Component Models in Java

Slide 1: The Whiteboard Component model is a model, that facilitates communication through the usage of a Component Registry. Therefore, Components are not directly aware of each other, but they use the Component Registry to compose. The Component Registry works as a facility that pairs components with one another, depending on what interfaces they provide, and which interface they need. Furthermore, the Component Registry retains a list of provided interfaces and the corresponding provider objects.

In the example, Component A provides an Interface X and registers the provided interface within the Component Registry. Component B can then query the Component Registry, as it needs Interface X and finds Component A, which has provided Interface X. The Component Registry then retrieves the object registered by Component A and returns it to Component B.

Slide 2: An example of when we used the Whiteboard Model was during a NetbeansLab. The components are registered through POM files, which contain statements that declare, which services we want to register. The ServiceLoader can then locate all the classes, which implement the given interface. This is done through XML files, which a component can have one or multiple of. We then use the LookUp-API to access the modules, as they are registered to an App Module, and have an Install Module. Lastly, we used the ServiceProvider Annotation, which eased the creation of XML files, as they are auto generated.

Slide 3: Dependency Injection helps to bind independent components together without using the *new* operator. The components that are dependent on other components define their dependencies through annotations on constructor arguments or properties. The container can then inject the dependencies while creating and instantiating these components. As such, the requiring component can just specify the list of dependencies needed, and the framework provides the requiring component with instances of other components that fulfill the needed dependencies. This is also called Inversion of Control, which is the concept that the Spring Framework is built upon.

Slide 4: The Spring Container is the component runtime of the Spring Component model. The container handles the instantiating of components, configuration of their properties and wiring them together as needed. Thus, the Spring Container completely manages the life cycle of components from their creation through to their destruction. As the Spring Container is built on the principle of IoC, it uses Dependency Injection to manage the relationships across multiple components deployed in the container. The information needed for these relationships is specified within the configuration metadata, which is typically a Bean.xml file.

Spring Beans are the business components of the Spring Framework. They are developed and deployed in the Spring Container. Spring Beans are POJO's *(Plain Old Java Objects)* and are categorized as two types, at the time of instantiation.

Spring Beans can be either Singleton, or Prototype. If a Spring Bean is a Singleton, then there is only one instance of the Bean created per each Spring Container. This singular instance is cached and used for subsequent requests. If a Spring Bean is a Prototype, then there is any number of instances created based on the number of requests. By default, a Spring Bean is singleton in nature, but their type can be specified in the Bean configuration file.

Spring provides three ways of configuration. (1) XML-based configuration, (2) Annotation-based configuration and (3) Java-based configuration. In our AsteroidsSpring lab we utilized the first method of configuration, which was the XML-based configuration. In this lab, we utilized the ApplicationContextAPI to configure the various Beans. We had an overarching Beans.xml file, whose job was to collect all the other resources from the other modules and their respective Bean.xml files. Once collected, the Beans were used within the Game.java file and cast through a .getBean method. The various classes were then provided their respective services.

Some important metadata that Beans can be configured with are, id, class, name *(alias)*, scope *(singleton or prototype)* and more.

Slide 5: In our SpringLab, we utilized the ApplicationContextAPI to configure the Beans used. This was done through the "ClassPathXMLApplicationContext" method, and by providing the instantiation with the "Beans.xml" file. The file provided is an overarching file, whose job is to collect all the other resources from the other modules and their respective Bean.xml files.

Once collected. The Beans are used within the “Game.java” file and cast through a “.getBean”-method. The various classes are then provided their respective services by adding them to an ArrayList, which we iterate through.

The various Beans.xml files are configured through declaring namespaces and providing an ‘id’ and the implement class for the ‘id’.

Lastly, the XML file is used to uniquely identify the Beans and creating the objects needed. The components get registered with the Spring container through the XML configuration file. The XML file is read first when the Spring application is loaded, and the container uses this configuration file to create all the required beans.

