Software Engineering Design
Spring Code Tracing

第四組 陳韋傑

Code Tracing

spring-framework/spring-beans/factory/

support/: Support classes for bean factory related operations

Major Component

Total Line of Code: 6137

- *BeanFactory*
    - *AbstractBeanFactory, AbstractAutowireCapableBeanFactory*
- *BeanDefinition*
    - *RootBeanDefinition, GenericBeanDefinition, ChildBeanDefinition*
- Related classes
    - *BeanDefinitionBuilder, BeanDefinitionReader, BeanDefinitionRegistry…*

Preliminary Knowledge of Spring

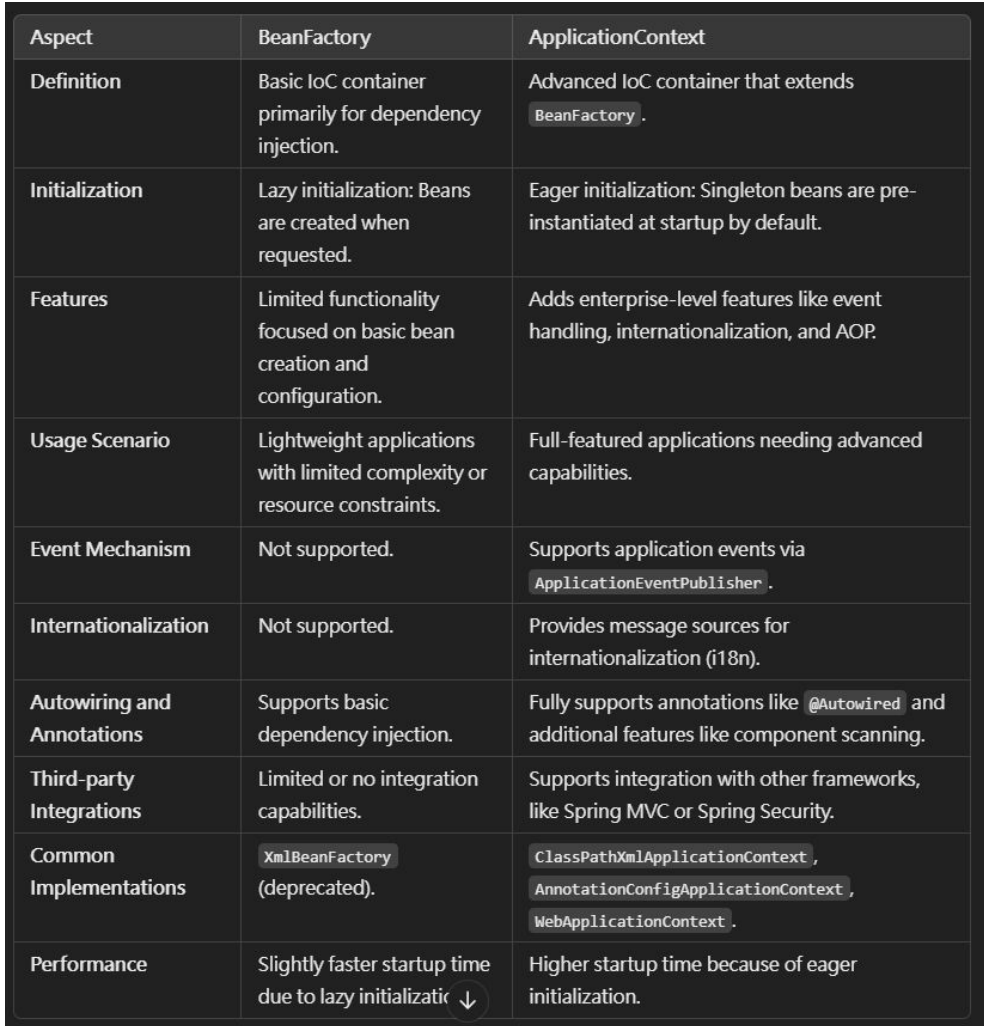
- *Bean*: Basic Java objects in Spring
- *BeanFactory*: Basic IoC container
    - It creates, configures and manages the life cycle of *Bean*
- *BeanDefinition*: Definition & configuration of *Bean*
- *ApplicationContext*: Advanced IoC container (extended from *BeanFactory*)
    - Which is also a main entry point when building a application using Spring
    - Supports automatic bean post-processing, annotation-based configuration etc.

Comparison

BeanFactory

vs

ApplicationContext



Aspect	BeanFactory	ApplicationContext
Definition	Basic IoC container primarily for dependency injection.	Advanced IoC container that extends `BeanFactory`.
Initialization	Lazy initialization: Beans are created when requested.	Eager initialization: Singleton beans are pre-instantiated at startup by default.
Features	Limited functionality focused on basic bean creation and configuration.	Adds enterprise-level features like event handling, internationalization, and AOP.
Usage Scenario	Lightweight applications with limited complexity or resource constraints.	Full-featured applications needing advanced capabilities.
Event Mechanism	Not supported.	Supports application events via `ApplicationEventPublisher`.
Internationalization	Not supported.	Provides message sources for internationalization (i18n).
Autowiring and Annotations	Supports basic dependency injection.	Fully supports annotations like `@Autowired` and additional features like component scanning.
Third-party Integrations	Limited or no integration capabilities.	Supports integration with other frameworks, like Spring MVC or Spring Security.
Common Implementations	`XmlBeanFactory` (deprecated).	`ClassPathXmlApplicationContext`, `AnnotationConfigApplicationContext`, `WebApplicationContext`.
Performance	Slightly faster startup time due to lazy initializatic	Higher startup time because of eager initialization.

Classes and Their Design Patterns

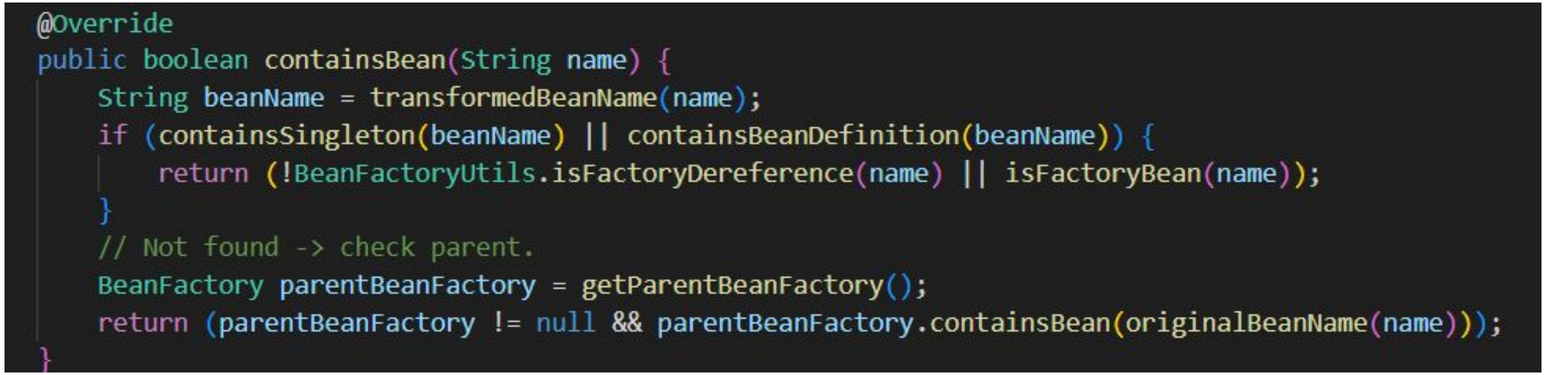
AbstractBeanFactory

This class provides bean instance retrieval, bean type (singleton/prototype) determination, bean definition merging, bean destruction, bean pre/post process etc.

Design Pattern: Chain of Responsibility / Delegation

#containsBean()

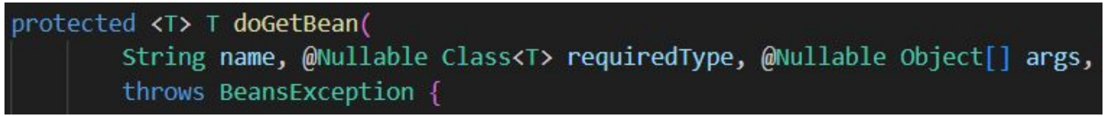
- Check if current instance have the specified bean, if not, delegate this job to its *parentBeanFactory*, until the job is done.
- Both the class itself and *parentBeanFactory* are *BeanFactory*.



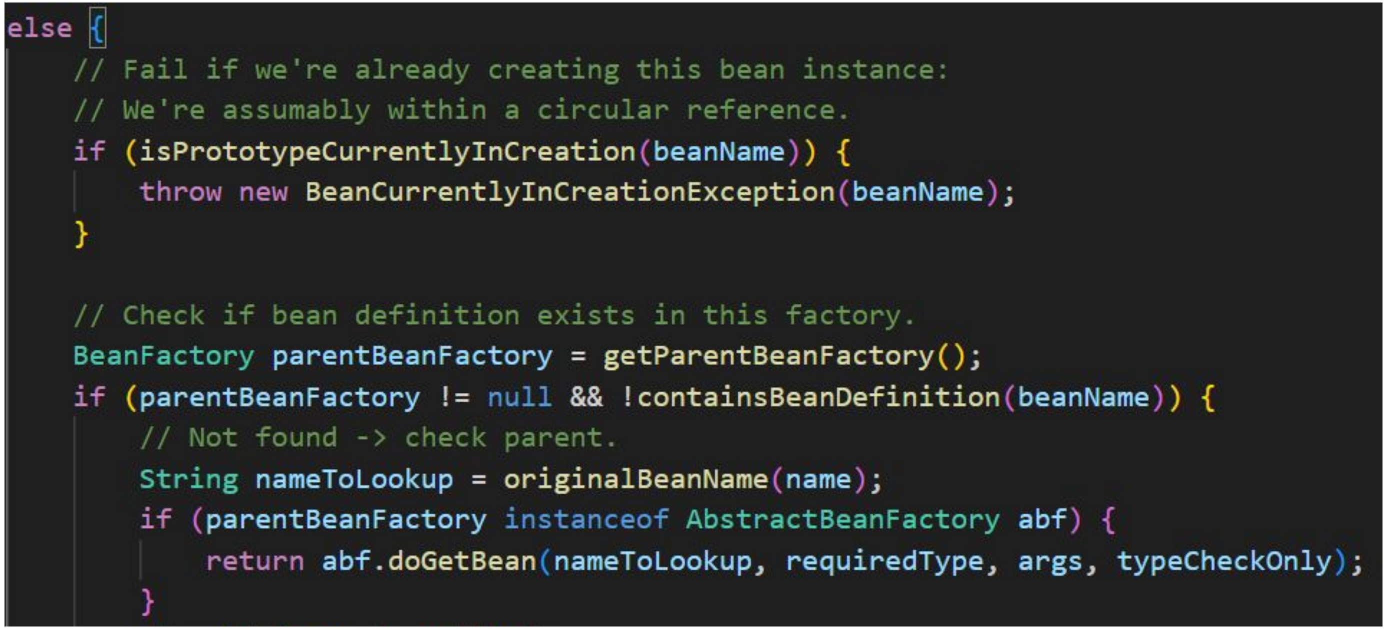
```
@Override
public boolean containsBean(String name) {
    String beanName = transformedBeanName(name);
    if (containsSingleton(beanName) || containsBeanDefinition(beanName)) {
        return (!BeanFactoryUtils.isFactoryDereference(name) || isFactoryBean(name));
    }
    // Not found -> check parent.
    BeanFactory parentBeanFactory = getParentBeanFactory();
    return (parentBeanFactory != null && parentBeanFactory.containsBean(originalBeanName(name)));
}
```

Design Pattern: Chain of Responsibility / Delegation

#doGetBean()

