

# 301AA - Advanced Programming

Lecturer: **Andrea Corradini**

[andrea@di.unipi.it](mailto:andrea@di.unipi.it)

<http://pages.di.unipi.it/corradini/>

***AP-21: 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.

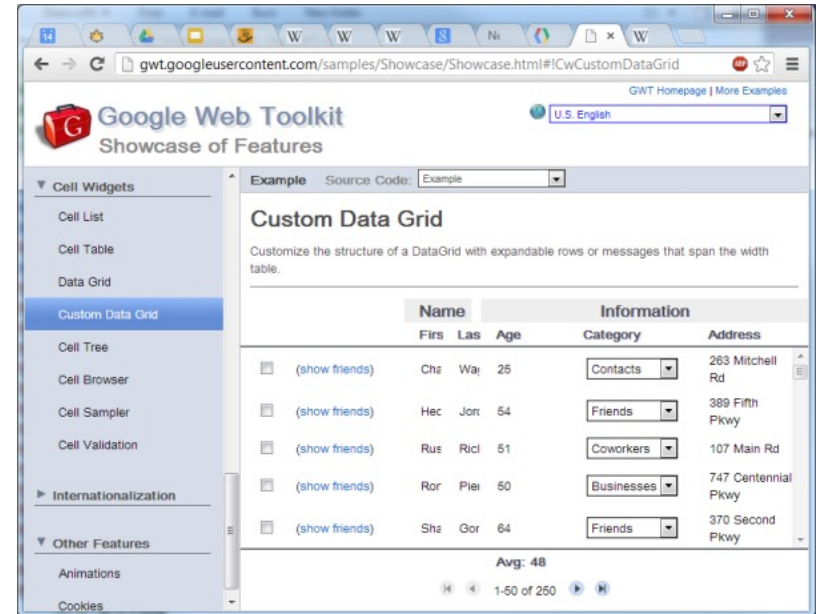
*Clemens Szyperski, ECOOP 1996*

- Examples: ***Java Beans, CLR Assemblies***
- ***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/delegates)

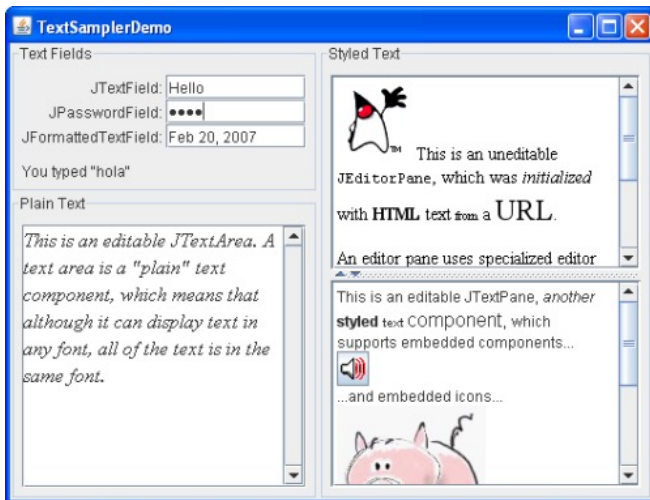
# Towards Component Frameworks

- **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.
- Examples:
  - **GUI Frameworks**
  - **Web Frameworks**
  - **Concurrency Frameworks**

# Examples of Frameworks



## Web Application Frameworks



## GUI Toolkits

# Examples: 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: GUI Frameworks

- Frameworks for Application with GUI
  - **MFC** - Microsoft Foundation Class Library. C++ object-oriented library for Windows.
  - **Gnome** – Written in C; mainly for Linux
  - **Qt** - Cross-platform; written in C++

# Examples: Web 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.
  - **Spring** - for Java-based enterprise web applications
  - **Flask** – micro-framework in Python, highly extensible (authentication, validation, OR mapper... as extensions)



# 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
- Allow 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 Integrated Development Environments (IDEs)

