations is decided by the code.
- In the GUI-based interaction, the GUI loop decides when to invoke the methods, based on the order of events.
- Also known as the Hollywood Principle – “Don't call us, we'll call you”.

https://martinfowler.com/bliki/InversionOfControl.html
Inversion of Control in Frameworks

• With Frameworks the **Inversion of Control** becomes dominant

• The application architecture is often fixed, even if customizable, and determined by the Framework
  – When using a framework, one usually just implements a few callback functions or specializes a few classes, and then invokes a single method or procedure.
  – The framework does the rest of the work for you, invoking any necessary client callbacks or methods at the appropriate time and place.

• Example: Java's Swing and AWT classes, NetBeans projects
  – They have a huge amount of code to manage the user interface, and there is inversion of control because you start the GUI framework and then wait for it to call your listeners
Inversion of Control

Traditional Program Execution

The app has control over the execution flow, calling library code when it needs to.

Inversion of Control

The framework has control over the execution flow, calling app code for app-specific behavior.
Frameworks vs Libraries

- Frameworks consist of large sets of classes / interfaces, suitably packaged
- Not much different from libraries
- (Possible) Key feature: wide use of Inversion of Control
- “Framework” sometimes intended as “well-designed library”
- “Java Collection Framework” vs “Standard Template Library”: are them frameworks or libraries?
Java Collection Framework

Standard Template Library
Components, Containers and IoC

• Often Frameworks provide containers for deploying components
• A container may provide at runtime functionalities needed by the components to execute
• Example: EJB containers are responsible of the persistent storage of data and of the availability of EJB’s for all authorized clients
• Using IoC, EJB containers can invoke on session beans methods like ejbRemove, ejbPassivate (store to secondary storage), and ejbActivate (restore from passive state).
• Spring’s IoC containers: a related concept...
Loosely coupled systems: advantages and techniques

• Good OO Systems should be organised as web of interacting objects
• Goal: High cohesion, low coupling
• Advantages of low coupling
  – Extensibility
  – Testability
  – Reusability
• We discuss *Dependency injection* and other techniques to achieve it

Nick Hines - *Dependency Injection and Inversion of Control* - ThoughtWorks, 2006
A Concrete Example – A Trade Monitor

As a trader I want the system to reject trades when my exposure reaches a certain limit.
Trade Monitor – The design

- TradeMonitor is coupled to LimitDao [Data Access Object] – this is not good!
  - **Extensibility** – what if we replace the database with a distributed cache?
  - **Testability** – where do the limits for test come from?
  - **Reusability** – logic is fairly generic . . .
Trade Monitor – The Design Refactored (1)

- Introduce interface/implementaVon/implementation separation
  - Logic does not depend on DAO anymore.
  - Does this really solve the problem?
- The constructor still has a static dependency on DAO

```csharp
public class TradeMonitor
{
    private ILimitRepository limitRepository;

    public TradeMonitor()
    {
        limitRepository = new LimitDao();
    }

    public bool TryTrade(string symbol, int amount)
    {
        //...}
    }
}

public interface ILimitRepository
{
    int GetExposure(string symbol);
    int GetLimit(string symbol);
}

public class TradeMonitor
{
    private ILimitRepository limitRepository;

    public TradeMonitor()
    {
        limitRepository = new LimitDao();
    }

    public bool TryTrade(string symbol, int amount)
    {
        //...}
    }
```
Trade Monitor – The Design Refactored (2)

- Introduce a **Factory**. It has the responsibility to create the required instance.
- TradeMonitor decoupled from LimitDao
- LimitDao still tightly-coupled albeit to Factory

```csharp
public class LimitFactory
{
    public static ILimitRepository GetLimitRepository()
    {
        return new LimitDao();
    }
}

public class TradeMonitor
{
    private ILimitRepository limitRepository;
    public TradeMonitor()
    {
        limitRepository = LimitFactory.GetLimitRepository();
    }

    public bool TryTrade(string symbol, int amount)
    {
        ...
    }
}
```
Trade Monitor – The Design Refactored (3)

- Introduce a `ServiceLocator`. This object acts as a (static) registry for the LimitDao you need.
- This gives us extensibility, testability, reusability

```csharp
public class ServiceLocator{
    public static void RegisterService(Type type, object impl) { . . . }
    public static object GetService(Type type) { . . . }
}

public class TradeMonitor{
    private ILimitRepository limitRepository;
    public TradeMonitor(){
        object o = ServiceLocator.GetService(typeof(ILimitRepository));
        limitRepository = o as ILimitRepository;
    }
    public bool TryTrade(string symbol, int amount){ . . . }
}
```
ServiceLocator – Pros and cons

• The Service Locator pattern succeeds in decoupling the **TradeMonitor** from the **LimitDao**
• It can be generalized in several ways, eg. to cover dynamic lookup
• Cons:
  – A form of sequence dependence remains
  – Cumbersome setup in tests
  – Service depends on infrastructure code (the **ServiceLocator**)
  – Code needs to handle lookup problems
Towards Dependency Injection

• In the original situation, we aim at relaxing the coupling using solutions based on **Inversion of Control**

Q: Which “control” is inverted?

