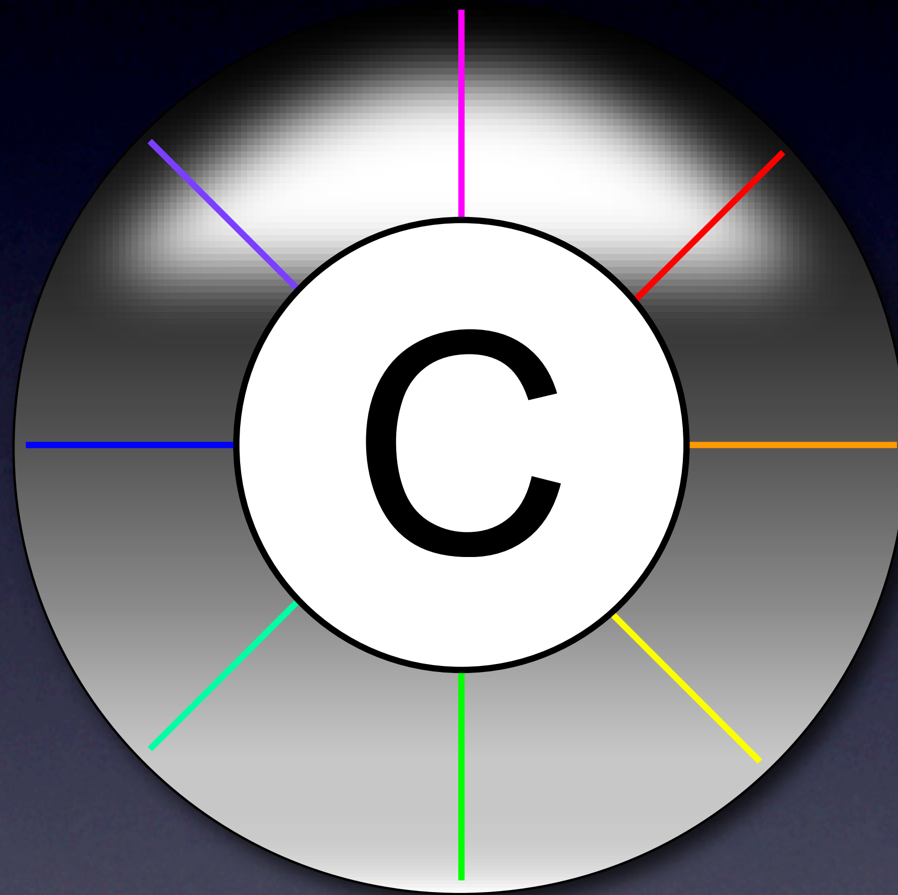
Cytoscape 3, OSGi, and Spring Framework



Spring Framework

Keiichiro Ono
University of California, San Diego
Cytoscape Core Developer

Outline

- Dependency Injection
- What is Spring Framework?
- OSGi and Spring
- Integration to Cytoscape 3

Dependency Injection

Dependency Injection (DI)

A Design Pattern to achieve:

- ✓ Separate configuration from use
- ✓ Low Coupling
- ✓ Easy unit testing

Why DI? (1)

- Cytoscape 3 will be an expandable, and UI-independent platform
  - Desktop application, Web application, or Web Service
  - Plugins interact to each other to build complex services/functions

Why DI? (2)

- OSGi solves:
  - Library dependency/conflict problem

    ➡ Important for applications with plugin architecture
  - Dynamic loading problem
  - Modularity
  - Interface-Based Design (Service Framework)

Why DI? (3)

- Remaining Issues
  - How can we make Cytoscape re-configurable?
    - From Desktop to Sever Backend
  - How can we manage instances of components

➡ DI Pattern helps to solve these problems

DI

- Relationships between components are separated from objects themselves
- Components are communicating through interfaces, not concrete classes