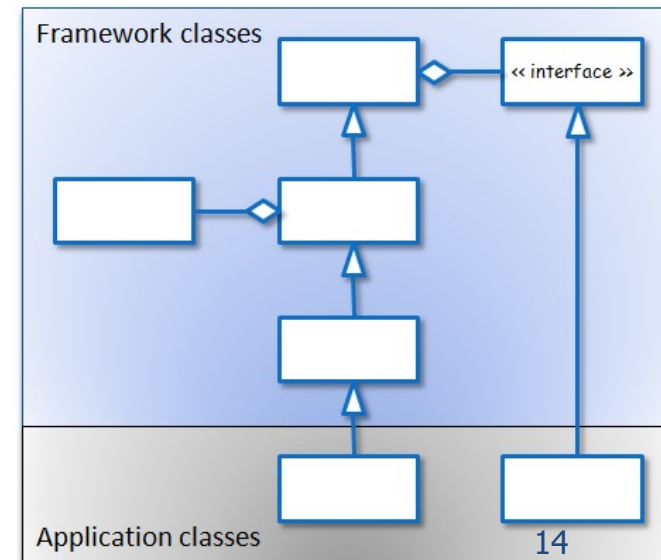
- Orthogonal concepts
- A framework can be supported by several IDEs
  - Eg: [Spring](#) supported by [Spring Tool Suite](#) (based on Eclipse), [NetBeans](#), [IntelliJ IDEA](#), [Eclipse](#), ...
- An IDE can support several frameworks
  - Eg: [NetBeans](#) supports JavaBeans, Spring, J2EE, Maven, Hibernate, JavaServer Faces, Struts, Qt,...

# 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 within the framework language:
    - Subclassing & overriding methods
    - Implementing interfaces
    - Registering event handlers
  - Plug-ins: framework can load certain extra code in a specific format



# 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

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

TEXT

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

GUI

- In text-based interaction, the order of interactions and of invocations is decided by the the code.
- In the GUI-based interaction, the GUI loop decides when to invoke the methods (listeners), based on the order of events