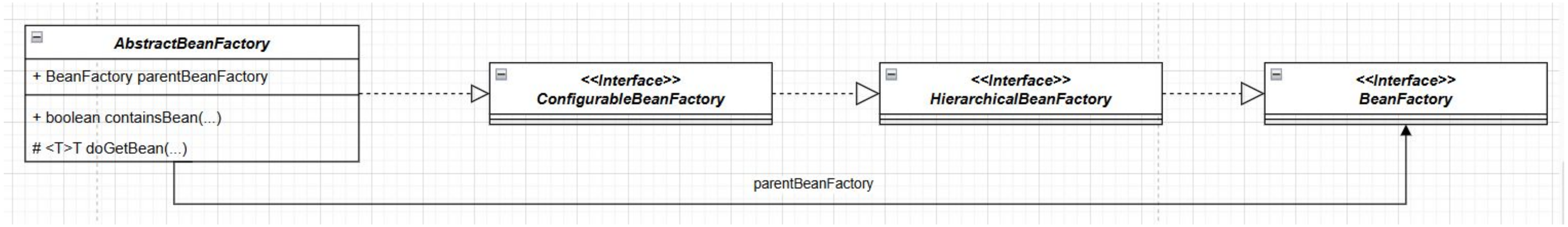


```java
protected <T> T doGetBean(
        String name, @Nullable Class<T> requiredType, @Nullable Object[] args,
        throws BeansException {
```



```java
else {
    // Fail if we're already creating this bean instance:
    // We're assumably within a circular reference.
    if (isPrototypeCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }

    // Check if bean definition exists in this factory.
    BeanFactory parentBeanFactory = getParentBeanFactory();
    if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
        // Not found -> check parent.
        String nameToLookup = originalBeanName(name);
        if (parentBeanFactory instanceof AbstractBeanFactory abf) {
            return abf.doGetBean(nameToLookup, requiredType, args, typeCheckOnly);
        }
```

Design Pattern: Chain of Responsibility / Delegation

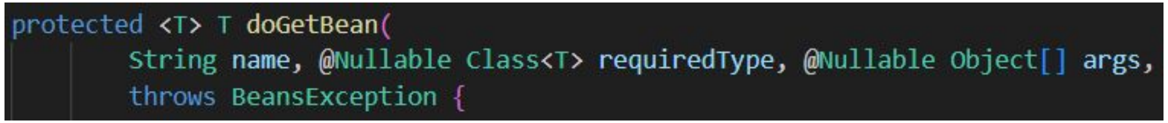


Design Pattern: Adapter

#doGetBean()

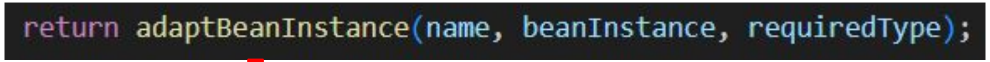
- At the end of the function, the result will be passed to #*adaptBeanInstance()* before return to caller, to adapt it to *requiredType*.
- But #*adaptBeanInstance()* is just a function, not a class.

Design Pattern: Adapter

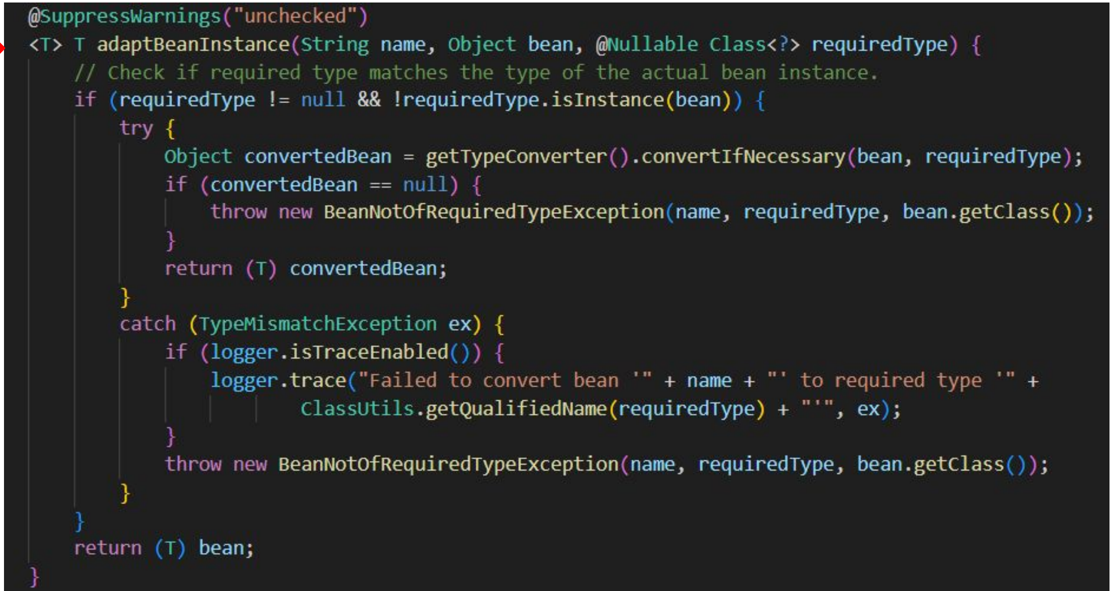


```java
protected <T> T doGetBean(
        String name, @Nullable Class<T> requiredType, @Nullable Object[] args,
        throws BeansException {
```

…

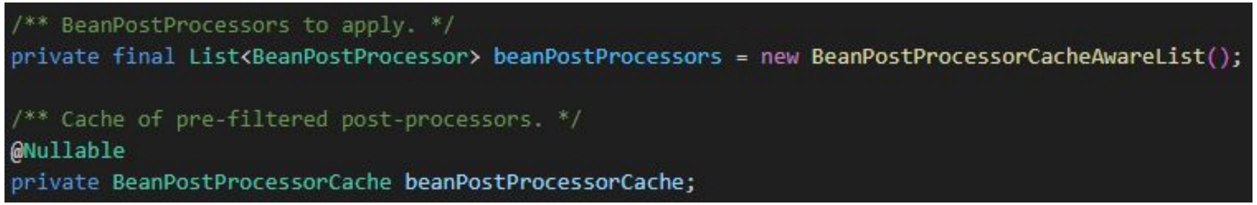


```java
return adaptBeanInstance(name, beanInstance, requiredType);
```

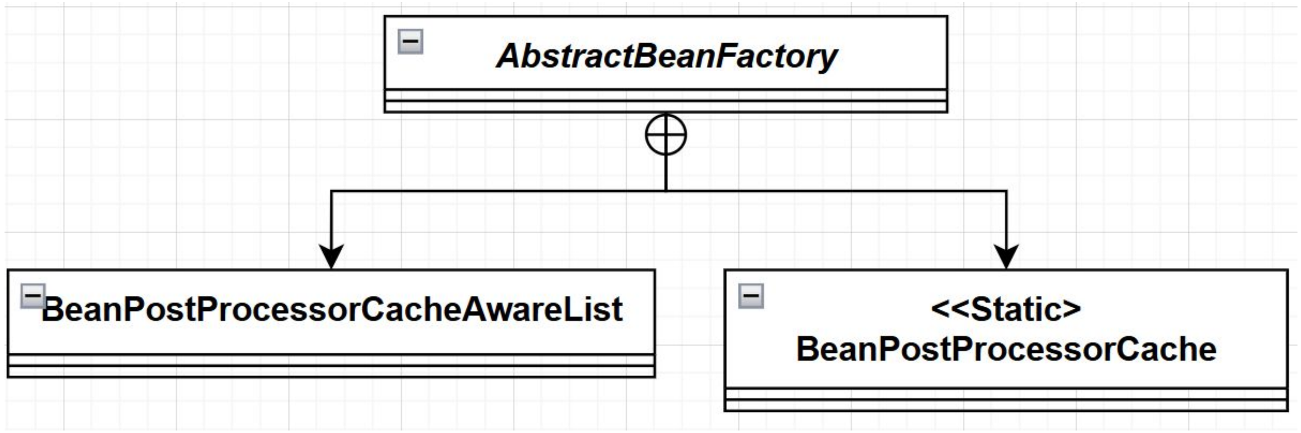


```java
@SuppressWarnings("unchecked")
<T> T adaptBeanInstance(String name, Object bean, @Nullable Class<?> requiredType) {
    // Check if required type matches the type of the actual bean instance.
    if (requiredType != null && !requiredType.isInstance(bean)) {
        try {
            Object convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);
            if (convertedBean == null) {
                throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
            }
            return (T) convertedBean;
        }
        catch (TypeMismatchException ex) {
            if (logger.isTraceEnabled()) {
                logger.trace("Failed to convert bean '" + name + "' to required type '" +
                        ClassUtils.getQualifiedName(requiredType) + "'", ex);
            }
            throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
        }
    }
    return (T) bean;
}
```

Design Pattern: Lazy Initialization (w/ Thread-safe Protection)



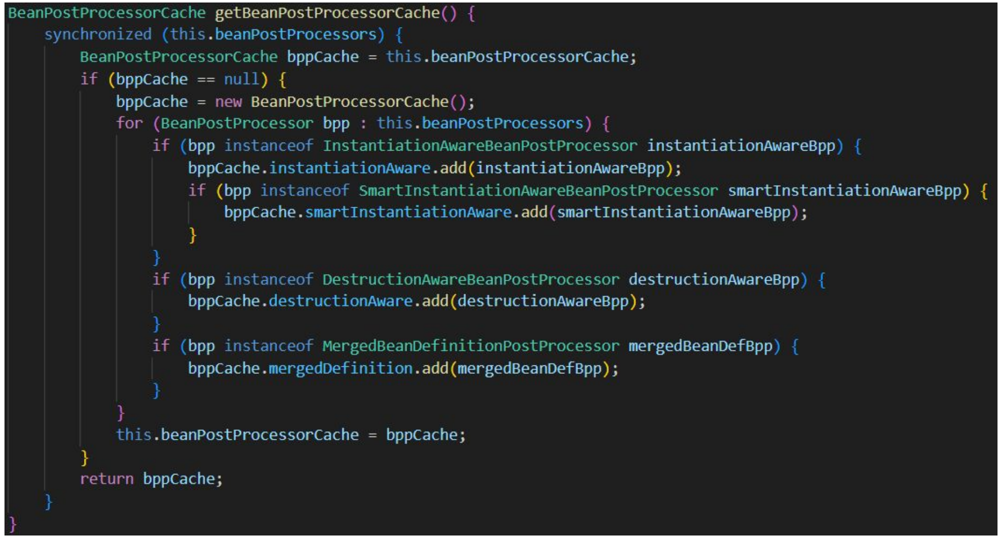
```java
/** BeanPostProcessors to apply. */
private final List<BeanPostProcessor> beanPostProcessors = new BeanPostProcessorCacheAwareList();

/** Cache of pre-filtered post-processors. */
@Nullable
private BeanPostProcessorCache beanPostProcessorCache;
```



Design Pattern: Lazy Initialization (w/ Thread-safe Protection)

#getBeanPostProcessorCache()

- Synchronized on *this.beanPostProcessors*, if *this.beanPostProcessorCache* is null, initialize it with objects in *this.beanPostProcessor*.



```
BeanPostProcessorCache getBeanPostProcessorCache() {
    synchronized (this.beanPostProcessors) {
        BeanPostProcessorCache bppCache = this.beanPostProcessorCache;
        if (bppCache == null) {
            bppCache = new BeanPostProcessorCache();
            for (BeanPostProcessor bpp : this.beanPostProcessors) {
                if (bpp instanceof InstantiationAwareBeanPostProcessor instantiationAwareBpp) {
                    bppCache.instantiationAware.add(instantiationAwareBpp);
                    if (bpp instanceof SmartInstantiationAwareBeanPostProcessor smartInstantiationAwareBpp) {
                        bppCache.smartInstantiationAware.add(smartInstantiationAwareBpp);
                    }
                }
                if (bpp instanceof DestructionAwareBeanPostProcessor destructionAwareBpp) {
                    bppCache.destructionAware.add(destructionAwareBpp);
                }
                if (bpp instanceof MergedBeanDefinitionPostProcessor mergedBeanDefBpp) {
                    bppCache.mergedDefinition.add(mergedBeanDefBpp);
                }
            }
            this.beanPostProcessorCache = bppCache;
        }
        return bppCache;
    }
}
```