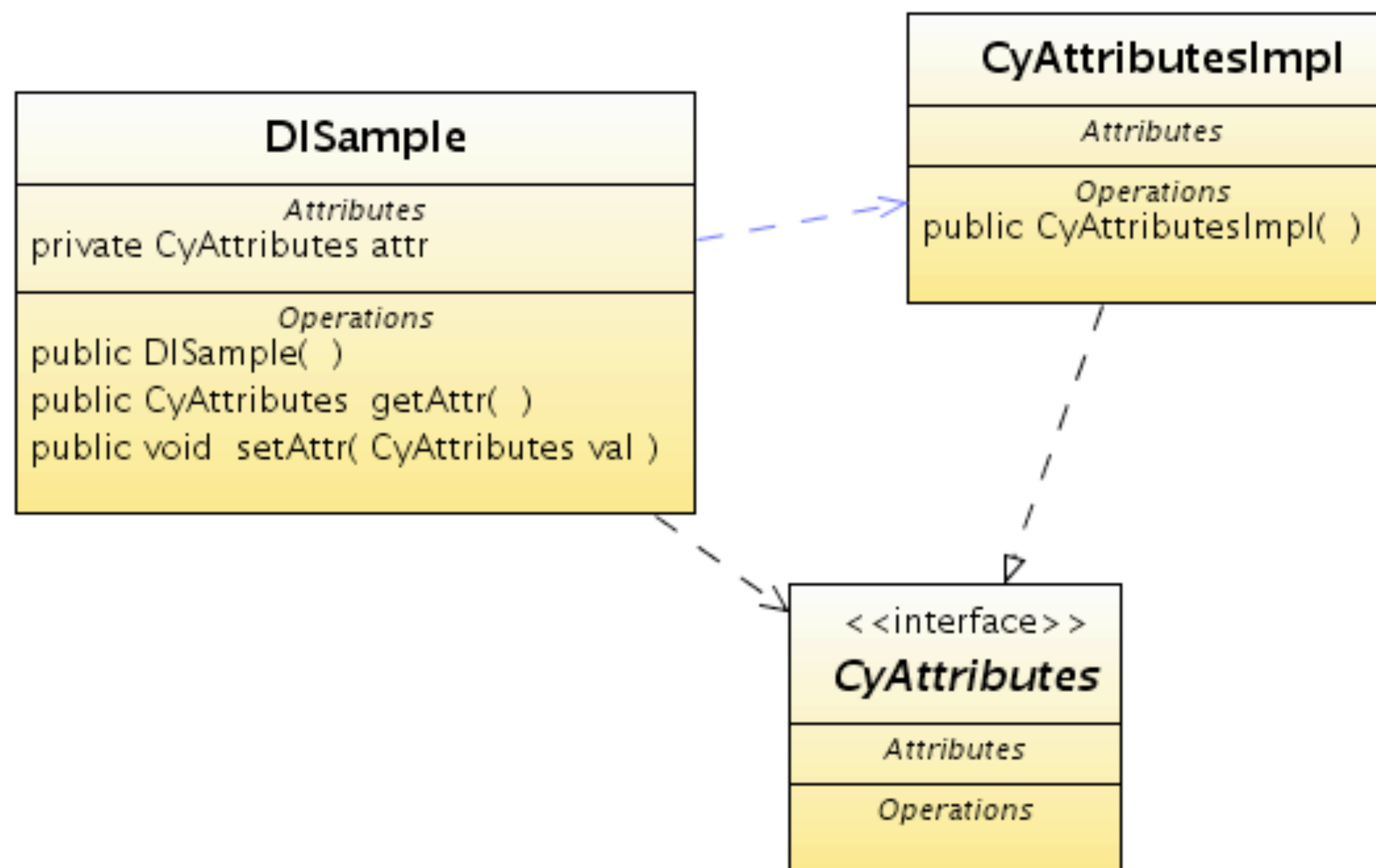
Example

- Class *DISample* has a private field called attr, which is a *CyAttribute* Object
- Need a new instance for the field

Example: Without DI

```
public class DISample {
    private CyAttribute attr;
    public DISample() {
        this.attr = new CyAttributesImpl();
    }

    public void doSomething(String key, String value) {
        attr.setStringAttribute(key, value);
        .
        .
    }
}
```