---

1. **What is OSGI?**

OSGI is a dynamic module system for Java which enables modularity, as OSGI has a set of standards for dynamic modularity systems in Java. There are three conceptual layers in the OSGI Framework, (1) Module Layer, which is responsible for packaging and sharing code, (2) Life Cycle Layer, which is responsible for managing the life cycle of a deployed module during runtime and (3) Service Layer, which is responsible for dynamic service publication, searching and binding.

2. **What is a bundle in OSGI?**

Bundles in OSGI is a deplorable module in the form of a JAR file. These bundles contain the usual class files and resource files. However, they also contain MANIFEST information that specifies metadata about the bundle. The MANIFEST information includes the name, version number, dependencies and more. Furthermore, each Bundle has a unique identifier, which is the Bundle name and version. The addition of MANIFEST information easily enhances modularity and maintainability, as a Bundle must explicitly declare any external dependencies, which gives us, and the Service Registry knowledge on, how each Bundle interacts with one another, and the various dependencies each Bundle needs or can provide.

3. **What is the importance of the OSGI Service Registry?**

Bundles are deployed in the OSGI Framework through the Service Registry. It works akin to the Whiteboard Model, as the bundles that provide a service can publish the service in the Registry.

A service is defined by a Java Interface, which represents a conceptual contract between the provider and the consumer. Once a consumer finds the appropriate provider, it can bind and use the service.

4. **What is COMPONENT.XML in OSGI?**

The Component.xml files map between an SPI and a Service Provider. It defines which class is the Service Provider and which SPI it implements. It includes which methods bind the referenced service to the bundle and which methods unbinds. If the Component Configuration is not satisfied, it will deactivate.

Component-Oriented Design and Architecture

Slide 1: Component Models are essential, as they encourage the usage of composition and interfaces, which helps prevent tight coupling. Tight Coupling is when classes become too dependent on other classes, which can happen through various scenarios. One such scenario is when a class assumes too many responsibilities, while another scenario is when one concern is spread over too many classes instead of one. Tight Coupling can mean that classes become harder to maintain, test and enhance. Furthermore, it reduces flexibility and reusability. Lastly, changes in one class can create ripple effects, as various adjustments must be made in other classes to accommodate the change in the first class.

The amount of coupling between a class and other classes can be measured by assessing the class's Dependency Depth **(DP)**. A class that has no dependency has a DP of 0. Any class that depends on a class with DP 0 has a DP of 1 and so forth. Higher DP implies a tighter coupling between classes.

Slide 2: One way to avoid these issues are through Common Modules, which offer Common Business Entities that other modules can utilize. *(Business Entities are classes, which can be passed around by other components to pass data values across components).* As such, we have loose coupling, and we encourage that each module that utilize the Common Module has a single responsibility.

Furthermore, we can also use Interfaces to avoid tight coupling. Interfaces are “contracts”, which ensures that the modules implementing them perform specific behaviors according to the interface. As such, we provide functionality through the interface.

Slide 3: The Whiteboard Component model is a model, that facilitates communication through the usage of a Component Registry. Therefore, Components are not directly aware of each other, but they use the Component Registry to compose. The Component Registry works as a facility that pairs components with one another, depending on what interfaces they provide, and which interface they need. Furthermore, the Component Registry retains a list of provided interfaces and the corresponding provider objects.

In the example, Component A provides an Interface X and registers the provided interface within the Component Registry. Component B can then query the Component Registry, as it needs Interface X and finds Component A, which has provided Interface X. The Component Registry then retrieves the object registered by Component A and returns it to Component B.

Slide 4: Dependency Injection helps to bind independent components together without using the *new* operator. The components that are dependent on other components define their dependencies through annotations on constructor arguments or properties. The container can then inject the dependencies while creating and instantiating these components. As such, the requiring component can just specify the list of dependencies needed, and the framework provides the requiring component with instances of other components that fulfill the needed dependencies.