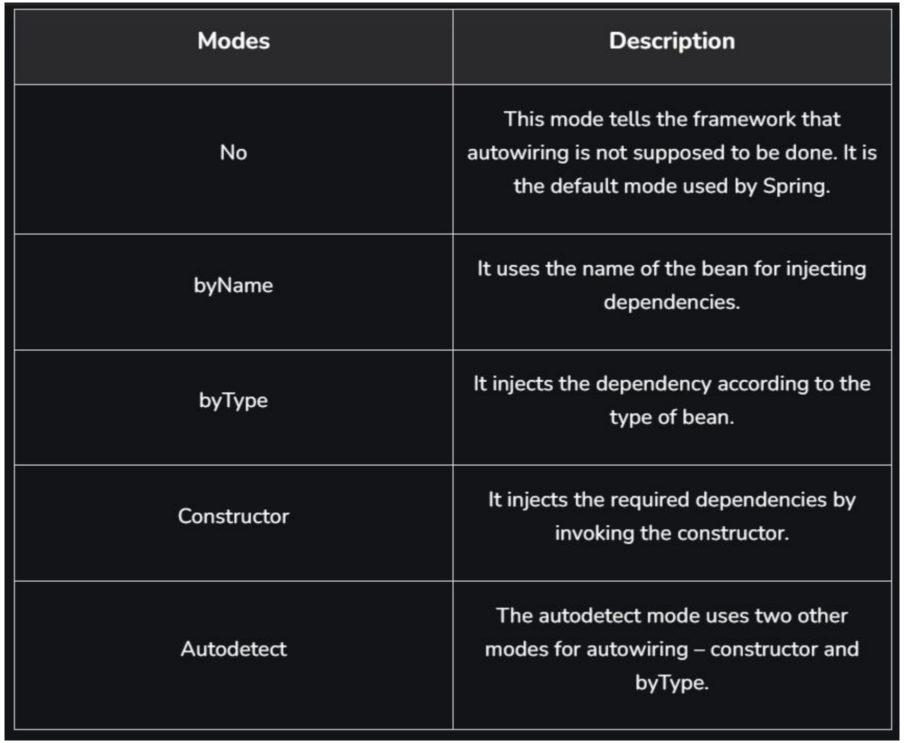
AbstractAutowireCapableBeanFactory

This class provides bean creation (instantiation), wiring and autowiring. This class is extended from *AbstractBeanFactory*.

Autowiring

Autowiring in the Spring framework can inject dependencies automatically.

The Spring container detects those dependencies specified in the configuration file and the relationship between the beans.



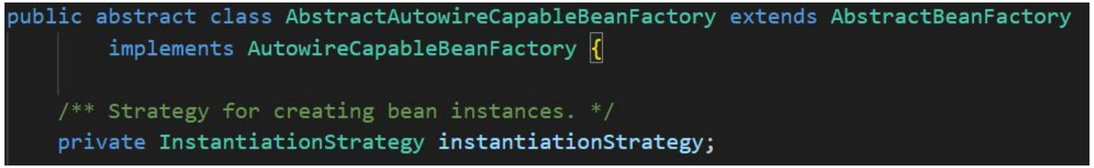
Modes	Description
No	This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.
byName	It uses the name of the bean for injecting dependencies.
byType	It injects the dependency according to the type of bean.
Constructor	It injects the required dependencies by invoking the constructor.
Autodetect	The autodetect mode uses two other modes for autowiring – constructor and byType.

Design Pattern: Strategy

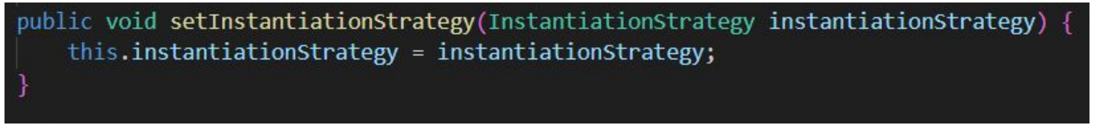
#instantiateBean() & this.instantiationStrategy

- This class holds an attribute of *InstantiationStrategy* to help determine how to instantiate a bean, and use it in other functions when instantiation is needed.

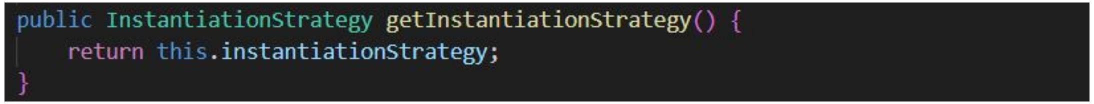
Design Pattern: Strategy



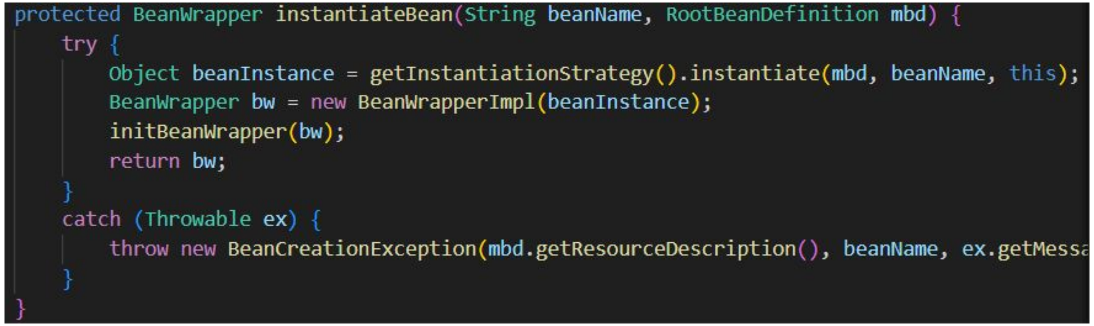
```java
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactory
        implements AutowireCapableBeanFactory {

    /** Strategy for creating bean instances. */
    private InstantiationStrategy instantiationStrategy;
```



```java
public void setInstantiationStrategy(InstantiationStrategy instantiationStrategy) {
    this.instantiationStrategy = instantiationStrategy;
}
```

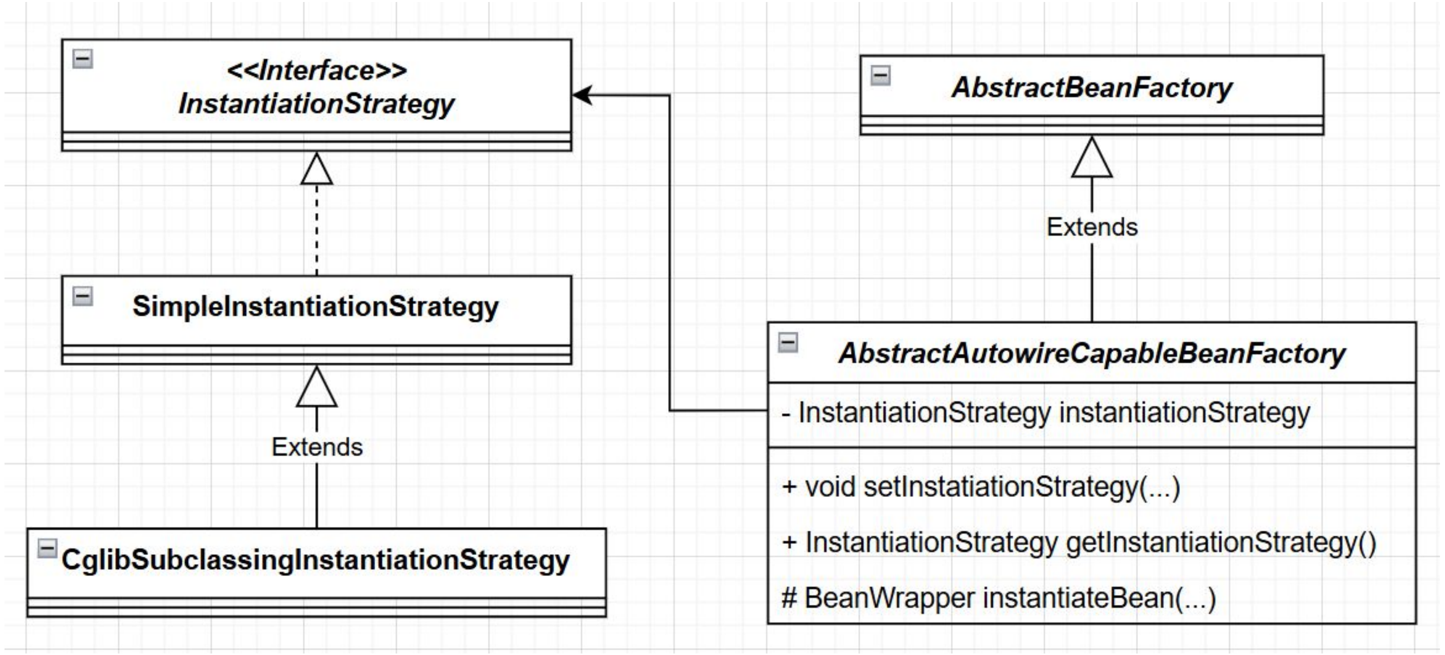


```java
public InstantiationStrategy getInstantiationStrategy() {
    return this.instantiationStrategy;
}
```



```java
protected BeanWrapper instantiateBean(String beanName, RootBeanDefinition mbd) {
    try {
        Object beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, this);
        BeanWrapper bw = new BeanWrapperImpl(beanInstance);
        initBeanWrapper(bw);
        return bw;
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName, ex.getMessa
    }
}
```

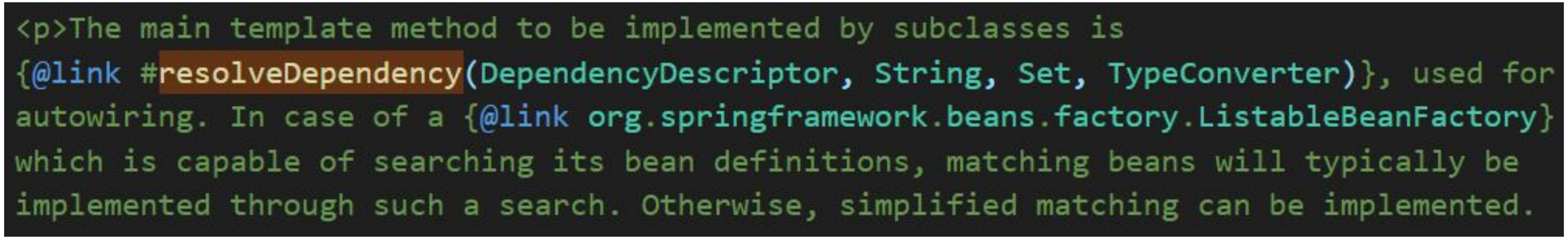
Design Pattern: Strategy


<<Interface>>
InstantiationStrategy
AbstractBeanFactory
Extends
SimpleInstantiationStrategy
AbstractAutowireCapableBeanFactory
- InstantiationStrategy instantiationStrategy
Extends
+ void setInstatiationStrategy(...)
+ InstantiationStrategy getInstantiationStrategy()
CglibSubclassingInstantiationStrategy
# BeanWrapper instantiateBean(...)