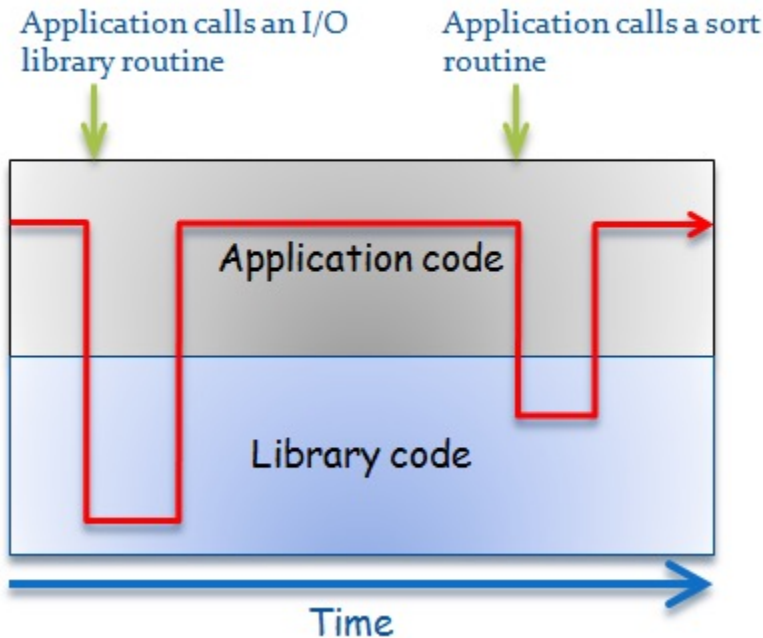


# Inversion of Control in Frameworks

- 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

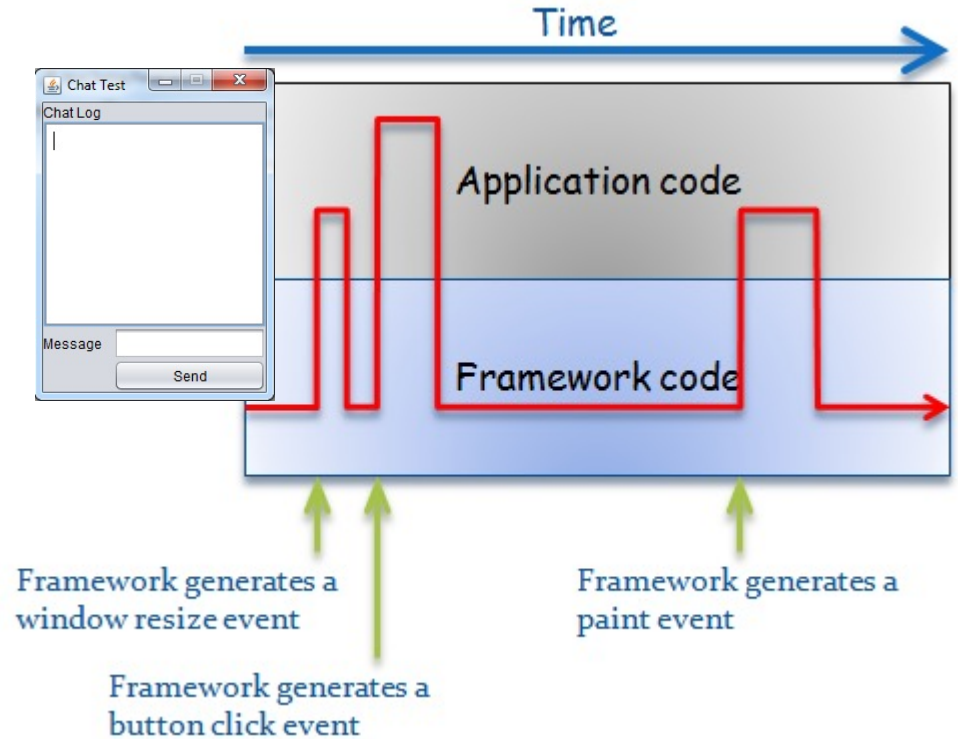
# Inversion of Control

## Traditional Program Execution



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

## Inversion of Control

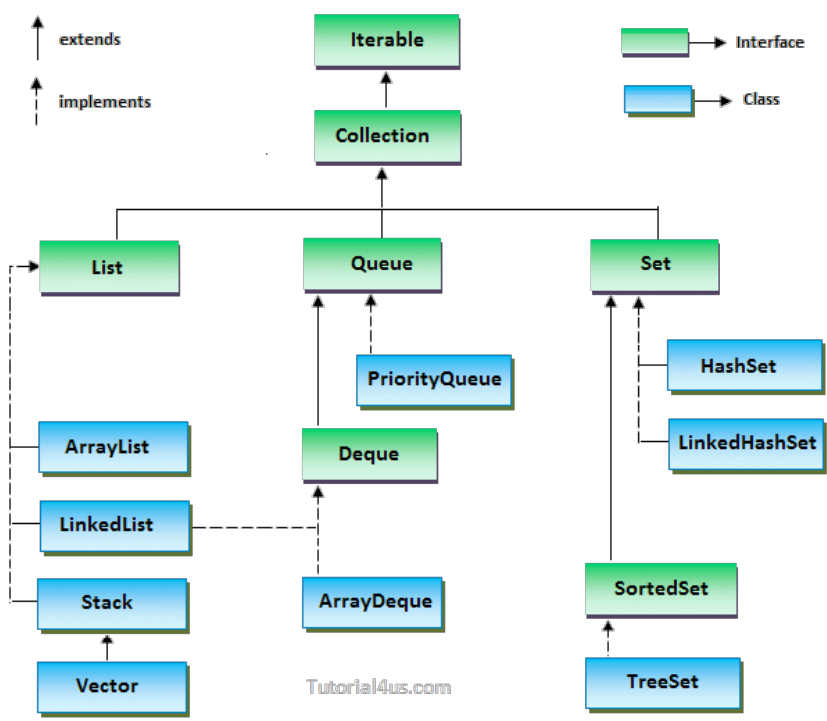


The framework has control over the execution flow, calling app code for app-specific behavior.

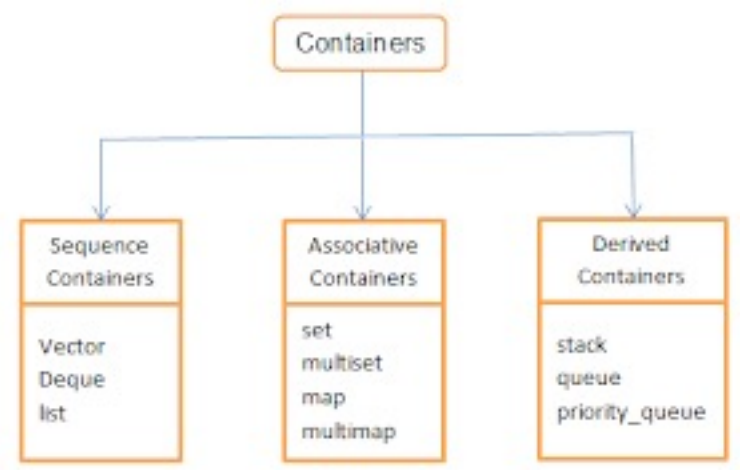
# Frameworks vs Libraries

- 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*?

# JCF vs STL



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...