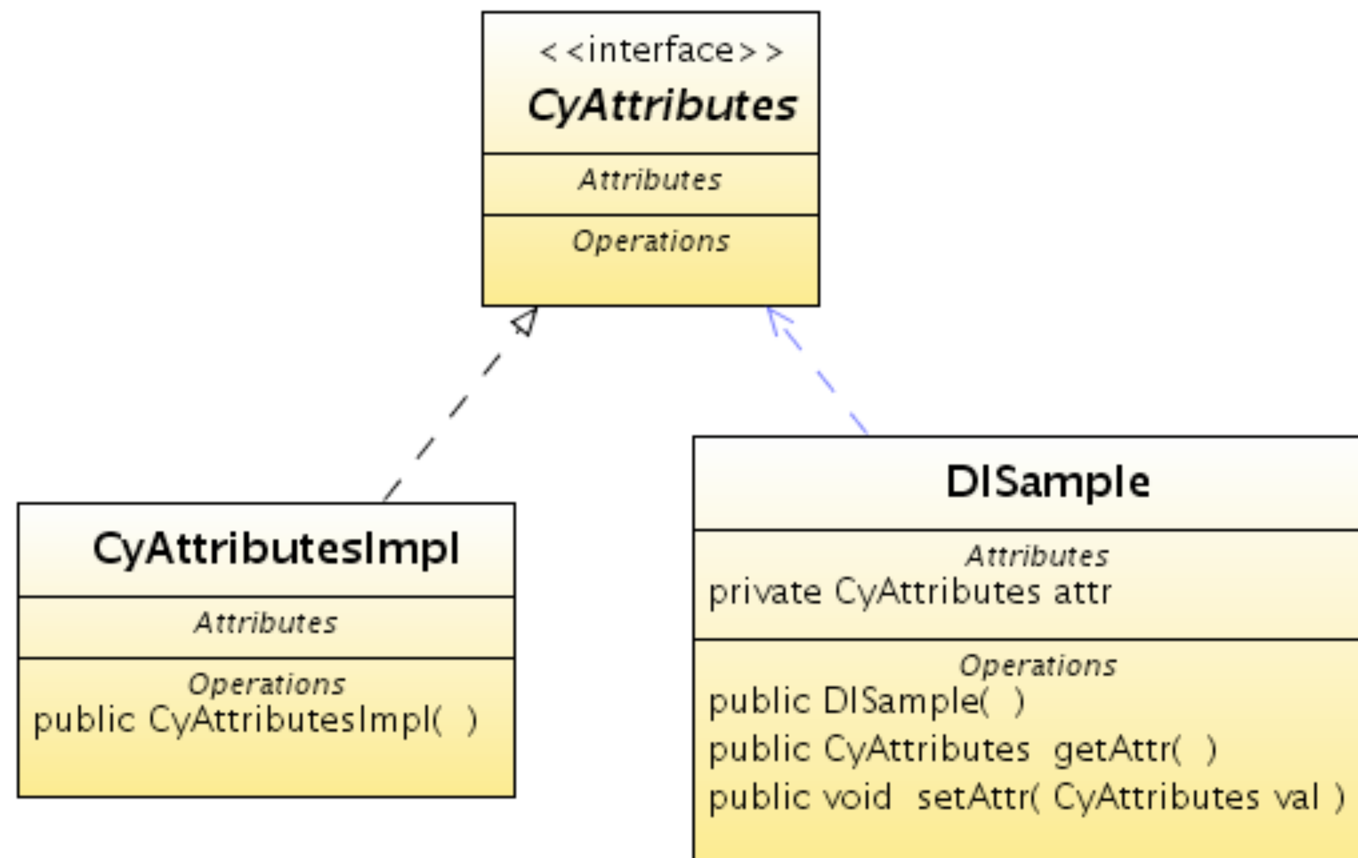
Example: Without DI



Example: With DI

```
public class DISample {
    private CyAttribute attr;
    public DISample(CyAttribute attr) {
      this.attr = attr;
    }

    public void doSomething(String key, String value) {
       attr.setStringAttribute(key, value);
       .
       .
    }
}
```

Example: With DI



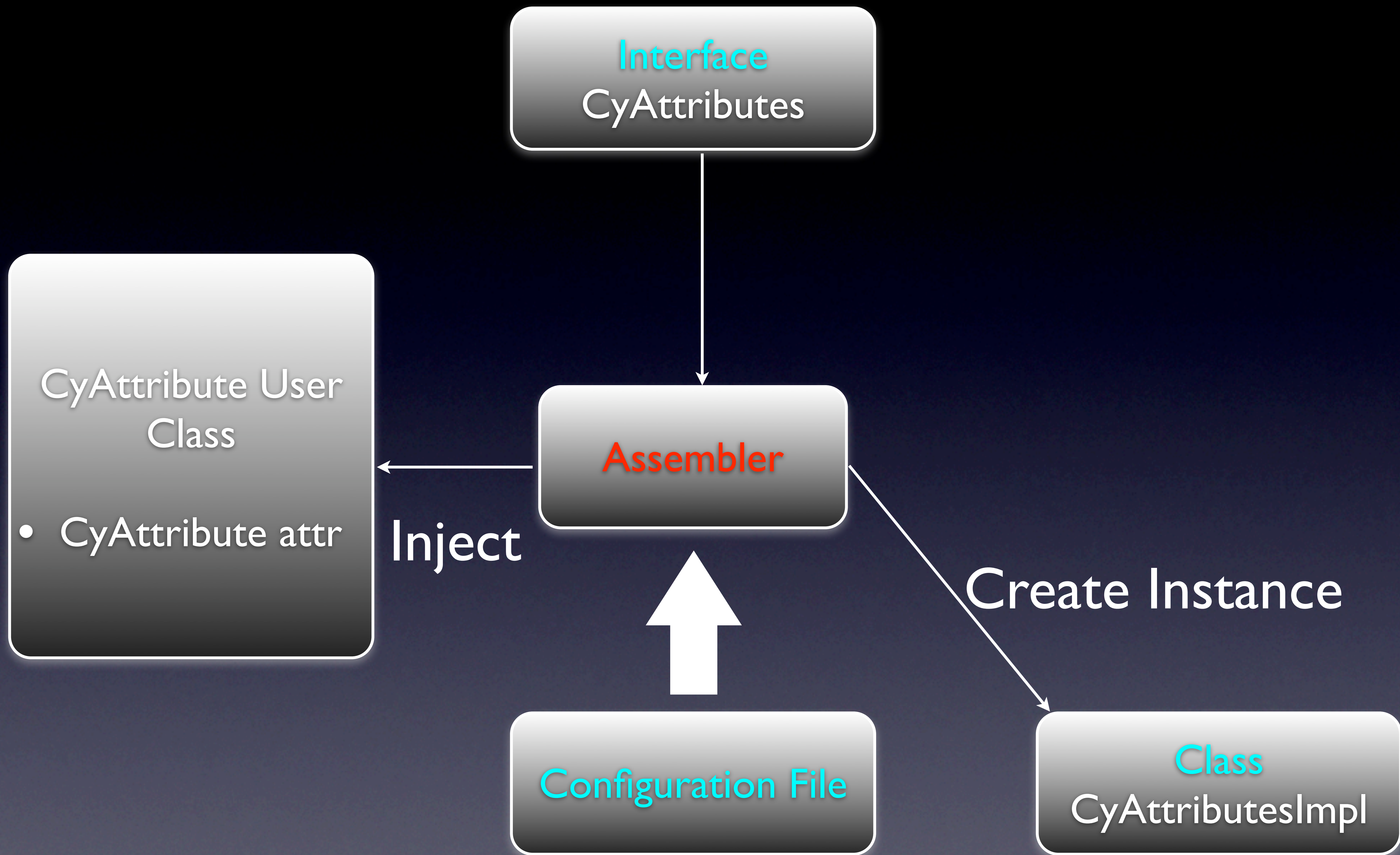
Simplify DI

- But, we need to manage all construction ourselves!
- This is complicated and redundant