A: The lookup of the `LimitRepository` instance from `TradeMonitor`

The plugin is created by an external **Assembler** and it is passed to `TradeMonitor` in some way.

The dependency is **injected** in the main component.
Dependency Injection

- **Dependency injection** allows avoiding hard-coded dependencies (strong coupling) and changing them.
- Allows selection among multiple implementations of a given dependency interface at run time.
- Examples:
  - load plugins dynamically
  - replace *mock objects* in test environments vs. real objects in production environments
- Three forms:
  - Setter injection
  - Constructor injection
  - *(Interface injection)*
Dependency injection based on setter methods

- What about adding a setter and let something else worry about creation and resolution?

```csharp
public class TradeMonitor
{
    private ILimitRepository limitRepository;

    public TradeMonitor()
    {
    }

    public ILimitRepository Limits
    {
        set { limitRepository = value;}
    }

    public bool TryTrade(string symbol, int amount){
        ...
    }
}
```

- The dependencies are injected from the outside
- Components are passive and are not concerned with locating or creating dependencies

This is **Setter Injection**
- Widely used in **Spring**
Dependency Injection based on Constructors

• Why not just use the constructor?

```csharp
public class TradeMonitor
{
    private ILimitRepository limitRepository;

    public TradeMonitor(ILimitRepository limitRepository)
    {
        this.limitRepository = limitRepository;
    }

    public bool TryTrade(string symbol, int amount)
    {
        ...
    }
}
```

This is **Constructor Injection**

• Widely used in *PicoContainer*

• No setters for dependent components, (obviously)

• One-shot initialisation – components are always initialised correctly

• All dependencies are clearly visible from code

• It is impossible to create cyclic dependencies
Solution based on Constructor Injection & Test Case

```csharp
public class TradeMonitor
{
    private ILimitRepository repository;

    public TradeMonitor(ILimitRepository repository) { this.repository = repository; }

    public bool TryTrade(string symbol, int amount)
    {
        int limit = repository.GetLimit(symbol);
        int exposure = repository.GetExposure(symbol);
        return ((amount + exposure) <= limit);
    }
}

[TestFixture]
public class TradeMonitorTest
{
    [Test]
    public void MonitorBlocks TradesWhenLimitExceeded()
    {
        DynamicMock mockRepository = new DynamicMock(typeof(ILimitRepository));
        mockRepository.SetupResult('GetLimit', 1000000, new Type[] { typeof(string) });
        mockRepository.SetupResult('GetExposure', 999999, new Type[] { typeof(string) });

        TradeMonitor monitor = new TradeMonitor((ILimitRepository)mockRepository.MockInstance);
        Assert.IsFalse(monitor.TryTrade('MSFT', 1000), 'Monitor should block trade');
    }
}
```
Which solution to use?

• Both Service Locator and Dependency Injection provide the desired decoupling
• With service locator, the desired component is obtained after request to the Locator
• With injection there is no explicit request: the component appears in the application class
• Inversion of control a bit harder to understand
• With Service Locator the application still depends on the locator
• It is easier to find dependencies of component if Dependency Injection is used
  – Check constructors and setters vs check all invocations to locator in the source code
Towards IoC Containers

• There are still some open questions
  – Who creates the dependencies?
  – What if we need some initialisation code that must be run after dependencies have been set?
  – What happens when we don’t have all the components?

• **IoC Containers** solve these issues [eg: Spring]
  – Have configuration – often external
  – Create objects
  – Ensure all dependencies are satisfied
  – Provide lifecycle support
Other possible solutions

• **Reflection** can be used to determine dependencies, reducing the need for config files.
  – Make components known to container.
  – Container examines constructors and determines dependencies.

• Most IoC containers support **auto-wiring**: automatic wiring between properties of a bean and other beans based, eg, on name or type.