# More on Inversion of Control

- **Control**: not only *control flow*, but also control over *dependencies, coupling, configuration*
- **Inversion**: component gives up control to a framework and agrees to play by some rules
- Framework calls component in well-defined ways (setters, template methods, interface)

## Dependency injection

- IoC with respect to *dependencies*
- something outside a component handles:
  - configuration (properties)
  - wiring / dependencies (components)
- component-oriented
- removes *coupling*
  - coupling of configuration and dependencies to the point of use
  - coupling of component to concrete dependent components
- somewhat contrary to encapsulation

# A Concrete Example – A Trade Monitor

- *A trader wants that the system rejects trades when the exposure reaches a certain limit*
- Thus the component **TradeMonitor** (a class...) provides a method **TryTrade** which checks the condition
- The **current exposure** and the **exposure limit** are stored in some persistent storage, and are accessed by **TryTrade** using another component, a **DAO** (**Data Access Object**)
- We discuss various solutions to limit dependencies among the two components

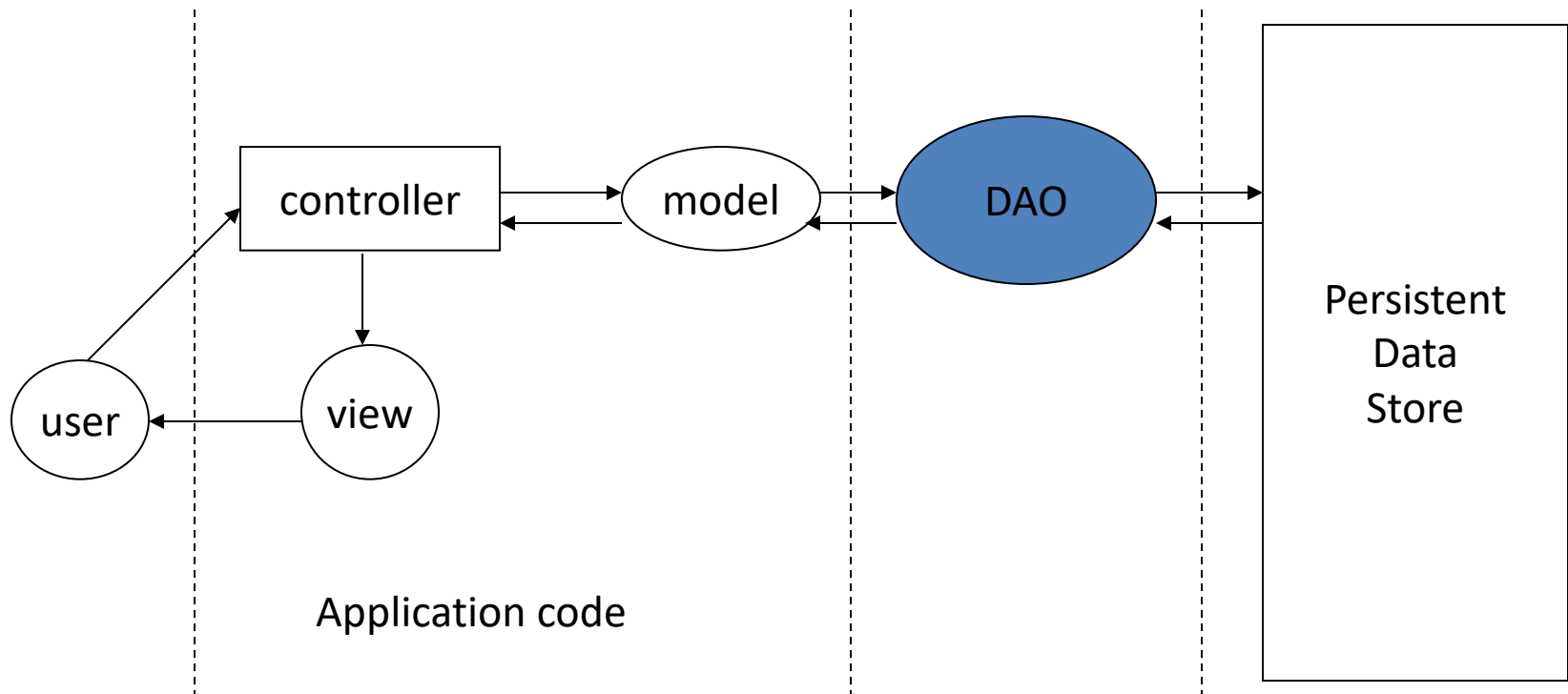
```
public class TradeMonitor
{
    // other stuff

    public bool TryTrade(string symbol, int amount){
        int limit = limitDao.GetLimit(symbol);
        int exposure = limitDao.GetExposure(symbol);
        return (exposure + amount > limit) ? false : true;
    }
}
```