Slide 5: In our SpringLab, we utilized the ApplicationContextAPI to configure the Beans used. This was done through the “ClassPathXMLApplicationContext”, and by providing the instantiation with the “Beans.xml” file. The file provided is an overarching file, whose job is to collect all the other resources from the other modules and their respective Bean.xml files.

Once collected. The Beans are used within the “Game.java” file and cast through a “.getBean”-method. The various classes are then provided their respective services by adding them to an ArrayList, which we iterate through.

The various Beans.xml files are configured through declaring namespaces and providing an ‘id’ and the implement class for the ‘id’. Lastly, the XML file is used to uniquely identify the Beans

and creating the objects needed. The components get registered with the Spring container through the XML configuration file. The XML file is read first when the Spring application is loaded, and the container uses this configuration file to create all the required beans.

---

1. **What Design Strategies are used to lower Coupling?**

There are various Design Strategies to lower Coupling, such as the introduction of Interfaces, or a Layered, Component-based Architecture, which improves maintainability and the ability to extract reusable parts for storage and future reuse.

2. **Why should the Business Entity Objects not be offered as Dynamic Component Services?**

Entity Classes that represent Domain Objects should be kept in a separate Common Library to be shared by all other components. Entity Classes are considered as Data Carrying Objects, and they should not be considered as components. Business Entity Classes can be instantiated by any other components. The instances of Business Entity Classes can be passed around by other components to pass Data Values across Components.

3. **Why is Coupling on a Stable Entity preferred over Coupling on an Unstable Entity?**

Concrete Classes should not be relied upon, as code can change rapidly, and that would create even more ripple effects and the developers would spend more time refactoring rather than creating actual code. Therefore, we should couple to Interfaces, as they are contracts, which ensures that changes have a reason and can accommodate appropriately. If changes were to be made, there should be made an extension of the interface.

4. **Should Componentization of Multiple Layer Applications preserve the Original Layering Structure?**

Yes, as we do not want unnecessary coupling between the layers or modules.

JDK Service Loader

Slide 1: There are four Basic Component Structural Constructs, (1) Java Programming Language, (2) Java Software Development Kit, (3) Java SE Application Programming Interface, (4) Java Runtime Environment.

Slide 2: Furthermore, it instantiates the providing component object and passes this object to the requiring component. The Glue Code is built as a component, and this component refers to the other two components and the interface. Some downsides are that custom glue code needs to be written for each group of components that depend on one another, as it is not universal and needs to be specific for these groups of components. This also means that glue code is not scalable and can be hard to manage.

Slide 3: **The Whiteboard Component** model is a model, that facilitates communication through the usage of a Component Registry. Therefore, Components are not directly aware of each other, but they use the Component Registry to compose. The Component Registry works as a facility that pairs components with one another, depending on what interfaces they provide, and which interface they need. Furthermore, the Component Registry retains a list of provided interfaces and the corresponding provider objects. **Dependency Injection** helps to bind independent components together without using the *new* operator. The components that are dependent on other components define their dependencies through annotations on constructor arguments or properties. The container can then inject the dependencies while creating and instantiating these components. As such, the requiring component can just specify the list of dependencies needed, and the framework provides the requiring component with instances of other components that fulfill the needed dependencies.

Slide 4: In the example, Component A provides an Interface X and registers the provided interface within the Component Registry. Component B can then query the Component Registry, as it needs Interface X and finds Component A, which has provided Interface X. The Component Registry then retrieves the object registered by Component A and returns it to Component B.

Slide 5: For registering a Service Provider, we place a Provider Configuration File in the META-INF.services folder. This way, the ServiceLoader can locate them. The file name is the service provided interface, while the content is the service provider.

Slide 6: Each service then gets a Loader, where we load the available Service Providers for the service, in this example, IEntityProcessingService. We iterate through the various Service Providers and pick the first one. Afterwards, we use the SPILocator to locate the modules, which implement the specific service.

---

1. **What are the Pros and Cons of Basic Component Constructs provided by Java SE Platform?**

Developers can package custom code into a JAR file for distribution and use with other applications. The same physical component can carry multiple logical components. However, a con is that creating components and relying solely on components can create tight coupling, the more the system expands.

