Information systems modeling

Tomasz Kubik

Examples of GoF Design Patterns in Java's core libraries

Creational patterns

Singleton
(recognizeable by creational methods returning the same instance (usually of itself) everytime)
`java.lang.Runtime#getRuntime()`
`java.awt.Desktop#getDesktop()`
`java.lang.System#getSecurityManager()`

Prototype
(recognizeable by creational methods returning a different instance of itself with the same properties)
`java.lang.Object#clone()` (the class has to implement `java.lang.Cloneable`)

Abstract factory
(recognizeable by creational methods returning the factory itself which in turn can be used to create another abstract/interface type)

- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

Factory method
(recognizeable by creational methods returning an implementation of an abstract/interface type)

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`
- `java.net.URLStreamHandlerFactory#createURLStreamHandler(String) (Returns singleton object per protocol)`
- `java.util.EnumSet#of()`
- `javax.xml.bind.JAXBContext#createMarshaller() and other similar methods`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

Examples of GoF Design Patterns in Java's core libraries

Creational patterns

Bridge

(recognizeable by creational methods taking an instance of *different* abstract/interface type and returning an implementation of own abstract/interface type which *delegates/uses* the given instance)

- None comes to mind yet. A fictive example would be `new LinkedHashMap(LinkedHashSet<K>, List<V>)` which returns an unmodifiable linked map which doesn't clone the items, but uses them. The `java.util.Collections#newSetFromMap()` and `singletonXXX()` methods however comes close.

Composite

(recognizeable by behavioral methods taking an instance of *same* abstract/interface type into a tree structure)

- `java.awt.Container#add(Component)` (practically all over Swing thus)
- `javax.faces.component.UIComponent#getChildren()` (practically all over JSF UI thus)

Builder

(recognizeable by creational methods returning the instance itself)

- `java.lang.StringBuilder#append()` (unsynchronized)
- `java.lang.StringBuffer#append()` (synchronized)
- `java.nio.ByteBuffer#put()` (also on `CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer`)
- `javax.swing.GroupLayout.Group#addComponent()`
- All implementations of `java.lang.Appendable`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

Examples of GoF Design Patterns in Java's core libraries

Creational patterns

Decorator

(recognizeable by creational methods taking an instance of *same* abstract/interface type which adds additional behaviour)

- All subclasses of `java.io.InputStream`, `OutputStream`, `Reader` and `Writer` have a constructor taking an instance of same type.
- `java.util.Collections`, the `checkedXXX()`, `synchronizedXXX()` and `unmodifiableXXX()` methods.
- `javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

Facade

(recognizeable by behavioral methods which internally uses instances of *different* independent abstract/interface types)

- `javax.faces.context.FacesContext`, it internally uses among others the abstract/interface types `LifeCycle`, `ViewHandler`, `NavigationHandler` and many more without that the enduser has to worry about it (which are however overrideable by injection).
- `javax.faces.context.ExternalContext`, which internally uses `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse`, etc.

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

Examples of GoF Design Patterns in Java's core libraries

Creational patterns

Adapter

(recognizeable by creational methods taking an instance of different abstract/interface type and returning an implementation of own/another abstract/interface type which decorates/overrides the given instance)

- `java.util.Arrays#asList()`
- `java.util.Collections#list()`
- `java.util.Collections#enumeration()`
- `java.io.InputStreamReader(InputStream) (returns a Reader)`
- `java.io.OutputStreamWriter(OutputStream) (returns a Writer)`
- `javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()`

Flyweight

(recognizeable by creational methods returning a cached instance, a bit the "multiton" idea)

- `java.lang.Integer#valueOf(int)` (also on `Boolean`, `Byte`, `Character`, `Short`, `Long` and `BigDecimal`)

Proxy

(recognizeable by creational methods which returns an implementation of given abstract/interface type which in turn *delegates/uses* a *different* implementation of given abstract/interface type)

- `java.lang.reflect.Proxy`
- `java.rmi.*`
- `javax.ejb.EJB` (explanation https://stackoverflow.com/questions/25514361/when-using-ejb-does-each-managed-bean-get-its-own-ejb-instance)
- `javax.inject.Inject` (explanation https://stackoverflow.com/questions/29651008/field-getobj-returns-all-nulls-on-injected-cdi-managed-beans-while-manually-i/29672591#29672591)
- `javax.persistence.PersistenceContext`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

Examples of GoF Design Patterns in Java's core libraries

Behavioral patterns

Chain of responsibility

(recognizeable by behavioral methods which (indirectly) invokes the same method in *another* implementation of *same* abstract/interface type in a queue)

- `java.util.logging.Logger#log()`
- `javax.servlet.Filter#doFilter()`

Command

(recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a *different* abstract/interface type which has been *encapsulated* by the command implementation during its creation)

- All implementations of `java.lang.Runnable`
- All implementations of `javax.swing.Action`

Interpreter

(recognizeable by behavioral methods returning a *structurally* different instance/type of the given instance/type; note that parsing/formatting is not part of the pattern, determining the pattern and how to apply it is)

- `java.util.Pattern`
- `java.text.Normalizer`
- All subclasses of `java.text.Format`
- All subclasses of `javax.el.ELResolver`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