# Data Access Object (DAO)

- A Java EE design pattern



# Trade Monitor – The first design

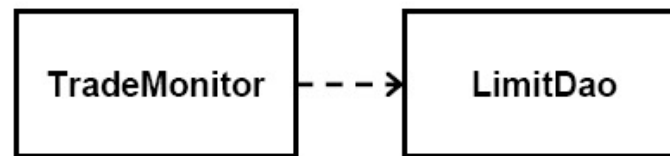
```
public class TradeMonitor
{
    private LimitDao limitDao;

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

    public bool TryTrade(string symbol, int amount)
    {
        int limit = limitDao.GetLimit(symbol);
        int exposure = limitDao.GetExposure(symbol);
        return (exposure + amount > limit)? false : true;
    }
}
```

```
public class LimitDao
{
    public int GetExposure(string symbol)
    {
        // Do something with the database
    }

    public int GetLimit(string symbol)
    {
        // Do something with the database
    }
}
```



- TradeMonitor is tightly coupled to LimitDao
  - **Extensibility** – what if we replace the database with a distributed cache?
  - **Testability** – where do the limits for test come from?

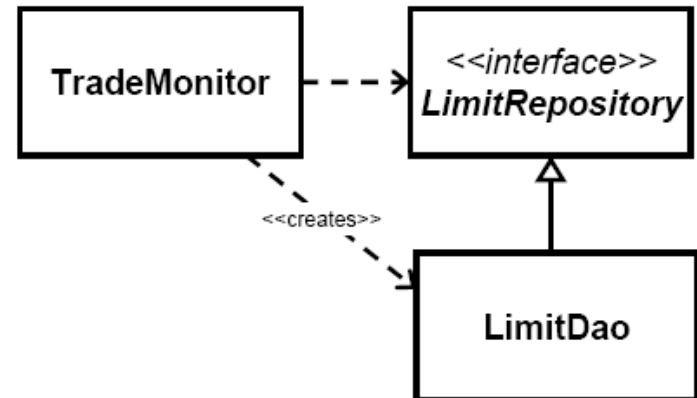
# Trade Monitor – The Design Refactored (1)

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

```
public interface LimitRepository
{
    int GetExposure(string symbol);
    int GetLimit(string symbol);
}
public class LimitDao extends LimitRepository
{
    public int GetExposure(string symbol){...}
    public int GetLimit(string symbol){...}
}
public class TradeMonitor
{
    private LimitRepository limitRepository;

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

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



# Trade Monitor – The Design Refactored (2)

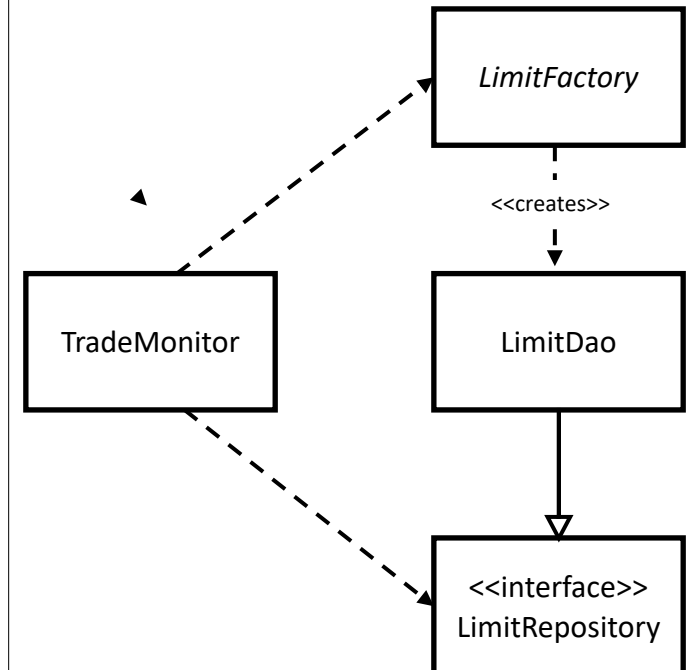
- Introduce a **Factory**. It has the responsibility to create the required instance.
- **TradeMonitor** decoupled from **LimitDao**
- **LimitDao** still tightly-coupled, this time to **Factory**

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

public class TradeMonitor
{
    private LimitRepository limitRepository;

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

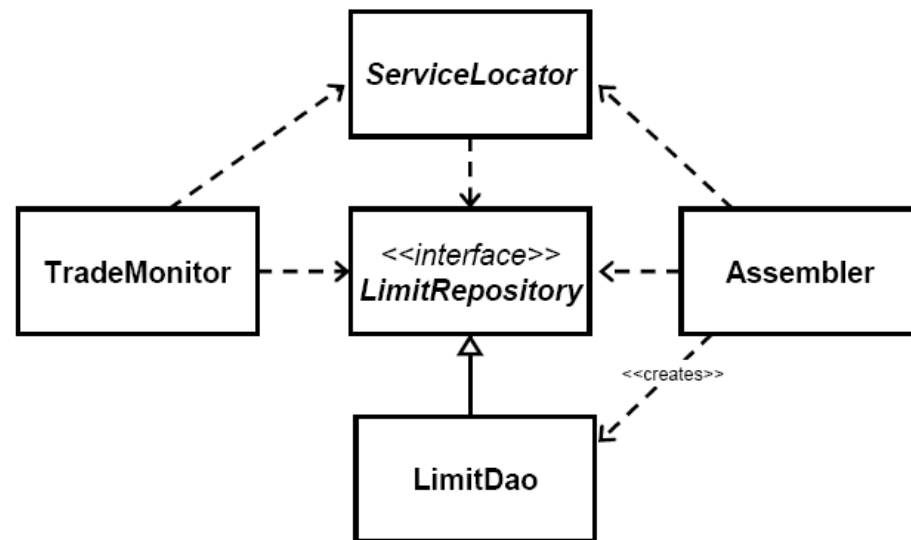
    public bool TryTrade(string symbol, int amount)
    {
        . . .
    }
}
```