• Auto-wiring provides other benefits:
  – Less typing, especially long assembly names.
  – Static type checking by IDE at edit time.
  – More intuitive for developer.
## IoC Containers and Features

<table>
<thead>
<tr>
<th>Container</th>
<th>Setter DI</th>
<th>Ctor DI</th>
<th>External config</th>
<th>Code config</th>
<th>Auto-wiring</th>
<th>Lifecycle support</th>
<th>Url</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.ComponentModel</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Part of .Net framework</td>
</tr>
<tr>
<td>Windsor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>✓</td>
<td>✓</td>
<td><a href="http://www.castleproject.org">http://www.castleproject.org</a></td>
</tr>
<tr>
<td>StructureMap</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓</td>
<td><a href="http://sourceforge.net/projects/structuremap">http://sourceforge.net/projects/structuremap</a></td>
</tr>
</tbody>
</table>

? = Setter based DI required for primitive dependencies  
P = Partial still requires configuration to point to assemblies to scan  
?? = More investigation
Dependency injection in Spring

- The objects that form the backbone of a Spring application are called **beans**
- A bean is an object that is instantiated, assembled, and otherwise managed by a **Spring IoC container** (*ApplicationContext*)
- Bean definition contains the information called **configuration metadata**, which is needed for the container to know the following
  - How to create a bean
  - Bean’s lifecycle details
  - Bean’s dependencies
- The configuration metadata can be supplied to the container in three possible ways:
  - **XML based configuration file** (the standard)
  - **Annotation-based** configuration
  - **Java-based** configuration
Spring IoC containers

- The **Spring container** is at the core of the Spring Framework.
- The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.
- The Spring container uses **Dependency Injection** to manage the components that make up an application.
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading the **configuration metadata** provided.
- The configuration metadata can be represented either by XML, Java annotations, or Java code.
- The diagram to the right represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.
- Quickly browsing the **Spring Architecture**...
  [https://docs.spring.io/spring/docs/3.0.x/reference/overview.html#overview-modules](https://docs.spring.io/spring/docs/3.0.x/reference/overview.html#overview-modules)
The main class, loading an Application Context

```java
public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
        HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
        obj.getMessage();
    }
}
```

The bean: a POJO (Plain Old Java Object)

```java
class HelloWorld {
    private String message;
    public void setMessage(String message) {
        this.message = message;
    }
    public void getMessage() {
        System.out.println("Your Message : " + message);
    }
}
```

The Configuration Metafile (XML)

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">
        <property name = "message" value = "Hello World!"/>
    </bean>
</beans>
```

Setter Injection (performed by the IoC container)
Software Framework Design

• Intellectual Challenging Task
• Requires a deep understanding of the problem domain
• Requires mastering of software (design) patterns, OO methods and polymorphism in particular
• Impossible to address in the course, but we can play a bit...
A Framework for the family of Divide and Conquer algorithms

• Idea: start from a well-known generic algorithm
• Apply known techniques and patterns to define a framework for a software family
• Instances of the framework, obtained by standard extension mechanism, will be concrete algorithms of the family

```plaintext
function solve (Problem p) returns Solution
{ if isSimple(p)
    return simplySolve(p);
 else
    sp[] = decompose(p);
    for (i= 0; i < sp.length; i = i+1)
        sol[i] = solve(sp[i]);
    return combine(sol);
}
```
Frozen/Hot Spots and Template Methods

- **Frozen Spot**: common (shared) aspect of the software family
- **Hot Spot**: variable aspect of the family
- **Template method**: concrete method of base class implementing behavior common to all members of the family
- A hot spot is represented by a group of abstract hook methods.
- A template method calls a hook method to invoke a function that is specific to one family member. [Inversion of Control]
- A hot spot is realized in a framework as a hot spot subsystem.
The Unification Principle

• The *unification principle* uses *inheritance* to implement the *hot spot subsystem*.

• Both the *template methods* and *hook methods* are defined in the same abstract base class.

• The hook methods are implemented in subclasses of the base class. In the figure below, the hot spot subsystem for the unification approach consists of the abstract base class and its subclasses.

A software framework is a generic application that allows the creation of different specific applications from a family. It is an abstract design that can be reused within a whole application domain. In a framework, the frozen spots of the family are represented by a set of abstract and concrete base classes that collaborate in some structure. A behavior that is common to all members of the family is implemented by a fixed, concrete template method in a base class. A hot spot is represented by a group of abstract hook methods. A template method calls a hook method to invoke a function that is specific to one family member.

A software framework is a system that is designed with generality and reuse in mind; and design patterns, which are well-established solutions to program design problems that commonly occur in practice, are the intellectual tools for achieving the desired level of generality and reuse. Two design patterns, corresponding to the two framework construction principles, are useful in implementation of the frameworks.
The Separation Principle

- The *separation principle* uses *delegation* to implement the *hot spot subsystem*.

- The *template methods* are implemented in a *concrete context class*; the *hook methods* are defined in a *separate abstract class* and implemented in its subclasses.

- The template methods thus delegate work to an instance of the subclass that implements the hook methods. In the figure below, the hot spot subsystem consists of both the client (context) class and the abstract base class and its subclasses.
Applying the unification principle: UML diagram of the solution