Examples of GoF Design Patterns in Java's core libraries

Behavioral patterns

Iterator

(recognizeable by behavioral methods sequentially returning instances of a *different* type from a queue)

- All implementations of `java.util.Iterator` (thus among others also `java.util.Scanner`!).
- All implementations of `java.util.Enumeration`

Mediator

(recognizeable by behavioral methods taking an instance of different abstract/interface type (usually using the command pattern) which delegates/uses the given instance)

- `java.util.Timer` (all `scheduleXXX()` methods)
- `java.util.concurrent.Executor#execute()`
- `java.util.concurrent.ExecutorService` (the `invokeXXX()` and `submit()` methods)
- `java.util.concurrent.ScheduledExecutorService` (all `scheduleXXX()` methods)
- `java.lang.reflect.Method#invoke()`

Memento

(recognizeable by behavioral methods which internally changes the state of the *whole* instance)

- `java.util.Date` (the setter methods do that, `Date` is internally represented by a `long` value)
- All implementations of `java.io.Serializable`
- All implementations `of javax.faces.component.StateHolder`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

Examples of GoF Design Patterns in Java's core libraries

Behavioral patterns

Observer (or Publish/Subscribe)

(recognizeable by behavioral methods which invokes a method on an instance of *another* abstract/interface type, depending on own state)

- `java.util.Observer/java.util.Observable` (rarely used in real world though)
- All implementations of `java.util.EventListener` (practically all over Swing thus)
- `javax.servlet.http.HttpSessionBindingListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.faces.event.PhaseListener`

State

(recognizeable by behavioral methods which changes its behaviour depending on the instance's state which can be controlled externally)

- `javax.faces.lifecycle.LifeCycle#execute()` (controlled by `FacesServlet`, the behaviour is dependent on current phase (state) of JSF lifecycle)

Strategy

(recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a *different* abstract/interface type which has been *passed-in* as method argument into the strategy implementation)

- `java.util.Comparator#compare()`, executed by among others `Collections#sort()`.
- `javax.servlet.http.HttpServlet`, the `service()` and all `doXXX()` methods take `HttpServletRequest` and `HttpServletResponse` and the implementor has to process them (and not to get hold of them as instance variables!).
- `javax.servlet.Filter#doFilter()`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

Examples of GoF Design Patterns in Java's core libraries

Behavioral patterns

Template method

(recognizeable by behavioral methods which already have a "default" behaviour definied by an abstract type)

- All non-abstract methods `of java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer`.
- All non-abstract methods of `java.util.AbstractList`, `java.util.AbstractSet` and `java.util.AbstractMap`.
- `javax.servlet.http.HttpServlet`, all the `doXXX()` methods by default sends a `HTTP 405 "Method Not Allowed"` error to the response. You're free to implement none or any of them.

Visitor

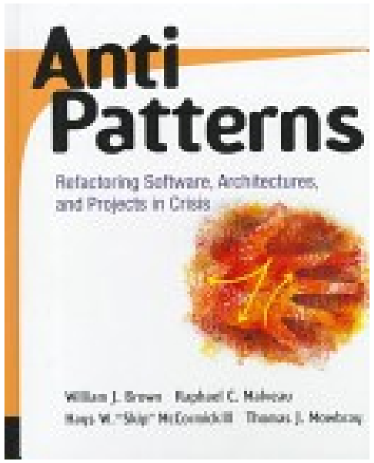
(recognizeable by two *different* abstract/interface types which has methods definied which takes each the *other* abstract/interface type; the one actually calls the method of the other and the other executes the desired strategy on it)

- `javax.lang.model.element.AnnotationValue` and `AnnotationValueVisitor`
- `javax.lang.model.element.Element` and `ElementVisitor`
- `javax.lang.model.type.TypeMirror` and `TypeVisitor`
- `java.nio.file.FileVisitor` and `SimpleFileVisitor`
- `javax.faces.component.visit.VisitContext` and `VisitCallback`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195

AntiPatterns

- like their design pattern counterparts, define an industry vocabulary for the common defective processes and implementations within organizations
  - (see: https://sourcemaking.com/antipatterns)
- are practices that appear very easy to follow, but have bad side effects in reality
- some of bad code may not look so obviously bad to beginners
  - (see: https://www.odi.ch/prog/design/newbies.php)



Anti Patterns
Refactoring Software, Architectures, and Projects in Crisis
William J. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray

Bad

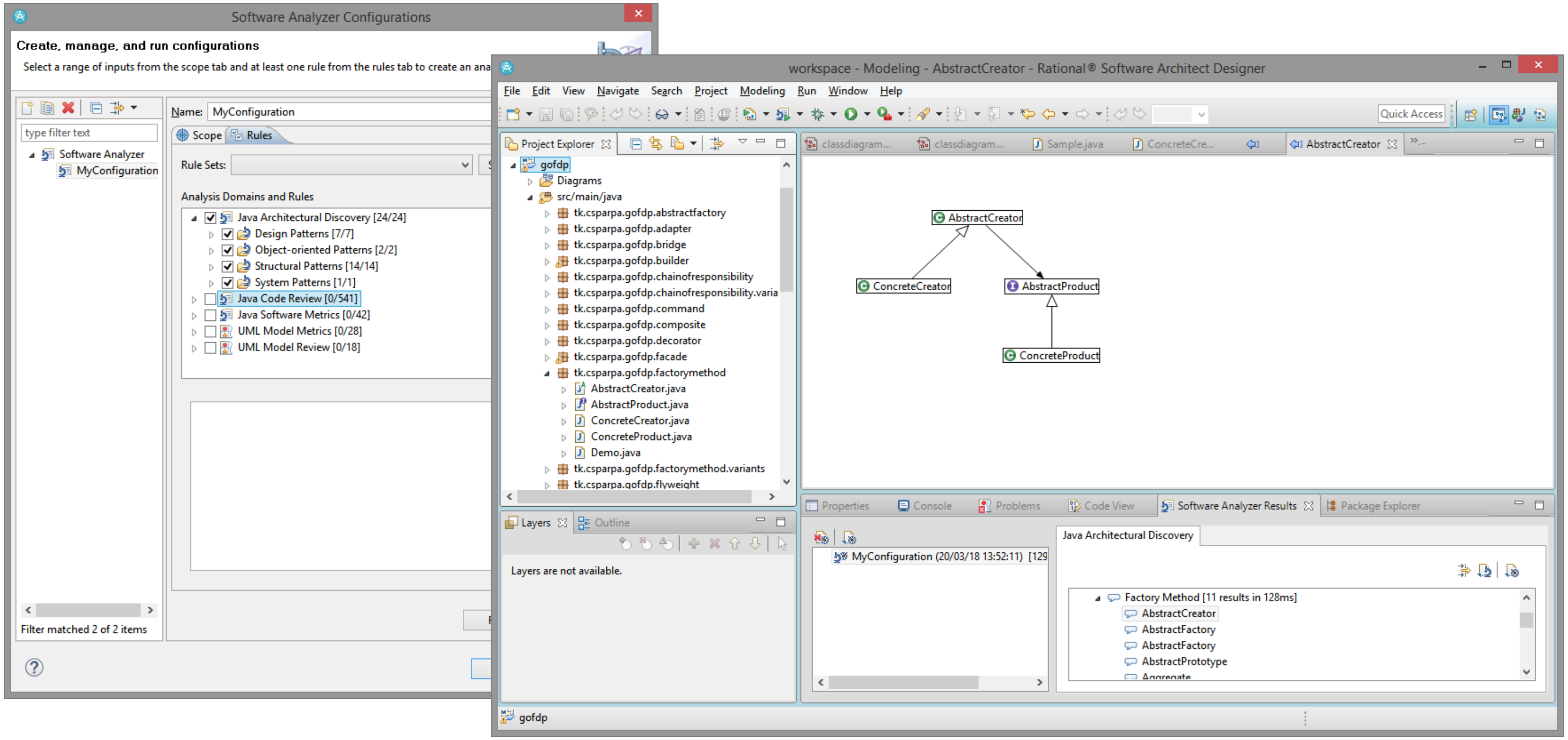
```
String s = "";
for (Person p : persons) {
    s += ", " + p.getName();
}
//remove first comma
s = s.substring(2);
```

Good

```
StringBuilder sb = new
// well estimated buffer
StringBuilder(persons.size() * 16);
for (Person p : persons) {
// the JIT optimizes the if away out of the loop (peeling)
    if (sb.length() > 0) sb.append(", ");
    sb.append(p.getName);
}
```

Rational® Software Architect Designer: Software Analyzer

- This tool can recognize design patterns used by the programmer but also deliver some statistics and results of code review
  - (see: https://www.ibm.com/developerworks/rational/library/08/0429_gutz1/)



Software Analyzer Configurations

Create, manage, and run configurations

Select a range of inputs from the scope tab and at least one rule from the rules tab to create an ana

type filter text

- Software Analyzer
  - MyConfiguration

Filter matched 2 of 2 items

Name: MyConfiguration

Scope | Rules

Rule Sets:

Analysis Domains and Rules

- [✓] Java Architectural Discovery [24/24]
  - [✓] Design Patterns [7/7]
  - [✓] Object-oriented Patterns [2/2]
  - [✓] Structural Patterns [14/14]
  - [✓] System Patterns [1/1]
- [ ] Java Code Review [0/541]
- [ ] Java Software Metrics [0/42]
- [ ] UML Model Metrics [0/28]
- [ ] UML Model Review [0/18]

workspace - Modeling - AbstractCreator - Rational® Software Architect Designer

File Edit View Navigate Search Project Modeling Run Window Help

Quick Access

Project Explorer

- gofdp
  - Diagrams
  - src/main/java
    - tk.csparpa.gofdp.abstractfactory
    - tk.csparpa.gofdp.adapter
    - tk.csparpa.gofdp.bridge
    - tk.csparpa.gofdp.builder
    - tk.csparpa.gofdp.chainofresponsibility
    - tk.csparpa.gofdp.chainofresponsibility.varia
    - tk.csparpa.gofdp.command
    - tk.csparpa.gofdp.composite
    - tk.csparpa.gofdp.decorator
    - tk.csparpa.gofdp.facade
    - tk.csparpa.gofdp.factorymethod
      - AbstractCreator.java
      - AbstractProduct.java
      - ConcreteCreator.java
      - ConcreteProduct.java
      - Demo.java
    - tk.csparpa.gofdp.factorymethod.variants
    - tk.csparpa.gofdp.flyweight

classdiagram... | classdiagram... | Sample.java | ConcreteCre... | AbstractCreator

Layers | Outline

Layers are not available.