# 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
- Note that an external **Assembler** sets up the registry

```
public class ServiceLocator{  
  
    public static void RegisterService(Type t, object o)  
        { . . . }  
    public static object GetService(Type t)  
        { . . . }  
}  
  
public class TradeMonitor{  
    private LimitRepository limitRepository;  
  
    public TradeMonitor() {  
        object o =  
            ServiceLocator.GetService(typeof(LimitRepository));  
        limitRepository = (LimitRepository) o;  
    }  
  
    public bool TryTrade(string symbol, int amount) {  
        . . .  
    }  
}
```



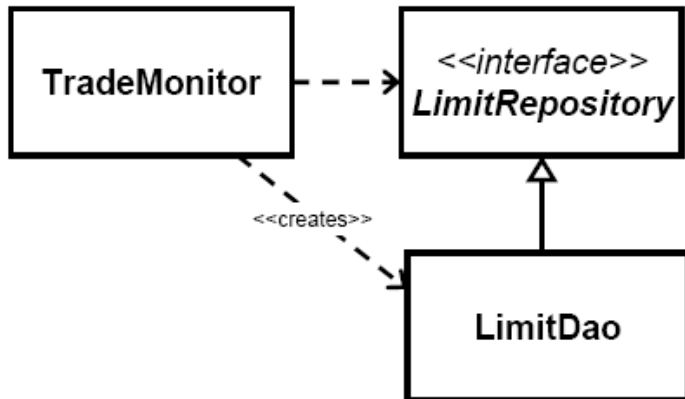
# ServiceLocator – Pros and cons

- The Service Locator pattern succeeds in decoupling the **TradeMonitor** from the **LimitDao**
- Allows new components to be dynamically created and used by other components later
- It can be generalized in several ways, eg. to cover dynamic lookup

## Cons:

- Every component that needs a dependency must have a reference to the service locator
- All components need to be registered with the service locator
- If bound **by name**:
  - Services can't be type-checked
  - Component has a dependency to the dependent component names
  - if many components share an instance but later you want to specify different instance for some, this becomes difficult
- If bound **by type**:
  - Can only bind one instance of a type in a container
- Code needs to handle lookup problems