➡We can use a framework to simplify this problem

DI Frameworks

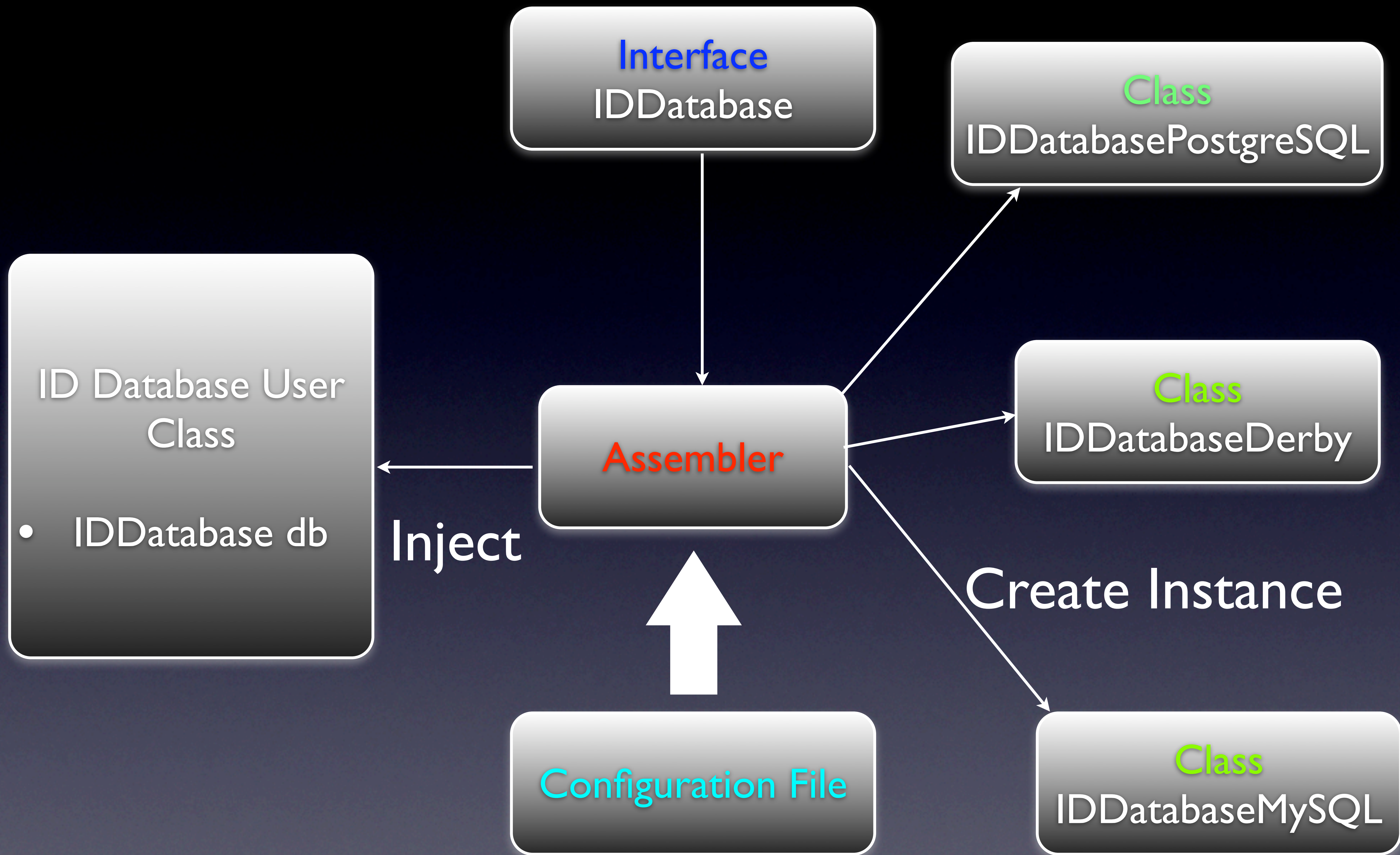
- Google Guice
- Apache iPOJO
- Seasar2
- PicoContainer
- Spring Framework



Interface
CyAttributes
CyAttribute User Class
• CyAttribute attr
Assembler
Inject
Create Instance
Configuration File
Class
CyAttributesImpl

With DI Container

- Implementation is hidden from user object
  - Implementations are replaceable
- Instances are managed by container


Interface
IDDatabase
Class
IDDatabasePostgreSQL
ID Database User Class
• IDDatabase db
Inject
Assembler
Class
IDDatabaseDerby
Create Instance
Configuration File
Class
IDDatabaseMySQL