Properties | Console | Problems | Code View | Software Analyzer Results | Package Explorer

MyConfiguration (20/03/18 13:52:11) [129

Java Architectural Discovery

- Factory Method [11 results in 128ms]
  - AbstractCreator
  - AbstractFactory
  - AbstractFactory
  - AbstractPrototype
  - Aggregate

gofdp

Source code for GoF patterns implementation: https://github.com/csparpa/gof-design-patterns/tree/master/java

Java custom annotations

- introduced in jdk1.5
- many frameworks are based on them
- **have no representation in UML !!!**

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import
java.lang.annotation.RetentionPolicy;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MyAn {
    public String name();
    public String value();
}
```

```
@Retention
     RetentionPolicy.RUNTIME
     RetentionPolicy.CLASS
     RetentionPolicy.SOURCE
@Target
     ElementType.ANNOTATION_TYPE
     ElementType.CONSTRUCTOR
     ElementType.FIELD
     ElementType.LOCAL_VARIABLE
     ElementType.METHOD
     ElementType.PACKAGE
     ElementType.PARAMETER
     ElementType.TYPE
@Inherited
@Documented
```

http://tutorials.jenkov.com/java/annotations.html

Java custom annotation use

- often used trough reflection API
- this might make some troubles when working with modules (from jdk9)
- therefore jdk1.8 is still alive (!)

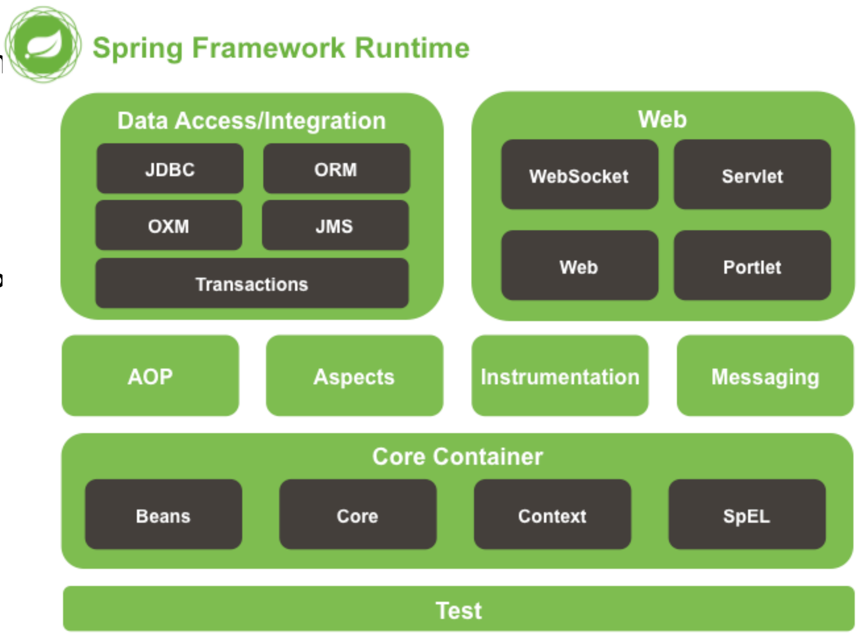
```
import java.lang.annotation.Annotation;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.ElementType;