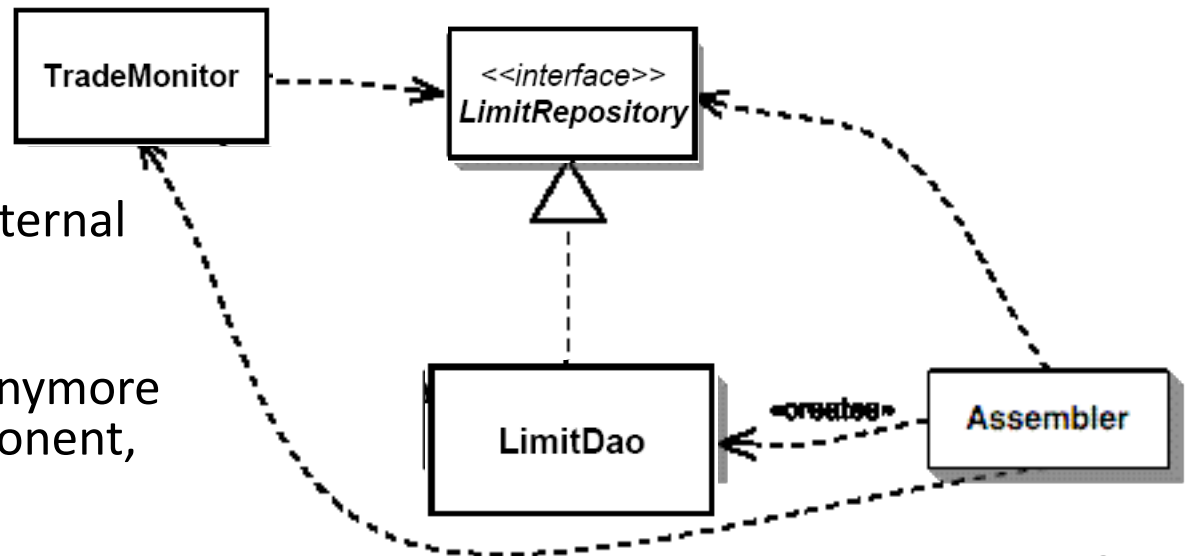
# Towards Dependency Injection



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

Q: Which “control” is inverted?

A: The **dependency** of **TradeMonitor** from the **LimitDao**



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

Thus the dependency is not anymore in the code of the main component, but it is **injected** into it

# 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**

- Idea: add a **setter**, leaving creation and resolution to others

```
public class TradeMonitor
{
    private LimitRepository limitRepository;

    public TradeMonitor()
    {
    }

    public LimitRepository Limits
    {
        set { limitRepository = value;}
    }
    public bool TryTrade(string symbol, int amount){
        . . .
    }
}
```

This is **Setter Injection**

- Widely used in **Spring**

- Pros:
  - Leverages existing JavaBean reflective patterns
  - Simple, often already available
- Cons:
  - Possible to create partially constructed objects
  - Advertises that dependency can be changed at runtime (as opposed to constructor)

# Dependency Injection based on **Constructors**

- Why not just use the constructor?

```
public class TradeMonitor
{
    private LimitRepository limitRepository;

    public TradeMonitor(LimitRepository
                        limitRepository)
    {
        this.limitRepository = limitRepository;
    }
    public bool TryTrade(string symbol, int amount){
        . . .
    }
}
```

This is ***Constructor Injection***

- Widely used in ***PicoContainer***

Pros:

- Object can't be partially constructed
- Simple, often already available

Cons:

- Bidirectional dependencies between objects can be tricky
- Constructors can easily get big and parameters confusing
- If lots of optional dependencies, may have lots of constructors
- Can make class evolution more complicated (an added dependency affects all users of the class) wrt setter injection