```java
function solve (Problem p) returns Solution
{ if isSimple(p)
    return simplySolve(p);
else
    sp[] = decompose(p);
for (i= 0; i < sp.length; i = i+1)
    sol[i] = solve(sp[i]);
return combine(sol);
}
```
public interface Problem {}
public interface Solution {};

abstract public class DivConqTemplate
{
  public final Solution solve(Problem p)
  {
    Problem[] pp;
    if (isSimple(p)) { return simplySolve(p); }
    else { pp = decompose(p); }
    Solution[] ss = new Solution[pp.length];
    for(int i=0; i < pp.length; i++)
    {
      ss[i] = solve(pp[i]);
    }
    return combine(p,ss);
  }

  abstract protected boolean isSimple (Problem p);
  abstract protected Solution simplySolve (Problem p);
  abstract protected Problem[] decompose (Problem p);
  abstract protected Solution combine(Problem p,Solution[] ss);
}

function solve (Problem p) returns Solution
{ if isSimple(p)
    return simplySolve(p);
 else
  sp[] = decompose(p);
  for (i= 0; i < sp.length; i = i+1)
  {
    sol[i] = solve(sp[i]);
    return combine(sol);
  }"
An application of the framework: QuickSort (unification principle)

```java
public class QuickSort extends DivConqTemplate {
    protected boolean isSimple (Problem p) {
        return ((QuickSortDesc)p).getFirst() >= ((QuickSortDesc)p).getLast();
    }
    protected Solution simplySolve (Problem p) {
        return (Solution) p;
    }
    protected Problem[] decompose (Problem p) {
        int first = ((QuickSortDesc)p).getFirst();
        int last = ((QuickSortDesc)p).getLast();
        int[] a = ((QuickSortDesc)p).getArr();
        int x = a[first]; // pivot value
        int sp = first;
        for (int i = first + 1; i <= last; i++) {
            if (a[i] < x) { swap (a, ++sp, i); }
        }
        swap (a, first, sp);
        Problem[] ps = new QuickSortDesc[2];
        ps[0] = new QuickSortDesc(a,first,sp-1);
        ps[1] = new QuickSortDesc(a,sp+1,last);
        return ps;
    }
    protected Solution combine (Problem p, Solution[] ss) {
        return (Solution) p;
    }
    private void swap (int[] a, int first, int last) {
        int temp = a[first];
        a[first] = a[last];
        a[last] = temp;
    }
}
```

Fig. 5. Quicksort Problem and Solution implementation.

```java
public class QuickSortDesc implements Problem, Solution {
    public QuickSortDesc(int[] arr, int first, int last) {
        this.arr = arr; this.first = first; this.last = last;
    }
    public int getFirst() { return first; }
    public int getLast() { return last; }
    private int[] arr; // instance data
    private int first, last;
}
```

Fig. 6. Quicksort application.
Applying the separation principle:
UML diagram of the solution

Fig. 7. Strategy pattern for divide and conquer framework.

```java
function solve (Problem p) returns Solution
{ if isSimple(p)
    return simplySolve(p);
else
    sp[] = decompose(p);
for (i= 0; i < sp.length; i = i+1)
    sol[i] = solve(sp[i]);
return combine(sol);
}
```
public final class DivConqContext
{
    public DivConqContext (DivConqStrategy dc)
    {
        this.dc = dc;
    }
    public Solution solve (Problem p)
    {
        Problem[] pp;
        if (dc.isSimple(p)) { return dc.simplySolve(p); }
        else { pp = dc.decompose(p); }
        Solution[] ss = new Solution[pp.length];
        for (int i = 0; i < pp.length; i++)
        {
            ss[i] = solve(pp[i]);
        }
        return dc.combine(p, ss);
    }
    public void setAlgorithm (DivConqStrategy dc)
    {
        this.dc = dc;
    }
    private DivConqStrategy dc;
}

Fig. 8. Strategy context class implementation.

abstract public class DivConqStrategy
{
    abstract public boolean isSimple (Problem p);
    abstract public Solution simplySolve (Problem p);
    abstract public Problem[] decompose (Problem p);
    abstract public Solution combine(Problem p, Solution[] ss);
}

Fig. 9. Strategy object abstract class.
Conclusions

• Frameworks as state-of-the-art solutions for supporting reuse and extensibility of software solutions
• Inversion of Control
• Sometimes large amount of glue code, but often generated automatically

• Suggested reading: *Why do I hate Frameworks?*
  http://discuss.joelonsoftware.com/default.asp?joel.3.219431