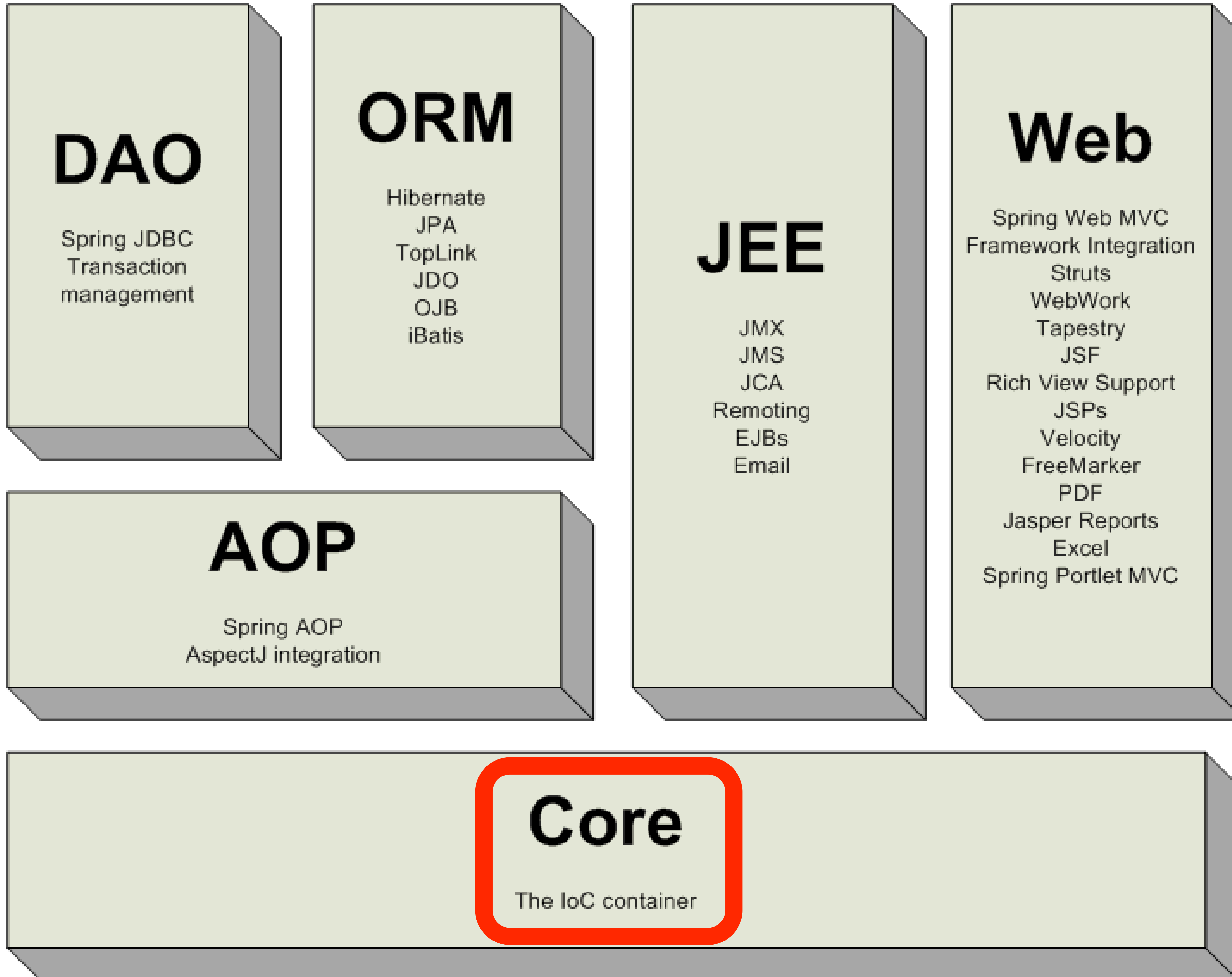
Applications

- Testing
  - Unit testing
  - Integration testing
    - ✓ Mock Object
- Reusable Software Components
  - Re-wire components to build systems for different environments
    - ▸ This is why DI pattern is used in many enterprise-level applications

Spring Framework

Spring Framework

- The concept was introduced by ***Expert One-on-One J2EE Design and Development*** (Rod Johnson)
- Light-weight container implementing Dependency Injection pattern
- Goal: Building a framework for expandable, maintainable large-scale applications without complexity of Enterprise Java Beans


DAO
Spring JDBC
Transaction management
ORM
Hibernate
JPA
TopLink
JDO
OJB
iBatis
JEE
JMX
JMS
JCA
Remoting
EJBs
Email
Web
Spring Web MVC
Framework Integration
Struts
WebWork
Tapestry
JSF
Rich View Support
JSPs
Velocity
FreeMarker
PDF
Jasper Reports
Excel
Spring Portlet MVC
AOP
Spring AOP
AspectJ integration
Core
The IoC container

DI in Spring Framework

- *Separating configuration from use*
- *Separate implementation from API*
- Configuration of the system is stored in XML files and annotation
- Components are called *Beans*

Example: Layout Manager

- Scenario
  - Layout Manager has a collection of actual layout algorithms
  - Needs to provide list of currently available layout algorithms to other part of application

Layout Algorithm Interface

```
public interface LayoutAlgorithm {
    public String getName();
    public void doLayout();
}
```