2. **Why is a Component Interface important? What is the primary reason for introducing a Component Interface?**

Component Interfaces are important as they reduce tight coupling between components during build time. Furthermore, it increases substitutability, and we can hide implementation details.

3. **What is a Component Model?**

A Component Model specifies the standards and conventions imposed on developers. A component framework is an implementation of services that support or enforce a component model.

4. **What is a relationship between a Component Model and a Component Framework?**

The Component Model is supported by a Component Framework, which usually is a runtime program that wires the components deployed.

OSGI

Slide 1: In the example, we have three JARs with three classes, respectively. With the FLAT CLASSPATH problem, JAR A is loaded first with all its three classes, then Class_5 and Class_7 from JAR B and JAR C, respectively. However, now the problem arises, as some Classes are not loaded, such as Class_2 in JAR C, as it has already been loaded from JAR A. The same occurs to JAR B and its classes, as it has already been loaded from JAR C. Classes can have different implementations in their respective JARs, but as we load classes from JARs in a sequence, that can create complications, as two classes must work with another class from a different JAR, which can change the behavior of components during runtime.

Slide 2: The Whiteboard Component model is a model, that facilitates communication through the usage of a Component Registry. Therefore, Components are not directly aware of each other, but they use the Component Registry to compose. The Component Registry works as a facility that pairs components with one another, depending on what interfaces they provide, and which interface they need. Furthermore, the Component Registry retains a list of provided interfaces and the corresponding provider objects.

In the example, Component A provides an Interface X and registers the provided interface within the Component Registry. Component B can then query the Component Registry, as it needs Interface X and finds Component A, which has provided Interface X. The Component Registry then retrieves the object registered by Component A and returns it to Component B.

Slide 3: Dependency Injection helps to bind independent components together without using the *new* operator. The components that are dependent on other components define their dependencies through annotations on constructor arguments or properties. The container can then inject the dependencies while creating and instantiating these components. As such, the requiring component can just specify the list of dependencies needed, and the framework provides the requiring component with instances of other components that fulfill the needed dependencies.

Slide 4: OSGI introduces another layer of abstraction, which is the module layer surrounding the JAR file. A bundle in OSGI is like a regular JAR file that contains class files, resource files, and metadata. However, there is also the additional MANIFEST data, which is metadata relevant

to OSGI. The Framework uses this information to provide stronger boundaries around the bundle, which means a bundle knows exactly what defines it and can find it faster at runtime.

Slide 5: A service consists of a service interface and a service object. The service interface is akin to a provided interface of a component, and the service object is akin to an internal component implementation of the provided interface. The service interface defines a set of API operators. A service interface represents a conceptual contract between the provider and consumer. A service object provides implementation of the service.

Once the service interface is defined, the service can be registered from the Activator of the bundle using the BundleContext API. Once a bundle registers a service, the service functionality is made available to other bundles under the runtime framework's control. It can also be registered in the OSGI.BND or MANIFEST.MF file with the use of an export statement.

Declarative Services work on top of the service layer for the dynamic wiring of components. These services require an XML file named component.xml, which defines the component and its provided and required interfaces. Furthermore, Declarative Service has additional methods, such as ACTIVATE, which is a method invoked when the component is activated. And gotService and lostService, which are methods that are invoked when a service object is binded or unbinded, respectively.

---

1. **Are all OSGI bundles JAR files?**

Yes, however, they are also more powerful than regular JAR files as they have enforcing module boundaries, as a bundle must explicitly define what portion of its internal code is externally visible to other bundles. And which bundles they rely on.

2. **How is the unique identity of an OSGI bundle defined?**

The unique identity is defined by a bundle name and version number, which is defined in the OSGI.BND file as extra manifest headers.

3. **Can an OSGI bundle transition its state from installed to active?**

Yes, but it requires a few states before that, such as resolved and starting.