@MyAn(name="someName",  value = "someValue")
public class MyClass {
  MyClass(){
     Class aClass = MyClass.class;
     Annotation[] annotations = aClass.getAnnotations();
     for(Annotation annotation : annotations){
       if(annotation instanceof MyAn){
         MyAn myAn = (MyAn) annotation;
         System.out.println("name: " + myAn.name());
         System.out.println("value: " + myAn.value());
       }
     }
  }
}
```

Spring framework

current version: 5.2.4

- consists of features
  - **core technologies:** dependency injection, events, resources, i18n, validation, data binding, type conversion SpEL, AOP
  - **testing:** mock objects, TestContext framework, Spring MVC Test, WebTestClient
  - **data access:** transactions, DAO support, JDBC, ORM, Marshalling XML
  - **web servlet:** Spring MVC, WebSocket, SockJS, STOMP Messaging
  - web reactive: Spring WebFlux, WebClient, WebSocket
  - **integration:** remoting, JMS, JCA, JMX, email, tasks, scheduling, cache
  - **languages:** Kotlin, Groovy, dynamic languages

  organized into various modules, grouped into:
  - Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Aspects, Instrumentation, Messaging, and Test
- provides a comprehensive programming and configuration model for modern Java-based enterprise applications
- its key element is infrastructural support at the application level:
  - it focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments



https://docs.spring.io/spring/docs/

https://spring.io/guides

https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/overview.html

Spring vs Spring Boot

- Spring
  - an application development framework simplify the Java EE development
  - provides comprehensive infrastructure support for developing Java applications
- Spring Boot
  - an extension of the Spring framework that eliminates the boilerplate configurations required for setting up a Spring application.

	Spring	Spring Boot
Configuration	A developer have to set up Hibernate data source, Entity Manager, Session Factory, Transaction Management etc. manually	A developer doesn't need to define everything individually, `SpringBootConfiguration` annotation is enough to manage everything at the time of deployment.
XML	In Spring MVC application some of the XML definition are mandatory to manage	Configuring `Spring Boot Application` can be done by the use of Java annotations
Controlling	As configuration can be easily manually handled, so Spring or Spring MVC can manage not load some of the unwanted default features for that specific application	In case of Spring Boot loading part is done automatically and by default, so the developer don't care about potentially not loading specific unusable spring default features
Use	Better to use if application type or characteristics are purely defined	Better to use when all the futures of application are not properly defined as integrating any Spring specific feature will be auto-configured

https://www.educba.com/spring-vs-spring-boot/

Spring vs Spring Boot

- **Convention over Configuration**
  It's a software design paradigm used by many software frameworks/systems that provides sensible defaults to its user obviously by following the best practices and without losing flexibility.
- The idea is that system/framework would provide sensible defaults for their users [by convention] and if one deviates/departs from the these defaults then only one needs to make any configuration changes.

- **Rapid Application Development**
  Maximize the code that actually adds the value or is related to the domain while reducing the boilerplate code.
- The idea is to reuse such features as serialization to/from XML or JSON because these have no direct values for customer nor developer.

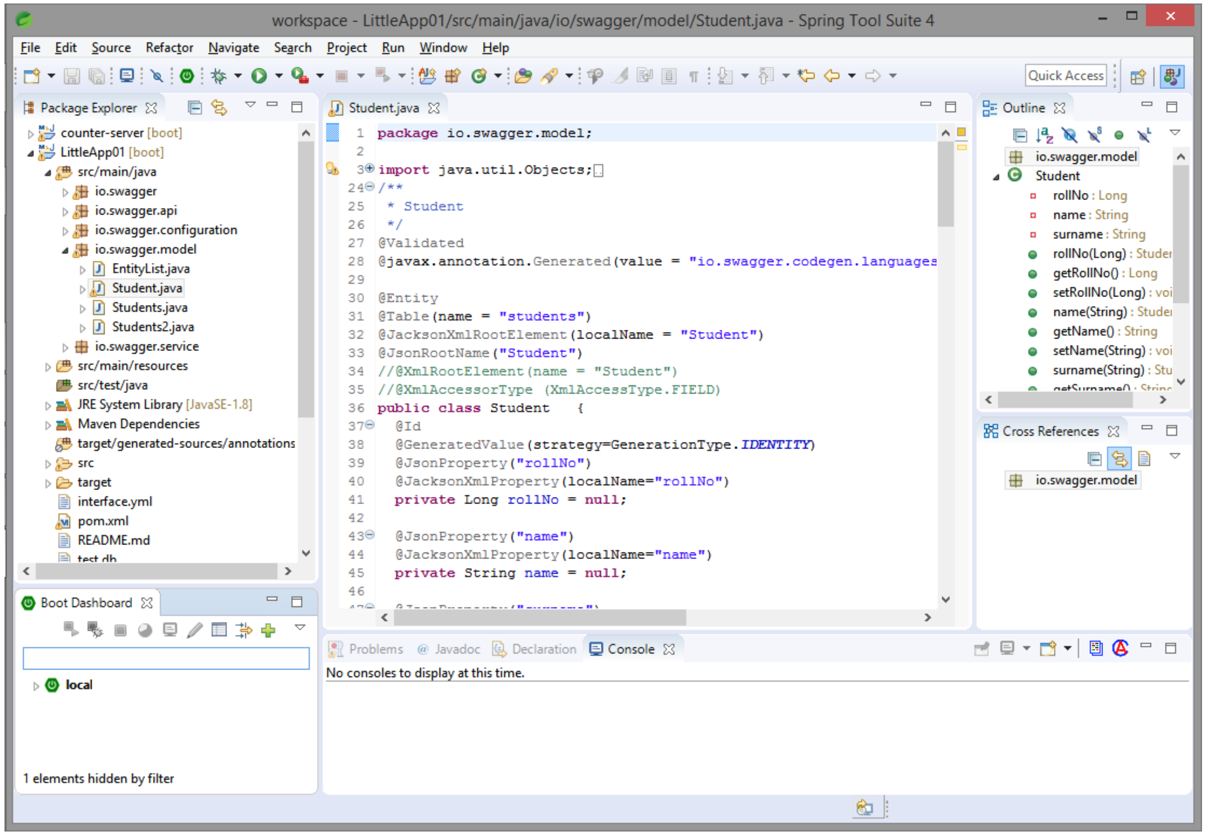
- ***Example 1: (Deployment Simplified)***
  Creating a web application following Spring MVC the user will need to set up a container like Tomcat to deploy it. But such container can be delivered by the framework, so wasting time & effort on installation and configuring of container instance vanishes. And the framework can also be flexible to cat of such kind of support.
  Work For You : Look out for the dependent jars of **spring-boot-starter-web** artifactId
  *<dependency><groupId>org.springframework.boot</groupId><artifactId>**spring-boot-starter-web**</artifactId></dependency>*

- **Example 2: (*Dependency Management Simplified*)**
  Developing enterprise application using Spring Framework (traditional way) causes a headache while finding right jars, right versions of jars, upgrading the version of jars etc. But all dependent jars along with their transitive dependencies can be delivered out of the box by simple selection of predefined sets (web or security or jpa etc.)
  Work For You: Check out Spring Boot Starter Packs like **spring-boot-starter-web, spring-boot-starter-actuator** etc

https://www.quora.com/What-is-the-difference-between-Spring-Boot-and-the-Spring-framework

Spring Tool Suite™

- customized all-in-one Eclipse based distribution that makes application development easy
- provide ready-to-use combinations of language support, framework support, and runtime support, and combine them with the existing Java, Web and Java EE tooling from Eclipse