Design Pattern: Template Method

#autowireByType()

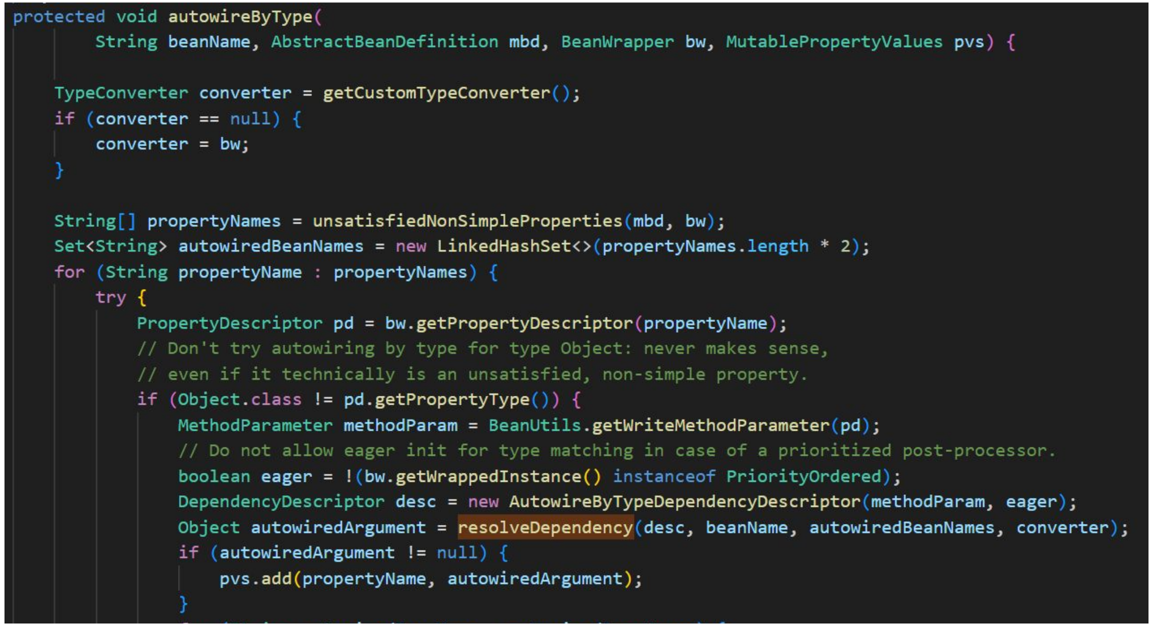
- In this function, *#resolveDependency()* is **used** but not **implemented**.
- Implementation is **deferred** to its subclass (*DefaultListableBeanFactory*).



```
<p>The main template method to be implemented by subclasses is
{@link #resolveDependency(DependencyDescriptor, String, Set, TypeConverter)}, used for
autowiring. In case of a {@link org.springframework.beans.factory.ListableBeanFactory}
which is capable of searching its bean definitions, matching beans will typically be
implemented through such a search. Otherwise, simplified matching can be implemented.
```

Design Pattern: Template Method

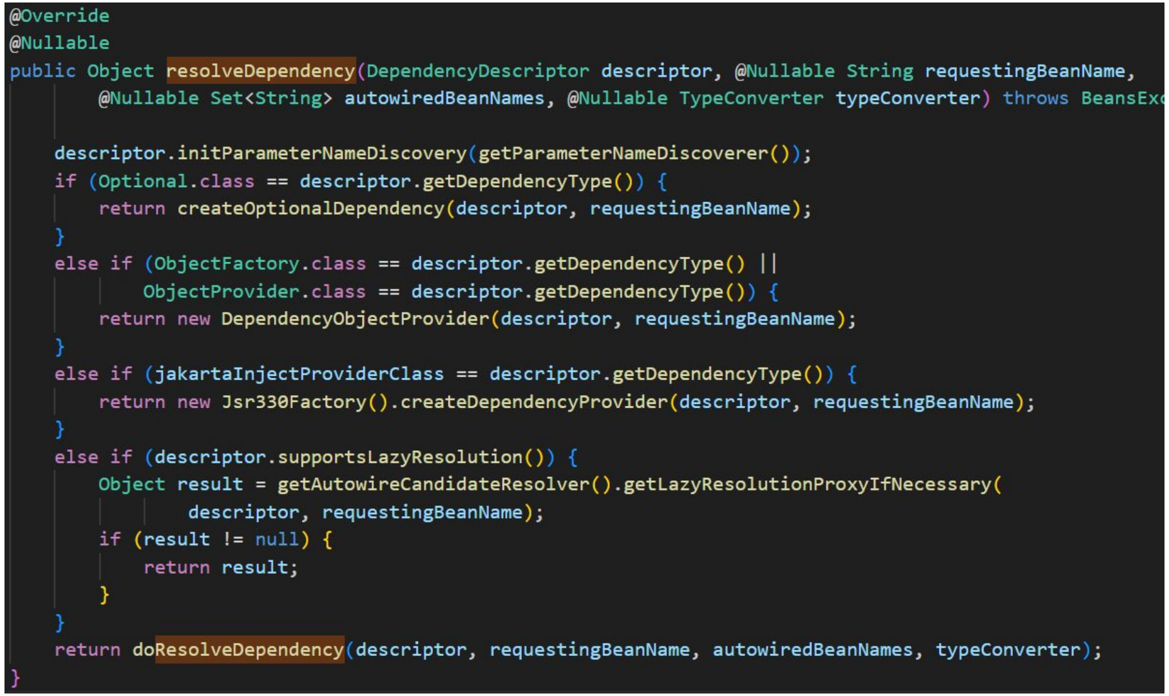
In *AbstractAutowireCapableBeanFactory#autowireByType()*, *#resolveDependency()* is used.



```
protected void autowireByType(
        String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {

    TypeConverter converter = getCustomTypeConverter();
    if (converter == null) {
        converter = bw;
    }

    String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
    Set<String> autowiredBeanNames = new LinkedHashSet<>(propertyNames.length * 2);
    for (String propertyName : propertyNames) {
        try {
            PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
            // Don't try autowiring by type for type Object: never makes sense,
            // even if it technically is an unsatisfied, non-simple property.
            if (Object.class != pd.getPropertyType()) {
                MethodParameter methodParam = BeanUtils.getWriteMethodParameter(pd);
                // Do not allow eager init for type matching in case of a prioritized post-processor.
                boolean eager = !(bw.getWrappedInstance() instanceof PriorityOrdered);
                DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(methodParam, eager);
                Object autowiredArgument = resolveDependency(desc, beanName, autowiredBeanNames, converter);
                if (autowiredArgument != null) {
                    pvs.add(propertyName, autowiredArgument);
                }
```

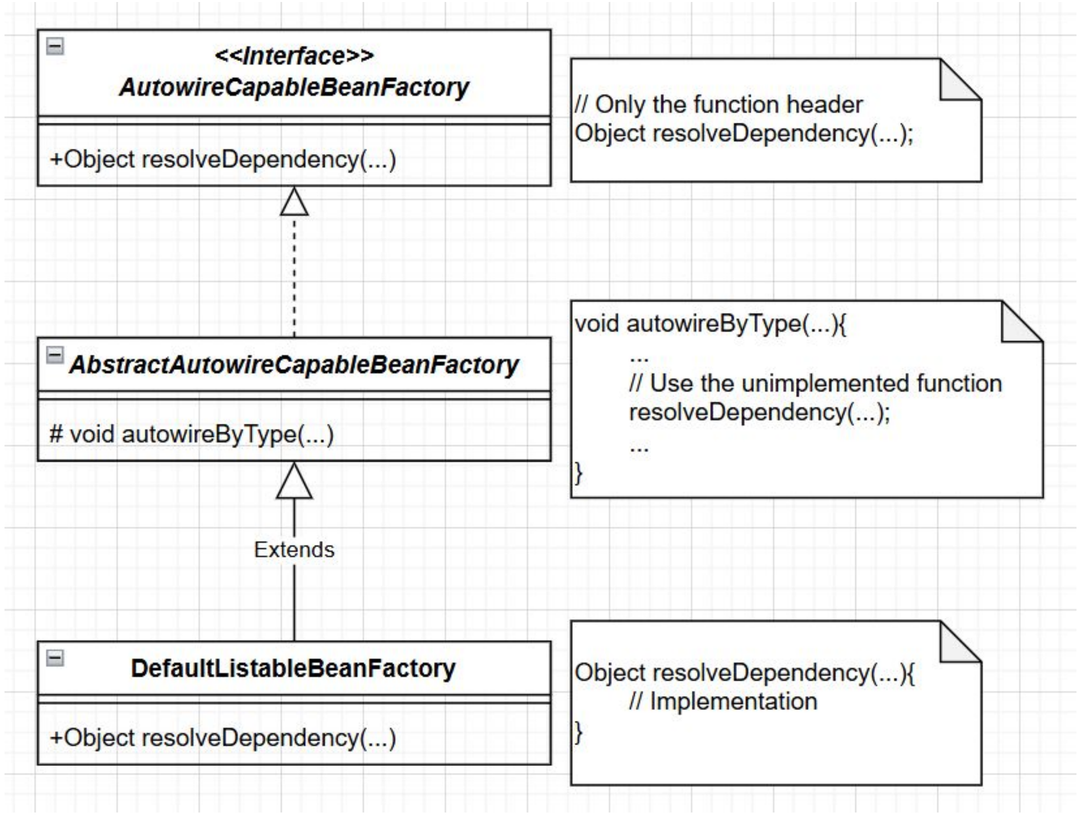
Design Pattern: Template Method

In *DefaultListableBeanFactory*, *#resolveDependency()* is implemented.



```
@Override
@Nullable
public Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requestingBeanName,
        @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter) throws BeansExc

    descriptor.initParameterNameDiscovery(getParameterNameDiscoverer());
    if (Optional.class == descriptor.getDependencyType()) {
        return createOptionalDependency(descriptor, requestingBeanName);
    }
    else if (ObjectFactory.class == descriptor.getDependencyType() ||
            ObjectProvider.class == descriptor.getDependencyType()) {
        return new DependencyObjectProvider(descriptor, requestingBeanName);
    }
    else if (jakartaInjectProviderClass == descriptor.getDependencyType()) {
        return new Jsr330Factory().createDependencyProvider(descriptor, requestingBeanName);
    }
    else if (descriptor.supportsLazyResolution()) {
        Object result = getAutowireCandidateResolver().getLazyResolutionProxyIfNecessary(
                descriptor, requestingBeanName);
        if (result != null) {
            return result;
        }
    }
    return doResolveDependency(descriptor, requestingBeanName, autowiredBeanNames, typeConverter);
}
```

Design Pattern: Template Method


<<Interface>>
AutowireCapableBeanFactory
+Object resolveDependency(...)
AbstractAutowireCapableBeanFactory
# void autowireByType(...)
Extends
DefaultListableBeanFactory
+Object resolveDependency(...)

```
// Only the function header
Object resolveDependency(...);
```

```
void autowireByType(...){
        ...
        // Use the unimplemented function
        resolveDependency(...);
        ...
}
```

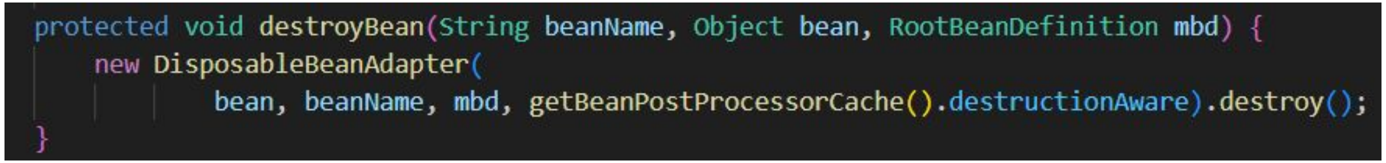
```
Object resolveDependency(...){
        // Implementation
}
```

Design Pattern: Adapter

#destroyBean()

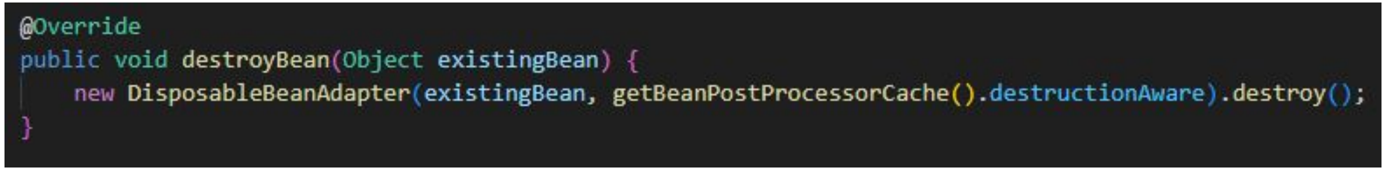
- Use a unified class, *DisposableBeanAdapter,* to help destroy bean for different classes in different settings.