4. **How does the OSGI Framework carry out bundle resolution? Is the same resolution mechanism applicable to the OSGI Service Layer?**

The OSGI Framework matches the exports and imports of deployed bundles to dynamically wire the entire application. This ensures consistency among the different bundles in terms of versions and other constraints.

5. **Do declarative services provide auto-wiring of required and provided interfaces?**

Once a reference is defined in the component description, the Service Component Runtime reads the information at runtime and locates the provided service that matches the reference and binds it.

6. **In what way does Declarative services specification enhance upon the Service Layer specification?**

It automates the process of providing components, as it requires an XML file instead, in which we specify what we require or provide. Furthermore, it reduces our job as the registration and LookUp of Services has been automated.

Spring

Slide 1: The Spring Container is the component runtime of the Spring Component model. The container handles the instantiating of components, configuration of their properties and wiring them together as needed. Thus, the Spring Container completely manages the life cycle of components from their creation through to their destruction. As the Spring Container is built on the principle of IoC, it uses Dependency Injection to manage the relationships across multiple components deployed in the container. The information needed for these relationships is specified within the configuration metadata, which is typically a Bean.xml file.

Slide 2: Spring Beans are the business components of the Spring Framework. They are developed and deployed in the Spring Container. Spring Beans are POJO's *(Plain Old Java Objects)* and are categorized as two types, at the time of instantiation.

Spring Beans can be either Singleton, or Prototype. If a Spring Bean is a Singleton, then there is only one instance of the Bean created per each Spring Container. This singular instance is cached

and used for subsequent requests. If a Spring Bean is a Prototype, then there is any number of instances created based on the number of requests. By default, a Spring Bean is singleton in nature, but their type can be specified in the Bean configuration file.

Beans are wired by using the annotation @Autowired it plays a massive role in binding references between two bean components. The Spring container uses dependency injection between collaborating beans through this annotation. The @Autowired annotation can be used to wire beans in the constructor, settings methods, and properties of the bean component.

Slide 3: Spring provides three ways of configuration. (1) XML-based configuration, (2) Annotation-based configuration and (3) Java-based configuration. For the **Annotation-based configuration**, it replaces the usage XML for describing Bean wiring but instead moves the Bean configuration directly into the component class. The same goes with the **Java-based annotation**, but it uses the @Configuration and @Bean annotation instead. The first indicates that the class can be used by the Spring Container and the @Bean annotation means that the method will return an object that should be registered as a Bean in the Spring Application Context.

Slide 4: In our SpringLab, we utilized the ApplicationContextAPI to configure the Beans used. This was done through the "ClassPathXMLApplicationContext", and by providing the instantiation with the "Beans.xml" file. The file provided is an overarching file, whose job is to collect all the other resources from the other modules and their respective Bean.xml files.

Once collected. The Beans are used within the "Game.java" file and cast through a ".getBean"-method. The various classes are then provided their respective services by adding them to an ArrayList, which we iterate through.

The various Beans.xml files are configured through declaring namespaces and providing an 'id' and the implement class for the 'id'. Lastly, the XML file is used to uniquely identify the Beans and creating the objects needed. The components get registered with the Spring container through the XML configuration file. The XML file is read first when the Spring application is loaded, and the container uses this configuration file to create all the required beans.

---

1. **How are the business components in Spring represented and identified?**

The @Service annotation is an extension of the @Component annotation in Spring, which is used for identifying service components (business components are popularly known as service components in Spring). This annotation allows the component runtime to detect the business components through CLASSPATH scanning. Alternatively, the details of the Spring beans can be entered in the Web configuration file.

NetBeans

Slide 1: In the example, we have three JARs with three classes, respectively. With the FLAT CLASSPATH problem, JAR A is loaded first with all its three classes, then Class_5 and Class_7 from JAR B and JAR C, respectively. However, now the problem arises, as some Classes are not loaded, such as Class_2 in JAR C, as it has already been loaded from JAR A. The same occurs to JAR B and its classes, as it has already been loaded from JAR C. Classes can have different implementations in their respective JARs, but as we load classes from JARs in a sequence, that can create complications, as two classes must work with another class from a different JAR, which can change the behavior of components during runtime.