```java
package io.swagger.model;

import java.util.Objects;
/**
 * Student
 */
@Validated
@javax.annotation.Generated(value = "io.swagger.codegen.languages

@Entity
@Table(name = "students")
@JacksonXmlRootElement(localName = "Student")
@JsonRootName("Student")
//@XmlRootElement(name = "Student")
//@XmlAccessorType (XmlAccessType.FIELD)
public class Student   {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @JsonProperty("rollNo")
  @JacksonXmlProperty(localName="rollNo")
  private Long rollNo = null;

  @JsonProperty("name")
  @JacksonXmlProperty(localName="name")
  private String name = null;
```

No consoles to display at this time.

The assisting examples were implemented using STS (!!!)

https://spring.io/tools

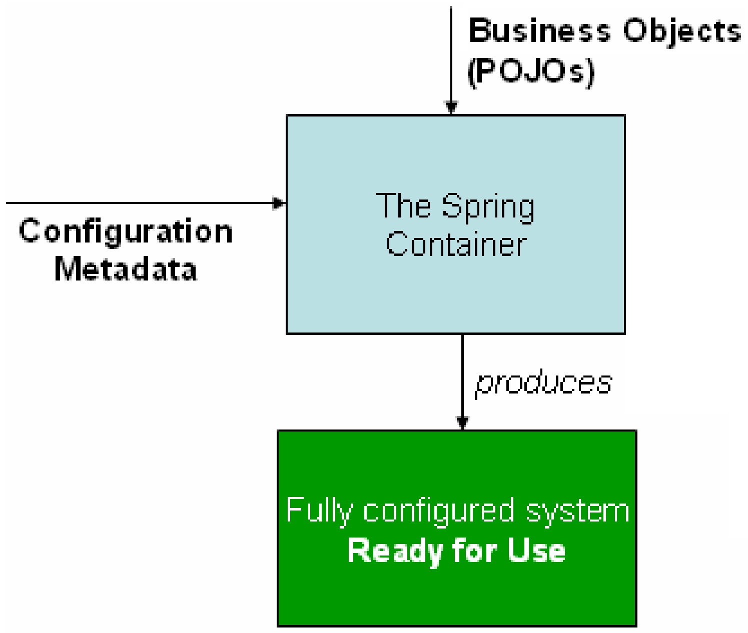
Spring IoC (*Inversion of Control*)

- Inversion of Control
  - most often used in the context of object-oriented programming
  - relates to principle where the control of objects or portions of a program is transferred to a **container** or **framework**
  - can be achieved through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and **Dependency Injection** (DI)
  - in this approach frameworks use abstractions with additional behavior built in (to add behaviors one need to extend the classes of the framework or plugin custom classes)

https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Spring container

- the core of the Spring Framework
- creates the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction
- the objects created from this process are called **Spring beans**
- uses DI (dependency injection) to manage the components that make up an application
- DI can be done through constructors, setters or fields
- configuration metadata are supplied in
  - XML format or
  - Java annotations or
  - Java code.



https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#spring-core

DI example

```
// traditional way - initialize field inside constructor
public class Store {
    private Item item;

    public Store() {
        item = new ItemImpl1();
    }
}

//DI way - let initialization to be done elsewhere, see next
public class Store {
    private Item item;
    public Store(Item item) {
        this.item = item;
    }
}
```

see: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

DI example

```
// instantiate a container
ApplicationContext context
= new ClassPathXmlApplicationContext("applicationContext.xml");
```

```
// constructor-based DI
@Configuration
public class AppConfig {
                                 .
    @Bean
    public Item item1() {
        return new ItemImpl1();
    }

    @Bean
    public Store store() {
        return new Store(item1());
    }
}
```

```
// setter-based DI
@Bean
public Store store() {
    Store store = new Store();
    store.setItem(item1());
    return store;
}
```