AbstractBeanFactory



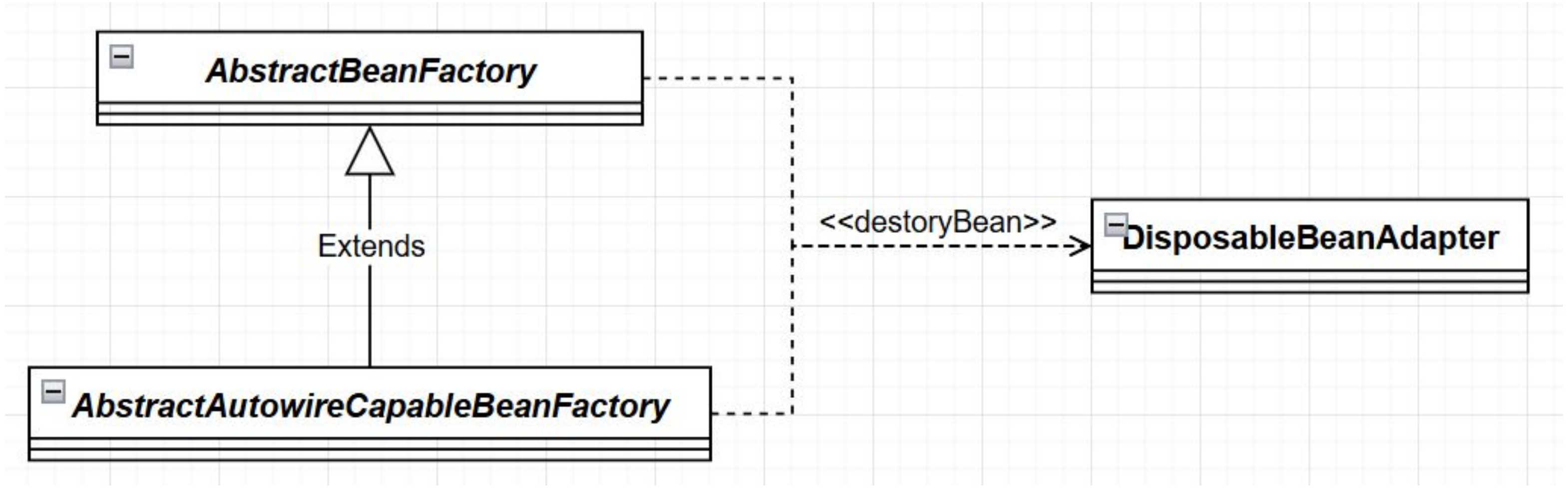
```
protected void destroyBean(String beanName, Object bean, RootBeanDefinition mbd) {
    new DisposableBeanAdapter(
            bean, beanName, mbd, getBeanPostProcessorCache().destructionAware).destroy();
}
```

AbstractAutowireCapableBeanFactory



```
@Override
public void destroyBean(Object existingBean) {
    new DisposableBeanAdapter(existingBean, getBeanPostProcessorCache().destructionAware).destroy();
}
```

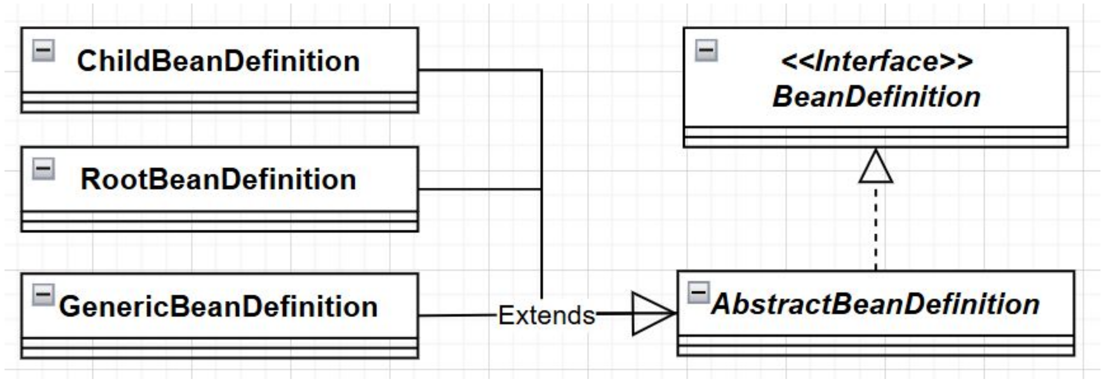
Design Pattern: Adapter



BeanDefinition

Three types of BeanDefinition:

- *ChildBeanDefinition*: Inheriting properties from a parent bean definition
- *RootBeanDefinition*: Fully independent, standalone bean definition
- *GenericBeanDefinition*: Supporting both configurations

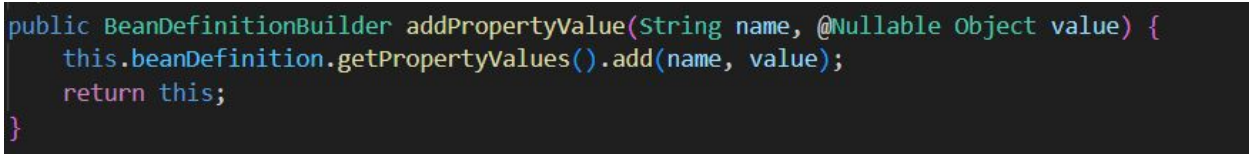


BeanDefinitionBuilder

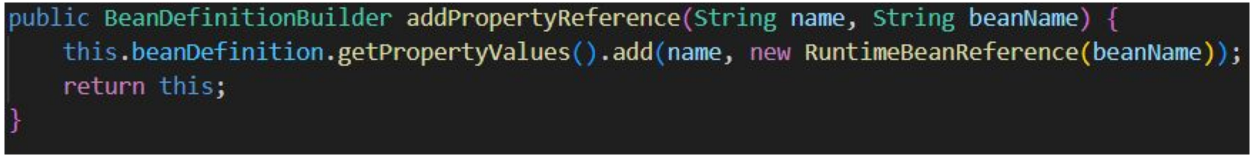
This class builds different types of *BeanDefinition* programmatically.

Design Pattern: Builder & Factory Method (Static)

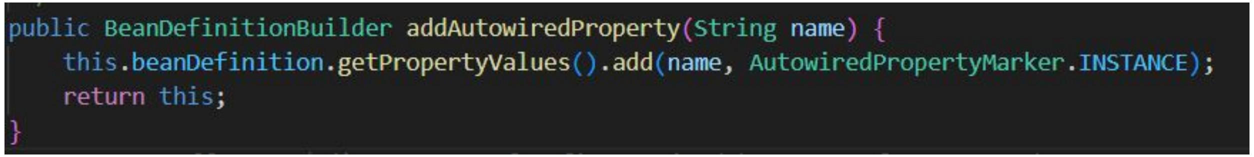
- Builder Pattern: return *this* (the builder itself) to the caller, the caller can thus use it to make another call.



```java
public BeanDefinitionBuilder addPropertyValue(String name, @Nullable Object value) {
    this.beanDefinition.getPropertyValues().add(name, value);
    return this;
}
```



```java
public BeanDefinitionBuilder addPropertyReference(String name, String beanName) {
    this.beanDefinition.getPropertyValues().add(name, new RuntimeBeanReference(beanName));
    return this;
}
```



```java
public BeanDefinitionBuilder addAutowiredProperty(String name) {
    this.beanDefinition.getPropertyValues().add(name, AutowiredPropertyMarker.INSTANCE);
    return this;
}
```

…

Design Pattern: Builder & Factory Method (Static)

All methods return a *BeanDefinitionBuilder* which can be used for following call.

```
public BeanDefinitionBuilder setParentName(String parentName);
public BeanDefinitionBuilder setFactoryMethod(String factoryMethod);
public BeanDefinitionBuilder setFactoryMethodOnBean(String factoryMethod, String factoryBean);
public BeanDefinitionBuilder addConstructorArgValue(@Nullable Object value);
public BeanDefinitionBuilder addConstructorArgReference(String beanName);
public BeanDefinitionBuilder addPropertyValue(String name, @Nullable Object value);
public BeanDefinitionBuilder addPropertyReference(String name, String beanName);
public BeanDefinitionBuilder addAutowiredProperty(String name);
public BeanDefinitionBuilder setInitMethodName(@Nullable String methodName);
public BeanDefinitionBuilder setDestroyMethodName(@Nullable String methodName);
public BeanDefinitionBuilder setScope(@Nullable String scope);
public BeanDefinitionBuilder setAbstract(boolean flag);
public BeanDefinitionBuilder setLazyInit(boolean lazy);
public BeanDefinitionBuilder setAutowireMode(int autowireMode);
public BeanDefinitionBuilder setDependencyCheck(int dependencyCheck);
public BeanDefinitionBuilder addDependsOn(String beanName);
public BeanDefinitionBuilder setPrimary(boolean primary);
public BeanDefinitionBuilder setFallback(boolean fallback);
public BeanDefinitionBuilder setRole(int role);
public BeanDefinitionBuilder setSynthetic(boolean synthetic);
public BeanDefinitionBuilder applyCustomizers(BeanDefinitionCustomizer... customizers);
```

Design Pattern: Builder & Factory Method (Static)

Over 50 classes use *BeanDefinitionBuilder*.
Example usage in *spring-aop/aop/config/ConfigBeanDefinitionParser.java*

Or can use it like

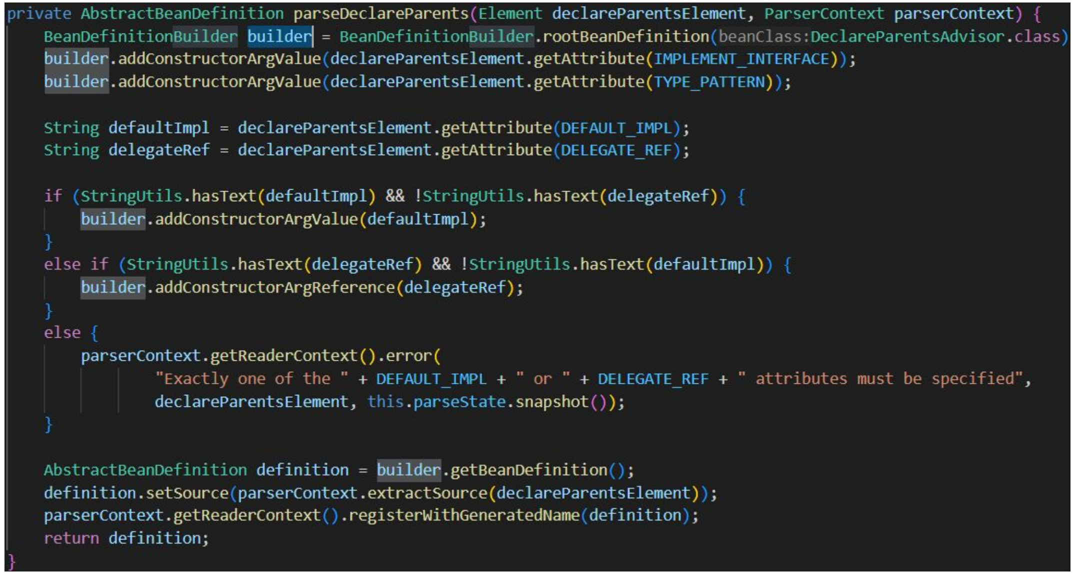
builder.genericBeanDefinition(MyClass.class)