Layout Algorithm Beans

```
<bean name="gridLayout"
    class="org.cytoscape.layout2.internal.algorithms.GridLayout">
    <property name="name" value="grid" />
</bean>
<bean name="circularLayout"
    class="org.cytoscape.layout2.internal.algorithms.CircularLayout">
    <property name="name" value="circular" />
</bean>
<bean name="organicLayout"
    class="org.cytoscape.layout2.internal.algorithms.OrganicLayout">
    <property name="name" value="organic" />
</bean>
```

Layout Manager Interface

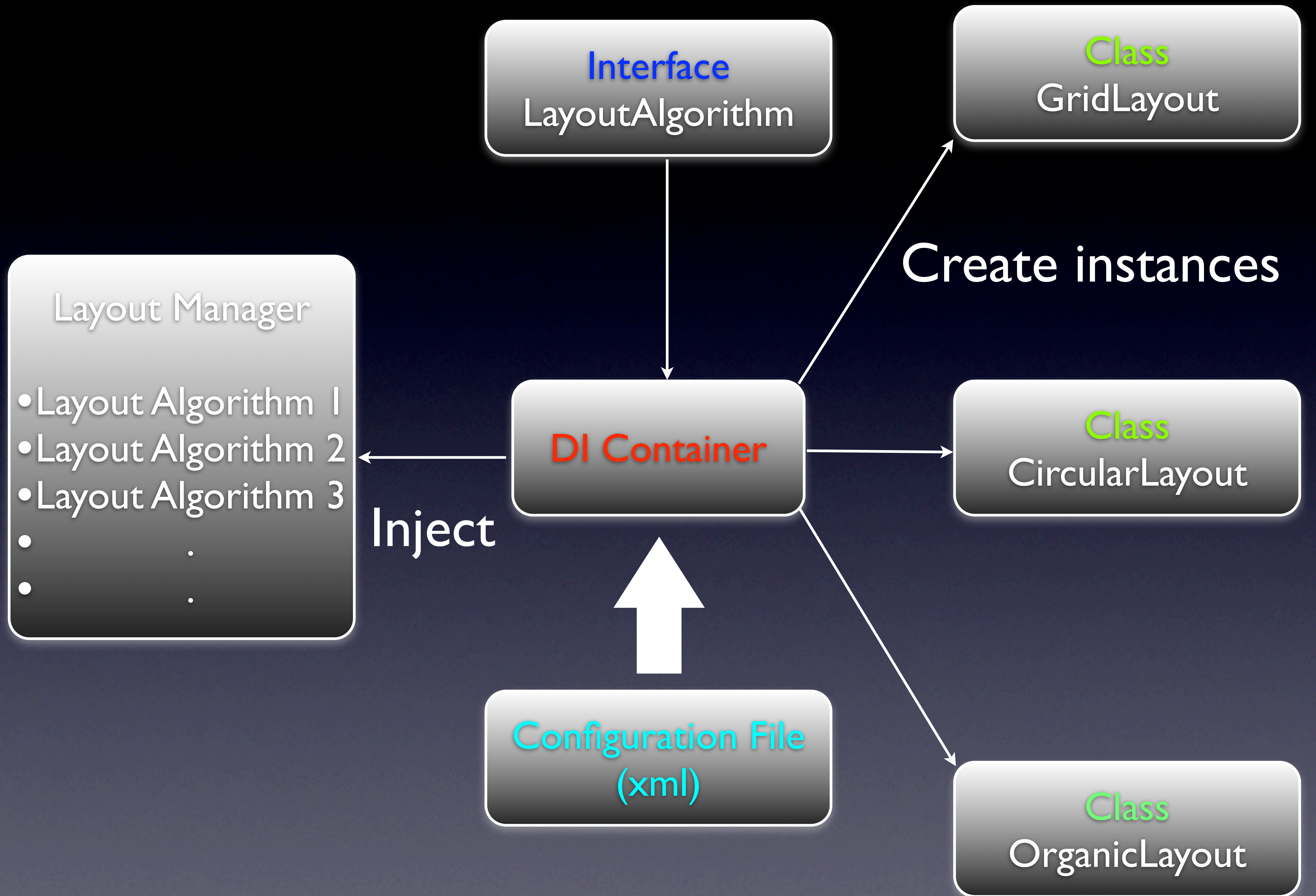
```
public interface LayoutManager {
    public List<LayoutAlgorithm> getLayouts();
}
```

Layout Manager Implementation

```
public class LayoutManagerImpl implements LayoutManager {

    private List<LayoutAlgorithm> layouts;

    public void setLayouts(List<LayoutAlgorithm> layouts) {
        this.layouts = layouts;
    }

    public List<LayoutAlgorithm> getLayouts() {
        return layouts;
    }
}
```

Inject Layout Beans

```
<bean name="layoutManager"

class="org.cytoscape.layout2.internal.LayoutManagerImpl"
    scope="singleton">
    <property name="layouts">
        <list>
            <ref bean="gridLayout" />
            <ref bean="circularLayout" />
            <ref bean="organicLayout" />
        </list>
    </property>
</bean>
```


Interface
LayoutAlgorithm
Class
GridLayout
Create instances
Layout Manager
•Layout Algorithm 1
•Layout Algorithm 2
•Layout Algorithm 3
•
•
DI Container
Class
CircularLayout
Inject
Configuration File
(xml)
Class
OrganicLayout

Advantages

- Instances are managed by the container
- Code is cleaner than managing singletons in each classes
- Components are wired through interfaces, so swapping implementations is easy
- Properties of objects are stored in one place

Problems

- Cytoscape 3 is an OSGi application
- OSGi bundles are dynamically loaded/ unloaded. How can Spring be aware those dynamic state changes?
- Access to classes is now strictly controlled by the OSGi platform. How can we share beans (objects) between bundles?