```
// field-based DI
public class Store {
    @Autowired
    private Item item;
}
```

see: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

DI example

- Please study sources inside simple-bean project (jdk1.8)
  - http://tomasz.kubik.staff.iiar.pwr.wroc.pl/dydaktyka/InformationSystemsModeling/2019/simple-beans.zip

Bean life cycle

Bean construction

1. The container finds the bean's definition and instantiates the bean.
2. Using dependency injection, Spring populates all of the properties as specified in the bean definition.
3. If the bean implements the `BeanNameAware` interface, the factory calls `setBeanName()` passing the bean's ID.
4. If the bean implements the `BeanFactoryAware` interface, the factory calls `setBeanFactory()`, passing an instance of itself.
5. If there are any `BeanPostProcessors` associated with the bean, their `postProcessBeforeInitialization()` methods will be invoked.
6. If an init-method is specified for the bean, it will be called.
7. Finally, if there are any `BeanPostProcessors` associated with the bean, their `postProcessAfterInitialization()` methods will be invoked.

Bean destruction

1. If the bean implements the `DisposableBean` interface, the `destroy()` method is called.
2. If a custom destroy() method is specified, it will be invoked.

Context

- Spring comes with several flavors of application context. Here are a few that you'll most likely encounter:
  - `AnnotationConfigApplicationContext`
    - Loads a Spring application context from one or more Java-based configuration classes
  - `AnnotationConfigWebApplicationContext`
    - Loads a Spring web application context from one or more Java-based configuration classes
  - `ClassPathXmlApplicationContext`
    - Loads a context definition from one or more XML files located in the classpath, treating context-definition files as classpath resources
  - `FileSystemXmlApplicationContext`
    - Loads a context definition from one or more XML files in the filesystem
  - `XmlWebApplicationContext`
    - Loads context definitions from one or more XML files contained in a web application

Bean lifecycle

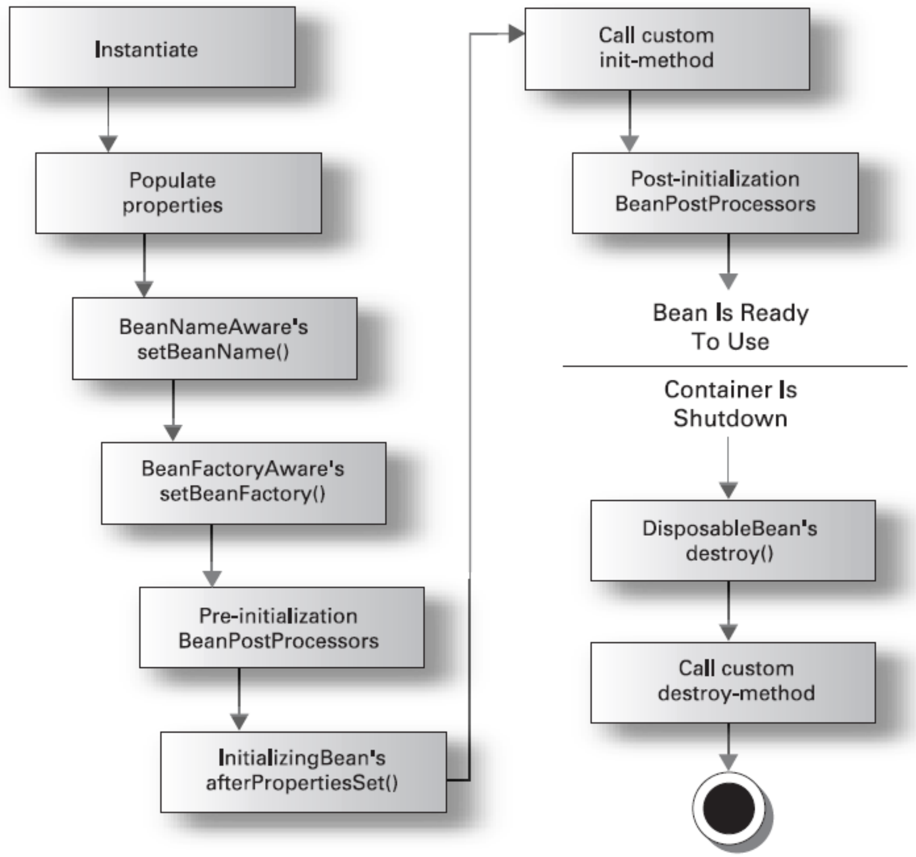

Instantiate
Populate properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean Is Ready To Use
Container Is Shutdown
DisposableBean's destroy()
Call custom destroy-method

Figure 2.1 **The life cycle of a bean within a Spring bean factory container**

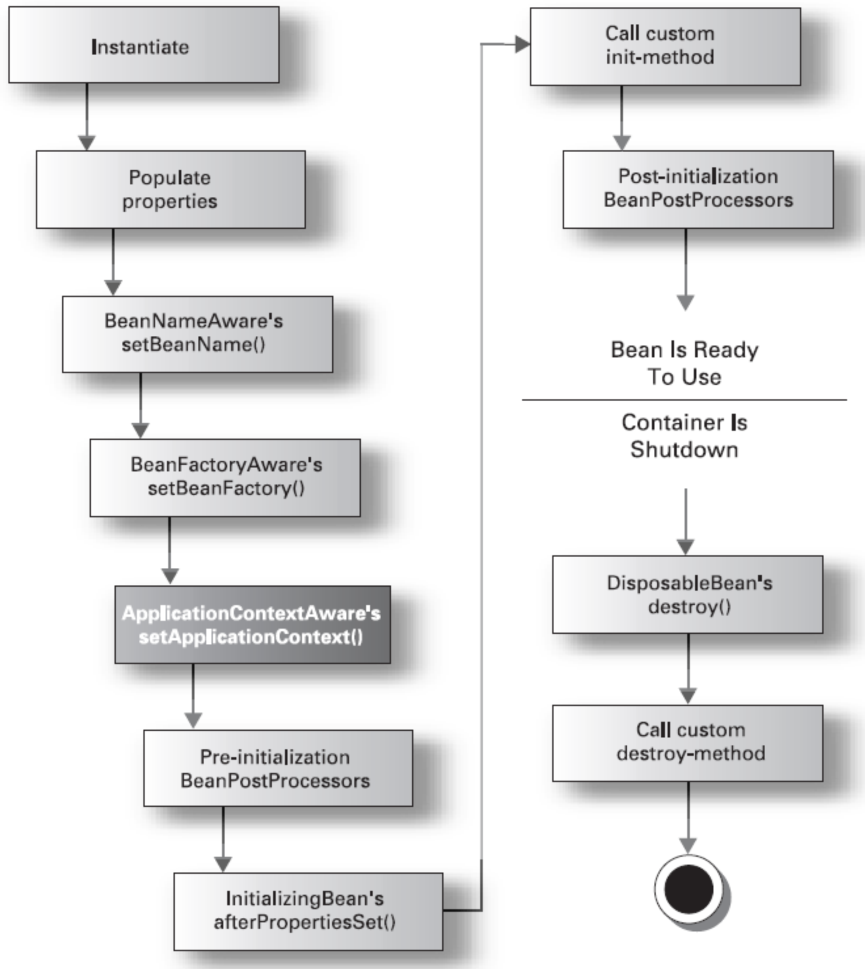

Instantiate
Populate properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext()
Pre-initialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean Is Ready To Use
Container Is Shutdown
DisposableBean's destroy()
Call custom destroy-method

Figure 2.2 **The life cycle of a bean in a Spring application context**

CRAIG WALLS, RYAN BREIDENBACH: Spring in Action, MANNING, 2005

Spring design patterns

- Dependency injection/ or IOC (inversion of control)
  - the main principle behind decoupling process.
- Factory
  - Spring uses factory pattern to create objects of beans using ApplicationContext reference.
- Proxy
  - used heavily in AOP & remoting.
- Singleton
  - By default all beans are singletons. Therefore no matter how many calls will be made using getBean() method, the object received will be created only once. This can be overridden by changing the bean scope to Prototype. Then a new bean will be created for every request.
- Model View Controller
  - The advantage with Spring MVC is that your controllers are POJOs as opposed to being servlets. This makes for easier testing of controllers.