# Exploiting Constructor Injection for Testing

```
public class TradeMonitor
{
    private LimitRepository repository;

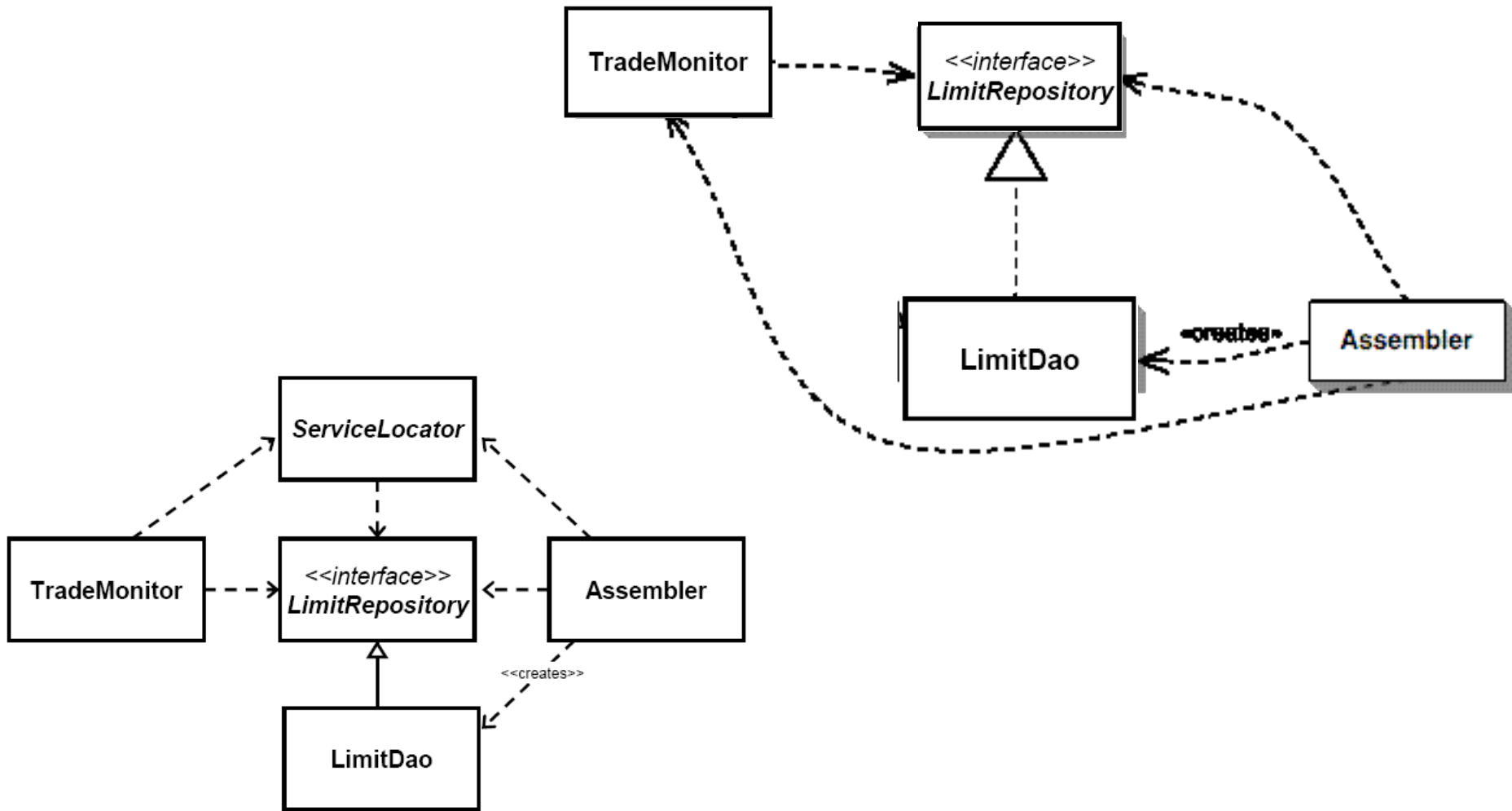
    public TradeMonitor(LimitRepository repository) { this.repository =
repository; }

    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 MonitorBlocksTradesWhenLimitExceeded()
    {
        DynamicMock mockRepository = new DynamicMock(typeof(LimitRepository));
        mockRepository.SetupResult('GetLimit', 1000000, new Type[] { typeof(string) });
        mockRepository.SetupResult('GetExposure', 999999, new Type[] { typeof(string) });

        TradeMonitor monitor = new
TradeMonitor((LimitRepository)mockRepository.MockInstance);
        Assert.IsFalse(monitor.TryTrade('MSFT', 1000), 'Monitor should block trade');
    }
}
```

# Summary: decoupling using Service Locator vs Dependency Injection



# Which solution to use?

- Both **Service Locator** and **Dependency Injection** provide the desired decoupling
- With service locator, the desired component is obtained after request by the **TradeMonitor** to the **Locator**: no IoC
- With dependency 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? (who is the “Assembler”?)
  - 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.
  - Static type checking by IDE at edit time.
  - More intuitive for developer.

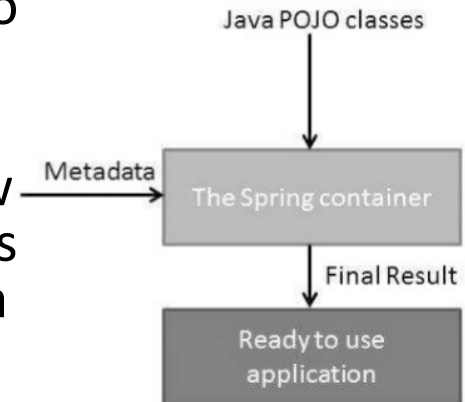
# Dependency injection in Spring

- 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**
- 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 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.



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

**The bean: a POJO (Plain Old Java Object)**

Setter Injection  
(performed by the  
IoC container)

```
<?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>
```

**The Configuration Metafile (XML)**

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

**The main class, loading an Application Context**