Slide 2: Through the Module Classloader, each module registered in the Module System has an instance of the module classloader created, by means of which every module obtains its own namespace. In the NetBeans Classloader System we have a System Classloader which is implicitly a parent of every module classloader. The multiparent module classloader enables classes to be loaded from other modules, while avoiding namespace conflicts. The loading of classes is delegated to the Parent Classloader, rather than the modules themselves. However, as it is the parent, it is possible to load everything provided by a module with the Parent Classloader, through e.g. LookUp. Lastly, the Original Class loader loads resources out of the Class Path of the launcher of the application

Slide 3: A module in NBM can consist of the Layer file, Class files, and Resource files, but it is mandatory that it has a MANIFEST.MF file, as it identifies the module. Furthermore, there is an XML file outside the JAR archive, which belongs to each module. Using these XML files, the NetBeans Platform knows the modules that are available to it, as well as their locations and the contracts that need to be satisfied for them to be allowed to be loaded. These dependencies are declared in the module's manifest file and resolved by the NetBeans runtime container, which

ensures that the application always starts up in a consistent state. Each module has a unique identity, which is identified by the OPEN-IDE-MODULE Name and Version Name.

Slide 5: NetBeans services were used in one of our labs, which takes inspiration from the Whiteboard Model, which has a Component Registry, in which modules can provide interfaces and requiring components can query the Registry to find needed interfaces. A service is registered through the META-INF folder. In which we then use the @ServiceProvider interface to utilize the service in the various modules. We can then locate and access the service through the LookUp API. We can then locate and access the service through the LookUp API, which is able to locate all registered services. An example is the Lookup.findAll(myInterface) to locate all the services that export the interface implementation.

---

1. **Are all NetBeans modules JAR Files?**

Yes. As they are an archive which consists of the manifest file, layer file, class files and resources.

2. **Are all JAR files NetBeans modules?**

No.

3. **How does NetBeans Runtime-Container carry out Module Resolution?**

By using the Installer.class where each method is defined. As we have different methods that influence each module's lifecycle. Such as, validate(), restored(), uninstalled(), closing(), and close().

Project

In our project we utilized OSGI and Dependency Injection. Our project is a combination of the old school mobile game, Snake and another Online Multiplayer game named Curve Fever. The objective for the game is to stay alive the longest while battling other snakes on the map. These snakes are handled by an AI.

Our project contains various Common Modules that are exported as packages for other modules to implement and use. This exportation happens explicitly in the OSGI.BND files. The modules that need these packages can then integrate them within their POM.XML files. We can also observe that Classes that require a provided interface from these packages explicitly declare which interface is needed in their XML files.

Furthermore, we also make usage of Service Components. An example of this is the Enemy module, in which we can observe that under the OSGI.BND file that there are two service components, which the Enemy Module needs.

We can find the Service Components within the Enemy's META-INF folder, where they both explicitly state the Class, which implements the Interface, and the Interface, which the Class needs. Furthermore, we can under the EntityProcessor.XML observe, that there is a reference to the AiSPI Interface, which binds to a setAiSPI method that can be located under EnemyControlSystem.java. During runtime, OSGI then knows that the setAiSPI method needs to be configured through the usage of Setter Injection.

We can observe under the EnemyControlSystem.java file that the aiSPI has been declared. And further down also observe the method, setAiSPI, that during runtime is Setter-Injected, which we then make use of under the Process method. If the method is properly handled, then the Enemy Snakes makes use of the implementation from the AiSPI Service, however, if it is not, then we use a Random Movement instead to move the various Enemy Snakes.

The Implementation for the AiSPI is given to us through the AI module. In which we can observe the OSGI.bnd and see that there are two Service Components, the essential one is AIProvider.xml. We can within this file observe that the implementation class is AiProvider, and observe that it can provide one interface, which is the AiSPI.