Spring Dynamic Modules for OSGi (Spring DM)

Spring DM

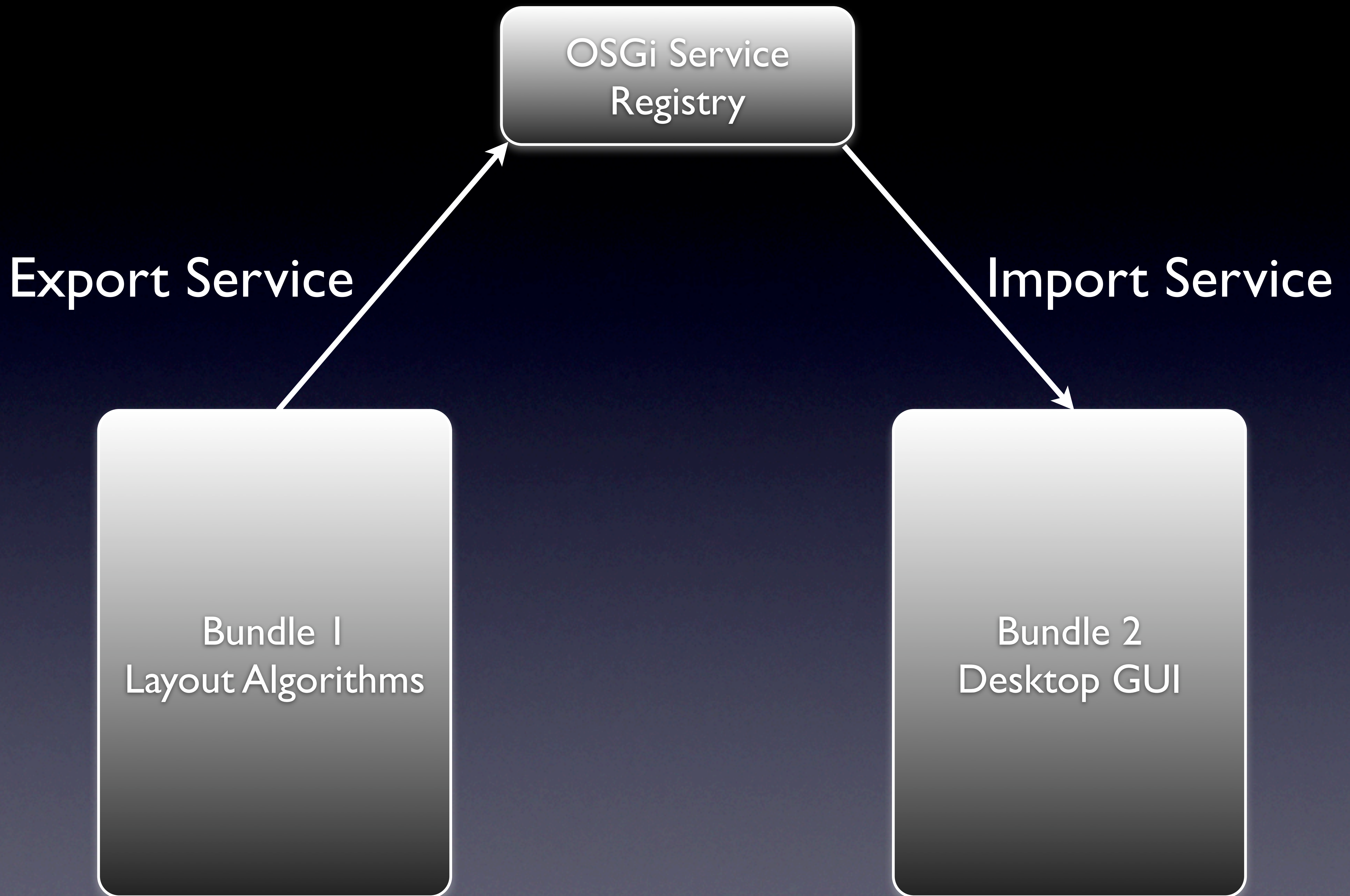
- Extension for Spring Framework to use beans in OSGi environment
- Apply Spring's philosophy to OSGi world
  - Keep as many objects as POJOs as possible
  - Make objects independent from OSGi API

Review: OSGi Service

- In OSGi systems, objects shared by multiple bundles are registered as OSGi service
- OSGi services are managed by service registry
- Services are just Java interfaces

OSGi Service Usecase

1. Bundles with layout algorithms export them as OSGi services
2. Cytoscape Desktop GUI accesses OSGi service registry and import available layout algorithms
3. Desktop creates menu items based on the properties of each layout algorithms and add them to the layout menu


OSGi Service
Registry
Export Service
Import Service
Bundle 1
Layout Algorithms
Bundle 2
Desktop GUI