  - The controller is only required to return a logical view name, and the view selection is left to a separate View-Resolver. This makes it easier to reuse controllers for different view technologies.
- Front Controller
  - Spring provides Dispatcher-Servlet to ensure an incoming request gets dispatched to your controllers.
- View Helper
  - Spring has several custom JSP tags, and velocity macros, to assist in separating code from presentation in views.
- Template method
  - used extensively to deal with boilerplate repeated code (such as closing connections cleanly, etc.). For example JdbcTemplate, JmsTemplate, JpaTemplate.

`http://www.javapathshala.com/architectures/design-patterns/springdesignpatterns/`

Spring annotation

Core Spring Framework Annotations https://springframework.guru/spring-framework-annotations/

- @Required
  - applied on bean setter methods and indicates that the affected bean must be populated at configuration time with the required property Otherwise an exception of type BeanInitializationException is thrown.
- @Autowired
  - applied on fields, setter methods, and constructors; injects object dependency implicitly.
- @Qualifier
  - used along with @Autowired annotation and offer more control of the dependency injection process; can be specified on individual constructor arguments or method parameters; used to avoid confusion which occurs when more than one bean of the same type was created and only of them should be wired with a property.
- @Configuration
  - used on classes which define beans; classes will have methods annotated with @Bean to instantiate and configure dependencies
- @ComponentScan
  - used with @Configuration annotation to allow Spring to know the packages to scan for annotated components
- @Bean
  - used at the method level; works with @Configuration to create Spring beans; the method annotated works as bean ID and it creates and returns the actual bean.
- @Lazy
  - used on component classes to declare lazy initialization of bean (by default all autowired dependencies are created and configured at startup)
- @Value
  - used at the field, constructor parameter, and method parameter level; indicates a default value expression for the field or parameter to initialize the property with

Spring annotation

Spring Framework Stereotype Annotations

https://springframework.guru/spring-framework-annotations/

- @Controller
  - used on classes to indicate their role as controllers; the proper classpath and auto-registering bean definitions for them marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context
- @Controller
  - used to indicate the class is a Spring controller; can be used to identify controllers for Spring MVC or Spring WebFlux
- @Service
  - used on a class that performs some service, such as execute business logic, perform calculations and call external APIs.
- @Repository
  - used on Java classes which directly access the database
  - works as marker for any class that fulfills the role of repository or Data Access Object.

Spring Boot Annotations

- @EnableAutoConfiguration
  - usually placed on the main application class.
  - implicitly defines a base "search package" and tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- @SpringBootApplication
  - used on the application class while setting up a Spring Boot project
  - annotated class must be kept in the base package (it will scan only its sub-packages)
  - adds all the following:
    - @Configuration, @EnableAutoConfiguration, @ComponentScan