.setScope("prototype")

.setLazyInit(true)

.addPropertyValue("propertyName", value);

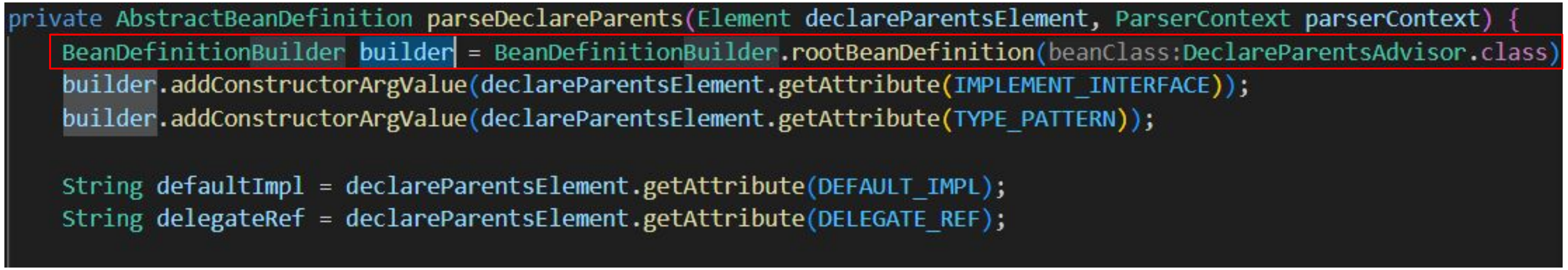
builder.getBeanDefinition();



```
private AbstractBeanDefinition parseDeclareParents(Element declareParentsElement, ParserContext parserContext) {
    BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition(beanClass:DeclareParentsAdvisor.class)
    builder.addConstructorArgValue(declareParentsElement.getAttribute(IMPLEMENT_INTERFACE));
    builder.addConstructorArgValue(declareParentsElement.getAttribute(TYPE_PATTERN));

    String defaultImpl = declareParentsElement.getAttribute(DEFAULT_IMPL);
    String delegateRef = declareParentsElement.getAttribute(DELEGATE_REF);

    if (StringUtils.hasText(defaultImpl) && !StringUtils.hasText(delegateRef)) {
        builder.addConstructorArgValue(defaultImpl);
    }
    else if (StringUtils.hasText(delegateRef) && !StringUtils.hasText(defaultImpl)) {
        builder.addConstructorArgReference(delegateRef);
    }
    else {
        parserContext.getReaderContext().error(
                "Exactly one of the " + DEFAULT_IMPL + " or " + DELEGATE_REF + " attributes must be specified",
                declareParentsElement, this.parseState.snapshot());
    }

    AbstractBeanDefinition definition = builder.getBeanDefinition();
    definition.setSource(parserContext.extractSource(declareParentsElement));
    parserContext.getReaderContext().registerWithGeneratedName(definition);
    return definition;
}
```

Design Pattern: Builder & Factory Method (Static)

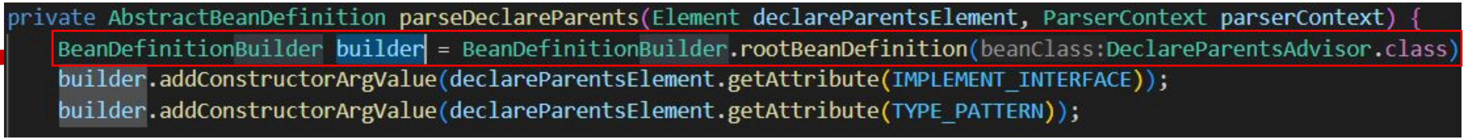
- Factory Method Pattern (Static): Various factory methods for building a *AbstractBeanDefinition*.
- Private constructor: *BeanDefinitionBuilder* instances are instantiable by static factory method.



```
private AbstractBeanDefinition parseDeclareParents(Element declareParentsElement, ParserContext parserContext) {
    BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition(beanClass:DeclareParentsAdvisor.class)
    builder.addConstructorArgValue(declareParentsElement.getAttribute(IMPLEMENT_INTERFACE));
    builder.addConstructorArgValue(declareParentsElement.getAttribute(TYPE_PATTERN));

    String defaultImpl = declareParentsElement.getAttribute(DEFAULT_IMPL);
    String delegateRef = declareParentsElement.getAttribute(DELEGATE_REF);
```

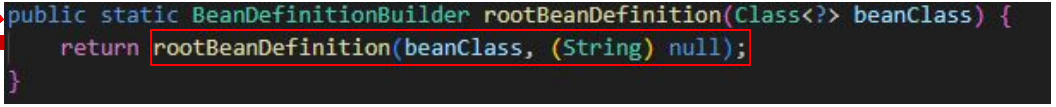
Design Pattern: Builder & Factory Method (Static)

ConfigBeanDefinitionParser.java

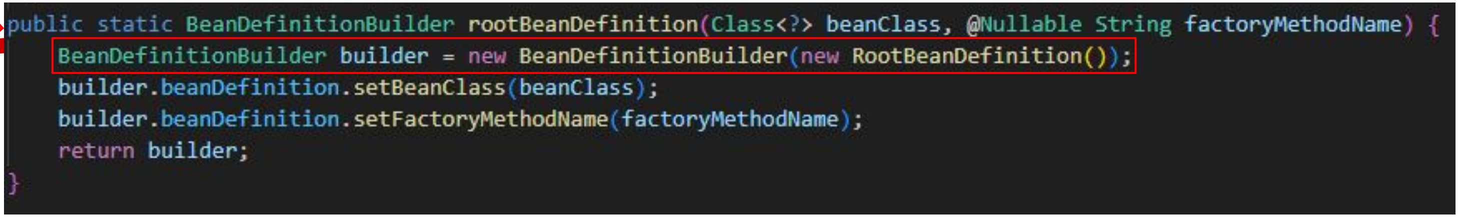


```java
private AbstractBeanDefinition parseDeclareParents(Element declareParentsElement, ParserContext parserContext) {
    BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition(beanClass:DeclareParentsAdvisor.class)
    builder.addConstructorArgValue(declareParentsElement.getAttribute(IMPLEMENT_INTERFACE));
    builder.addConstructorArgValue(declareParentsElement.getAttribute(TYPE_PATTERN));
```

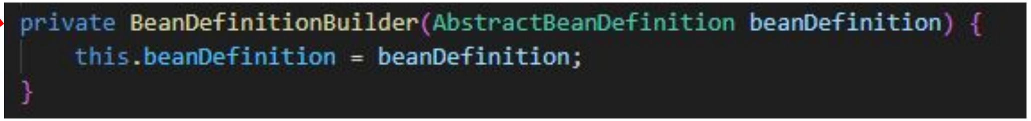
BeanDefinitionBuilder.java



```java
public static BeanDefinitionBuilder rootBeanDefinition(Class<?> beanClass) {
    return rootBeanDefinition(beanClass, (String) null);
}
```



```java
public static BeanDefinitionBuilder rootBeanDefinition(Class<?> beanClass, @Nullable String factoryMethodName) {
    BeanDefinitionBuilder builder = new BeanDefinitionBuilder(new RootBeanDefinition());
    builder.beanDefinition.setBeanClass(beanClass);
    builder.beanDefinition.setFactoryMethodName(factoryMethodName);
    return builder;
}
```

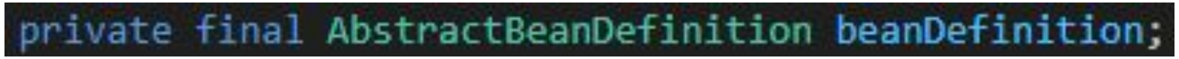


```java
private BeanDefinitionBuilder(AbstractBeanDefinition beanDefinition) {
    this.beanDefinition = beanDefinition;
}
```

Private constructor is called by internal static factory methods.

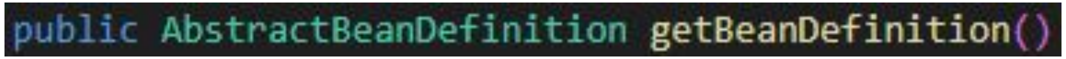
Design Pattern: Builder & Factory Method (Static)

All factory methods are static, and return a *BeanDefinitionBuilder*. The attribute



```java
private final AbstractBeanDefinition beanDefinition;
```

of the returned builder will be different child type of *AbstractBeanDefinition* based on the method called, and can be obtained using public method



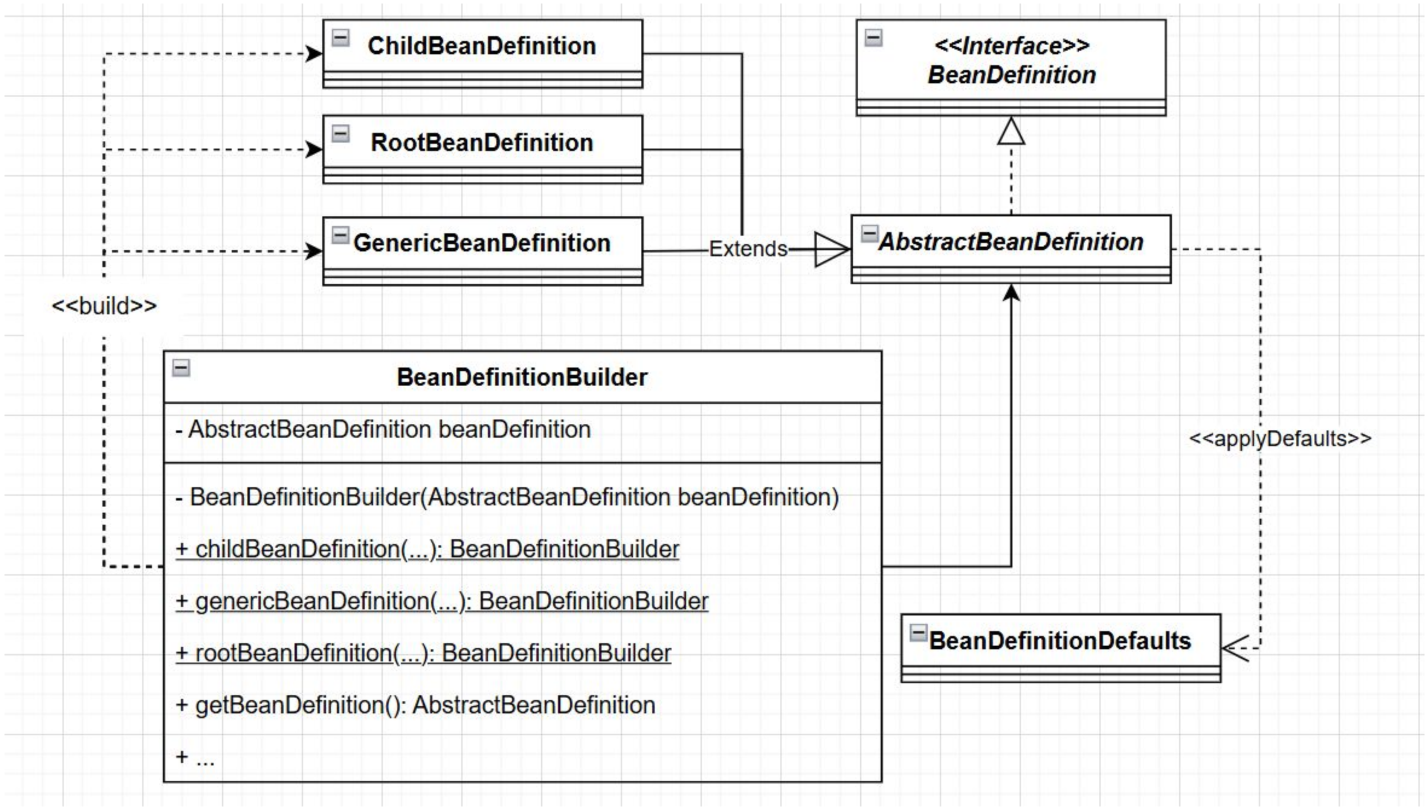
```java
public AbstractBeanDefinition getBeanDefinition()
```