Spring DM

- Compose OSGi services as a collection of beans
- Service dependency is managed automatically

Spring DM Architecture

Application contexts

Spring Bean

Exported service

Imported service

R u n t i m e

instantiates

Infrastructure bundles

Application bundles

D e p l o y m e n t

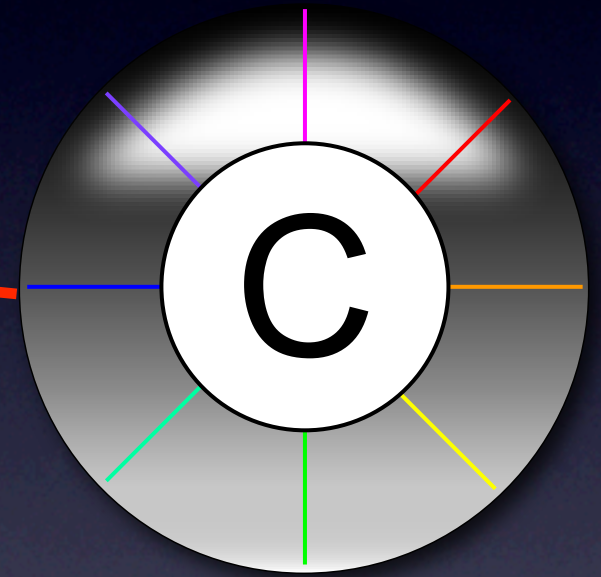
OSGi Service Platform

Java Virtual Machine

"Spring-powered" application bundles

Spring Framework

OSGi Services Defined as a Complex of Beans



Example: OSGi Service Based Layout Manager

- Layout algorithms are distributed in different bundles
    - Core layout algorithm bundle
    - Commercial algorithms bundle
    - User bundles with custom layout algorithms
- Cytoscape Desktop GUI needs to create structured menu for layout algorithms

Design

- One Layout Algorithm
  = One OSGi Service
  = One Spring Bean
- Desktop GUI generates menu based on information provided from services
- Timing and dependency of the services are controlled by Spring DM

Without Spring DM

- Each bundle needs to implement *BundleActivator* and export each layout algorithms in the activator's *start()* method
- Desktop GUI uses *ServiceTracker* to locate layout services in OSGi service registry
- Services will be created dynamically, so timing is important and it should be handled manually by Desktop GUI code

With Spring DM

- Define each layout algorithm as bean
- Export the bean as OSGi service
- Desktop GUI imports layout services as set of beans
- Timing will be controlled automatically by Spring DM
- No BundleActivator/ServiceTracker. Objects are independent from OSGi API

Calling OSGi API
vs
Spring DM

Calling OSGi API

- Straightforward
- Everything is in Java code
- Coupling

➡ Once user classes call OSGi API, now they are coupled with OSGi framework

Spring DM

- Everything is POJO
- No coupling between service objects and OSGi / Spring API
- Instances are managed by container
- XML setting files and annotation

➡ Configuration and relationship between classes are not in Java code

OSGi Best Practices

by Hargrave (IBM) and Kriens (aQute)

http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-1419.pdf

Avoid OSGi Framework API Coupling

Use an OSGi-ready IoC container like Declarative Services or Spring OSGi to express these dependencies in a declarative form

Let the IoC containers handle all of the OSGi API calls

Demo: Command Layer with Spring DM

Other Features

- Beans written in other programming languages
- Aspect Oriented Programming (AOP)
  - Logging
  - Benchmarking
- Interoperability with other frameworks
  - Web Service, Database Access, Web Application

Writing Plugin in Other Programming Languages

Implementing Cytoscape Command in Ruby

- From 3.0, Cytoscape has an interface *Command*
- In this example, implementing Cytoscpe Command in scripting language (Ruby)

Command Interface

```
package org.cytoscape.command;

public interface Command {
   public String getName();
   public String getDescription();
   public void execute()
      throws Exception;
}
```

Inject Java Object to Ruby

```
<bean name="networkAnalysisEngine"
  class="org.cytoscape.analysis.NetworkStatisticsUtil">
</bean>

<lang:jruby id="rubyPluginBean"
  script-source="META-INF/SampleRubyPlugin.rb"
  script-interfaces="org.cytoscape.plugin.Command"
scope="singleton">
  <lang:property name="name" value="Sample Ruby Plugin" />
  <lang:property name="description"
    value="Plugin written in ruby scripting language." />
  <lang:property name="analysisEngine"
    ref="networkAnalysisEngine" />
</lang:jruby>
```

Plugin Code Written in Ruby

```
require 'java'
include_class 'org.cytoscape.command.Command'

class SampleRubyPlugin
    def setName(name)
      @@name = name
    end

    def setDescription(description)
      @@description = description
    end

    def setVerison(version)
      @@version = version
    end

    def setAnalysisEngine(analysisEngine)
      @@analysisEngine = analysisEngine
    end

    def getName
      @@name
    end

    def getDescription
      @@description
    end

    def execute
      @@analysisEngine.process(network)
    end
end

SampleRubyPlugin.new
```

Summary

- Dependency Injection pattern is useful for applications with plugin architecture
- DI pattern makes application scalable
- Spring DM provides simpler way to manage OSGi services

Thank You!