Design Pattern: Builder & Factory Method (Static)

All methods are static, and return a *BeanDefinitionBuilder*. The attribute `private final AbstractBeanDefinition beanDefinition;` will be different child type of *AbstractBeanDefinition* based on the method called, and can be obtain using public method `public AbstractBeanDefinition getBeanDefinition()`

```
public static BeanDefinitionBuilder genericBeanDefinition();
public static BeanDefinitionBuilder genericBeanDefinition(String beanClassName);
public static BeanDefinitionBuilder genericBeanDefinition(Class<?> beanClass);
public static <T> BeanDefinitionBuilder genericBeanDefinition(Class<T> beanClass, Supplier<T> instanceSupplier);

public static BeanDefinitionBuilder rootBeanDefinition(String beanClassName);
public static BeanDefinitionBuilder rootBeanDefinition(String beanClassName, @Nullable String factoryMethodName);
public static BeanDefinitionBuilder rootBeanDefinition(Class<?> beanClass);
public static BeanDefinitionBuilder rootBeanDefinition(Class<?> beanClass, @Nullable String factoryMethodName);
public static <T> BeanDefinitionBuilder rootBeanDefinition(ResolvableType beanType, Supplier<T> instanceSupplier);
public static <T> BeanDefinitionBuilder rootBeanDefinition(Class<T> beanClass, Supplier<T> instanceSupplier);

public static BeanDefinitionBuilder childBeanDefinition(String parentName);
```

Design Pattern: Builder & Factory Method (Static)

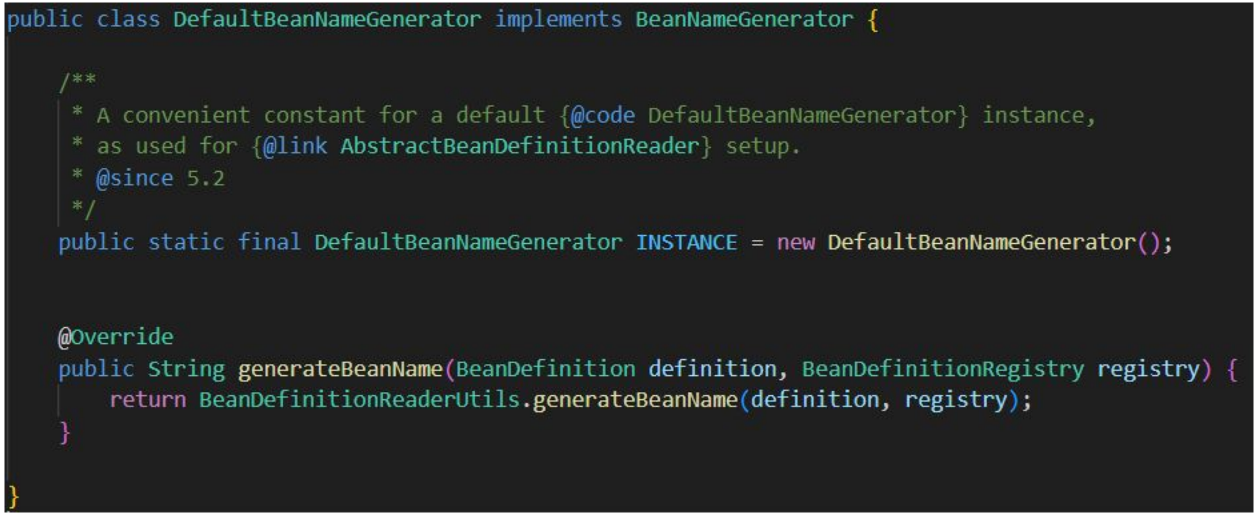

ChildBeanDefinition
RootBeanDefinition
GenericBeanDefinition
<<Interface>> BeanDefinition
AbstractBeanDefinition
Extends
<<build>>
BeanDefinitionBuilder
- AbstractBeanDefinition beanDefinition
- BeanDefinitionBuilder(AbstractBeanDefinition beanDefinition)
+ childBeanDefinition(...): BeanDefinitionBuilder
+ genericBeanDefinition(...): BeanDefinitionBuilder
+ rootBeanDefinition(...): BeanDefinitionBuilder
+ getBeanDefinition(): AbstractBeanDefinition
+ ...
<<applyDefaults>>
BeanDefinitionDefaults

DefaultBeanNameGenerator

This class provides unique bean name generation.

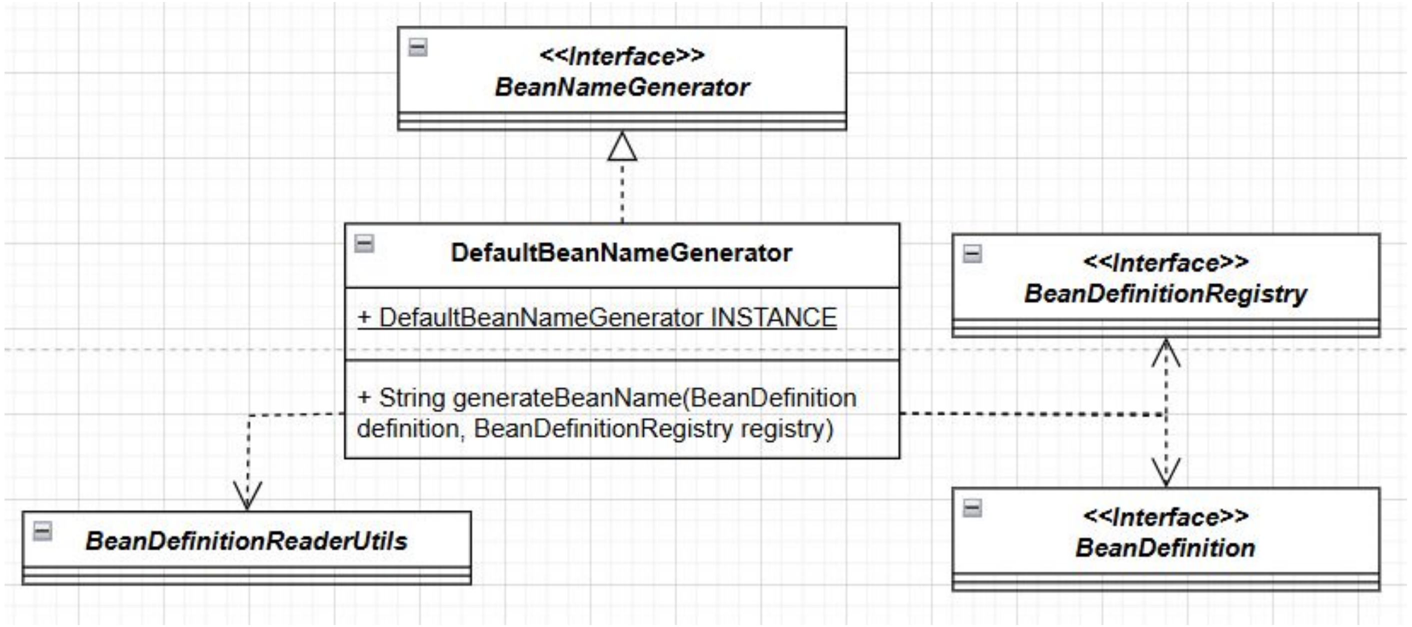
Design: Eager Instantiation & Global Accessor

- Eagerly instantiate a instance of itself, then make it immutable.
- **However, there are no private constructor to prevent instantitation, thus it’s not a Singleton pattern.**



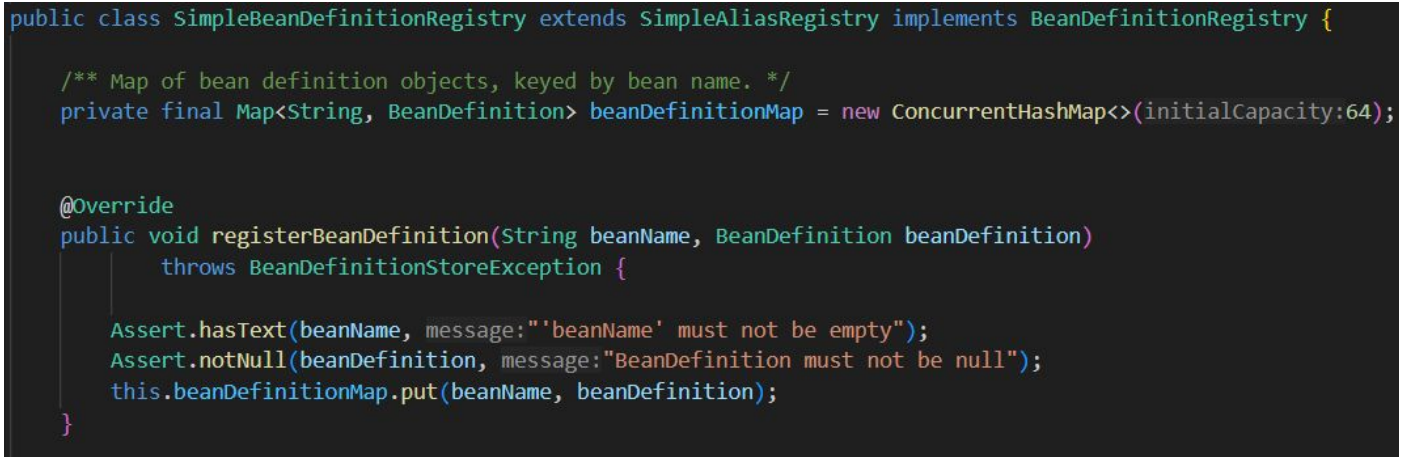
```
public class DefaultBeanNameGenerator implements BeanNameGenerator {

    /**
     * A convenient constant for a default {@code DefaultBeanNameGenerator} instance,
     * as used for {@link AbstractBeanDefinitionReader} setup.
     * @since 5.2
     */
    public static final DefaultBeanNameGenerator INSTANCE = new DefaultBeanNameGenerator();


    @Override
    public String generateBeanName(BeanDefinition definition, BeanDefinitionRegistry registry) {
        return BeanDefinitionReaderUtils.generateBeanName(definition, registry);
    }

}
```

Design: Eager Instantiation & Global Accessor

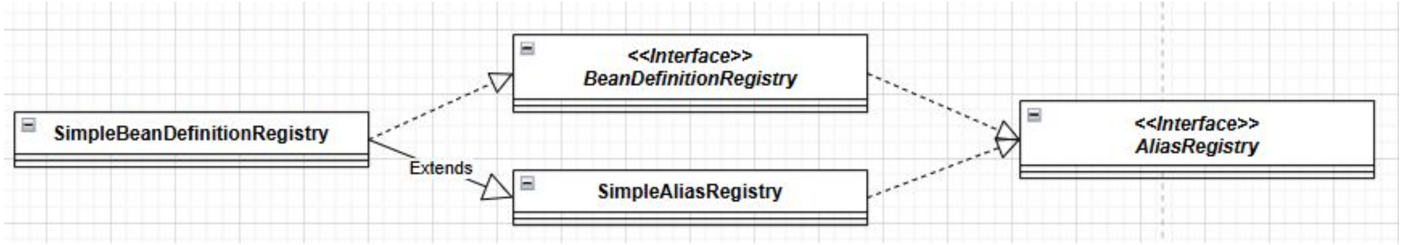

<<Interface>>
BeanNameGenerator
DefaultBeanNameGenerator
+ DefaultBeanNameGenerator INSTANCE
+ String generateBeanName(BeanDefinition definition, BeanDefinitionRegistry registry)
<<Interface>>
BeanDefinitionRegistry
BeanDefinitionReaderUtils
<<Interface>>
BeanDefinition

SimpleBeanDefinitionRegistry

This class simply holds a *Map<String, BeanDefinition>* to register bean definition.



```java
public class SimpleBeanDefinitionRegistry extends SimpleAliasRegistry implements BeanDefinitionRegistry {

    /** Map of bean definition objects, keyed by bean name. */
    private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<>(initialCapacity:64);


    @Override
    public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
            throws BeanDefinitionStoreException {

        Assert.hasText(beanName, message:"'beanName' must not be empty");
        Assert.notNull(beanDefinition, message:"BeanDefinition must not be null");
        this.beanDefinitionMap.put(beanName, beanDefinition);
    }
```



BeanDefinitionReaderUtils

Utility class that contains various methods useful for bean definition reader implementations.

AutowireUtils

Utility class that contains various methods useful for autowire-capable bean factories implementations.

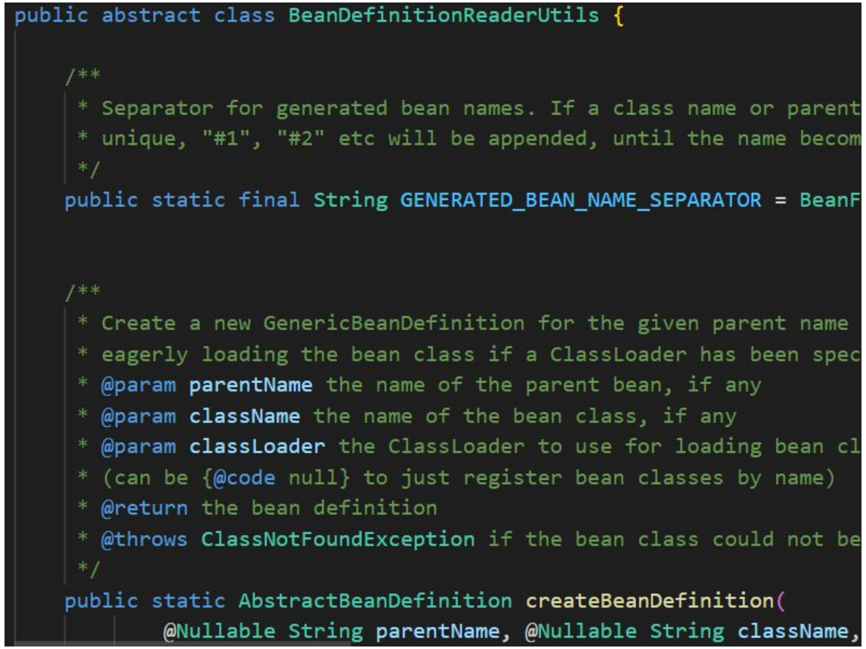
Design: Utility Class

BeanDefinitionReaderUtils & AutowireUtils

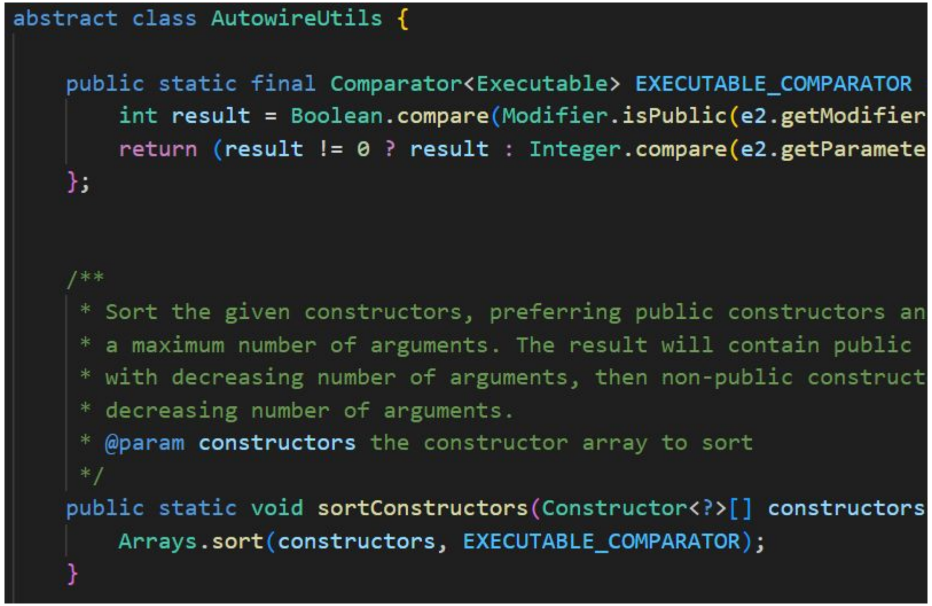
- These classes are **abstract** class, which means they **cannot be instantiated.**
- All of the methods are **static** and can be use directly.

Design: Utility Class

Abstract class with **static** methods



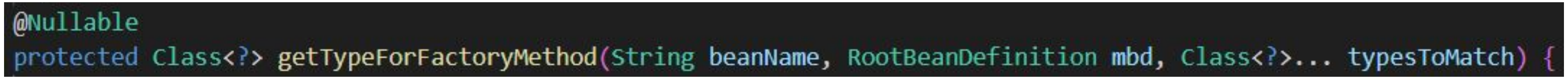
```
public abstract class BeanDefinitionReaderUtils {

    /**
     * Separator for generated bean names. If a class name or parent
     * unique, "#1", "#2" etc will be appended, until the name becom
     */
    public static final String GENERATED_BEAN_NAME_SEPARATOR = BeanF


    /**
     * Create a new GenericBeanDefinition for the given parent name
     * eagerly loading the bean class if a ClassLoader has been spec
     * @param parentName the name of the parent bean, if any
     * @param className the name of the bean class, if any
     * @param classLoader the ClassLoader to use for loading bean cl
     * (can be {@code null} to just register bean classes by name)
     * @return the bean definition
     * @throws ClassNotFoundException if the bean class could not be
     */
    public static AbstractBeanDefinition createBeanDefinition(
            @Nullable String parentName, @Nullable String className,
```



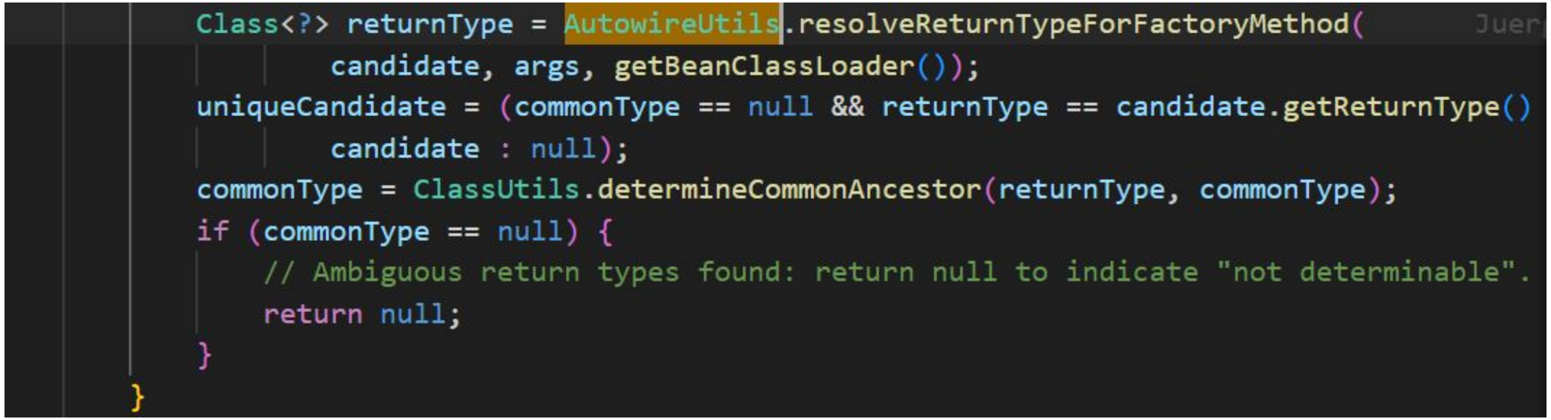
```
abstract class AutowireUtils {

    public static final Comparator<Executable> EXECUTABLE_COMPARATOR
        int result = Boolean.compare(Modifier.isPublic(e2.getModifier
        return (result != 0 ? result : Integer.compare(e2.getParamete
    };


    /**
     * Sort the given constructors, preferring public constructors an
     * a maximum number of arguments. The result will contain public
     * with decreasing number of arguments, then non-public construct
     * decreasing number of arguments.
     * @param constructors the constructor array to sort
     */
    public static void sortConstructors(Constructor<?>[] constructors
        Arrays.sort(constructors, EXECUTABLE_COMPARATOR);
    }
```

Design: Utility Class

In *AbstractAutowireCapableBeanFactory#getTypeForFactoryMethod()*:



```
@Nullable
protected Class<?> getTypeForFactoryMethod(String beanName, RootBeanDefinition mbd, Class<?>... typesToMatch) {
```



```
        Class<?> returnType = AutowireUtils.resolveReturnTypeForFactoryMethod(        Juer
                candidate, args, getBeanClassLoader());
        uniqueCandidate = (commonType == null && returnType == candidate.getReturnType()
                candidate : null);
        commonType = ClassUtils.determineCommonAncestor(returnType, commonType);
        if (commonType == null) {
            // Ambiguous return types found: return null to indicate "not determinable".
            return null;
        }
    }
```

Design: Utility Class

public static AbstractBeanDefinition createBeanDefinition(@Nullable String parentName, @Nullable String className, @Nullable ClassLoader classLoader);
public static String generateBeanName(BeanDefinition beanDefinition, BeanDefinitionRegistry registry);
public static String generateBeanName(BeanDefinition definition, BeanDefinitionRegistry registry, boolean isInnerBean);
public static String uniqueBeanName(String beanName, BeanDefinitionRegistry registry);
public static void registerBeanDefinition(BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry);
public static String registerWithGeneratedName(AbstractBeanDefinition definition, BeanDefinitionRegistry registry);

Design: Utility Class

```
public static void sortConstructors(Constructor<?>[] constructors);
public static void sortFactoryMethods(Method[] factoryMethods);
public static boolean isExcludedFromDependencyCheck(PropertyDescriptor pd);
public static boolean isSetterDefinedInInterface(PropertyDescriptor pd,
Set<Class<?>> interfaces);
public static Object resolveAutowiringValue(Object autowiringValue, Class<?>
requiredType);
public static Class<?> resolveReturnTypeForFactoryMethod(Method method,
Object[] args, @Nullable ClassLoader classLoader);
```

Conclusion

- We’ve seen many classes responsible for various Bean operations, including configuration reading, definition building and registration, bean instantiation, autowiring, and destruction.
- These classes use various design patterns, such as Chain of Responsibility, Adapter, Builder, Factory Method, Template, and Strategy, to solve common design problems, making the code more flexible, easier to maintain, and scalable as needed.

Thank you!