ECE 448/528
Application Software Design

Lecture 15. Object Relationship and Dependency Injection

Spring 2025

Won-Jae Yi, Ph.D.

Department of Electrical and Computer Engineering
Illinois Institute of Technology

Object Relationship

Object Relationship

- In object-oriented programming, functionality is described as objects and their interactions.
- **Object relationship**: objects need to know other objects before they can interact with each other.
- There are many types of relationships, depending on how closely the objects are related to each other.
    - Strongest: **composition** (A fundamental principle allowing to build complex objects by combining simpler ones)
    - Weakest: **association** (A relationship between 2 or more classes describing their relationship/how they interact with each other)
    - A major design choice (particularly in C++, 'not-so-much' in Java)
- Depends on your understanding of the system.
    - Let's focus on composition and association.
- Depends on language and library features.
    - For OOP, we almost always use member variables to represent relationship, while GC may make analysis and design simpler.
    - It is possible to completely decouple the objects by using the Pub/Sub or observer pattern.

Composition

```
public class HTTPCommands implements RequestHandler {
  private final TreeMap<String, PlugSim> plugs = new TreeMap<>();
...
public class MqttCommands {
  private final TreeMap<String, PlugSim> plugs = new TreeMap<>();
...
public class Main implements AutoCloseable {
  private final JHTTP http;
  private final MqttClient mqtt;
...
```

- Composition: ***ownership***
- A fundamental principle allowing to build complex objects by combining simpler ones.
  - Creating relationships between classes where one class contains an instance of another class as a member variable
  - A `HTTPCommands` object creates its member `plugs`.
- Members for composition are typically required.
  - Use `final` members to ensure creation in/before constructor.
  - Getters may be provided but there should be no setters.
- May be identified via part-whole and has-a relations.
  - one class (the whole) consists of or is composed of one or more instances of another class (the part).

Association

```
public class RequestProcessor implements Runnable {
  private final Socket connection;
  private final RequestHandler handler;

  public RequestProcessor(Socket connection, RequestHandler handler) {
    this.connection = connection;
    this.handler = handler;
...
```

- Association: ***knowledge of***
- A relationship between 2 or more classes describing their relationship/how they interact with each other
  - A `RequestProcessor` object interacts with `handler` directly by calling methods of `handler`.
- Members for associations could be optional.
  - Required ones are preferred to be `final` and are set in the constructor.
  - Optional ones may be set by setters.
- May be identified via use-a relations.
  - One class utilizes the functionality of another class without being composed of it or owning it.

An Example System

- An `App` composed of 9 objects/components.
  - Bean-A of type `ComponentA`
  - Bean-B1 and Bean-B2 of type `ComponentB`
  - Bean-C1 and Bean-C2 of type `ComponentC`
  - Bean-C3 of type `ComponentCX`
  - Bean-D1 and Bean-D2 of type `ComponentD`
  - List-C of type `List<InterfaceC>`
- Assume both `ComponentC` and `ComponentCX` implement the interface `InterfaceC`.
- Build a larger object by composition of smaller objects, and allow objects to interact with each other via association.

An Example System (Cont.)

```
public class App {
  public App() {
    this.beanA = new ComponentA("Bean-A");
    this.beanB1 = new ComponentB(this.beanA, "Bean-B1");
    this.beanB2 = new ComponentB(this.beanA, "Bean-B2");
    this.beanC1 = new ComponentC(this.beanB1, "Bean-C1");
    this.beanC2 = new ComponentC(this.beanB2, "Bean-C2");
    this.beanC3 = new ComponentCX("Bean-C3");
    this.beanD1 = new ComponentD(this.beanC1, "Bean-D1");
    this.beanD2 = new ComponentD(this.beanC2, "Bean-D2");
    this.beanC1.setComponentA(beanA);
    this.beanC1.setComponentD(beanD1);
    this.beanC2.setComponentA(beanA);
    this.beanC2.setComponentD(beanD2);
    this.listC = Arrays.asList(beanC1, beanC2, beanC3);
  }
...
```

- Complicated but realistic associations between objects.
  - `ComponentA` is a **singleton** (has a single instance within `App`), and is used by many objects.
  - Bi-directional (cyclic) associations between `ComponentC` and `ComponentD` – must use a setter.

Dependency Injection

The Design Problem

```
public class App {
  public App() {
    this.beanA = new ComponentA("Bean-A");
    this.beanB1 = new ComponentB(this.beanA, "Bean-B1");
    this.beanB2 = new ComponentB(this.beanA, "Bean-B2");
    this.beanC1 = new ComponentC(this.beanB1, "Bean-C1");
    this.beanC2 = new ComponentC(this.beanB2, "Bean-C2");
    this.beanC3 = new ComponentCX("Bean-C3");
    this.beanD1 = new ComponentD(this.beanC1, "Bean-D1");
    this.beanD2 = new ComponentD(this.beanC2, "Bean-D2");
    this.beanC1.setComponentA(beanA);
    this.beanC1.setComponentD(beanD1);
    this.beanC2.setComponentA(beanA);
    this.beanC2.setComponentD(beanD2);
    this.listC = Arrays.asList(beanC1, beanC2, beanC3);
  }
...
```

- What if there are more objects in the system?
- What if a team of developers is working on different class types, and no one has complete knowledge to write the entire constructor?

Dependency Injection

- Placing the responsibility of creating components and managing their associations on a system can be overwhelming.
- **Dependency Injection** shifts this responsibility by allowing each component to create itself and establish associations independently with the help of supporting libraries.
  - **Dependency**: object relationship (you want to use other's object)
  - **Injection**: object will be 'injected' into your class to use 'magically'
- Associations between components can often be inferred from the code structure.
  - If only one instance of `ComponentA` exists (singleton) and a constructor or setter requires a `ComponentA` object, then that `ComponentA` must be used.
  - Reflection, a powerful Java feature, enables objects to examine themselves and others, making dependency injection possible.

Spring Framework

- A popular Java framework to support dependency injection.
  - And many other features – probably the most comprehensive Java framework to support web application development.
- We'll focus on Spring Boot in this course.
  - A modern framework based on Spring to support a standalone web application that requires minimum configuration efforts.
- Please be advised that for a framework like Spring that is quite complicated and has a long history, many documentations and tutorials online could be either out-of-date or not relevant to this course, or modern web application development in general.
  - Use your own judgment!

Spring Boot Application

```
public class Main implements AutoCloseable {
  private final ConfigurableApplicationContext appCtx;
  public Main() throws Exception {
    SpringApplication app = new SpringApplication(App.class);
    logger.info("My Spring App is created.");
    ...
    this.appCtx = app.run();
    logger.info("My Spring App is running.");
    ...
  }
  ...
```

- In `src/main/ece448/lec15`, under branch `lec15-di`.
- A `SpringApplication` object represents a Spring Boot application, which depends on your `App` class.
- Use the `run` method to start the application.
  - For event-driven applications, the components will need to maintain their own threads for event loops like the `JHTTP` server or use a library like `MqttClient`.

Managing Application Lifetime

```
public class Main implements AutoCloseable {
  private final ConfigurableApplicationContext appCtx;
  public Main() throws Exception {
    ...
    this.appCtx = app.run();
    logger.info("My Spring App is running.");
    ...
  }
  @Override
  public void close() throws Exception {appCtx.close();}
  public static void main(String[] args) throws Exception {
    try (Main m= new Main()) {
      for (;;) Thread.sleep(60000);
    }
  }
  ...
```

- `run` returns a `ConfigurableApplicationContext` object that can be used to stop the application.
- Since components manage their own threads, when necessary, the main thread can be used for other purposes after the application starts.

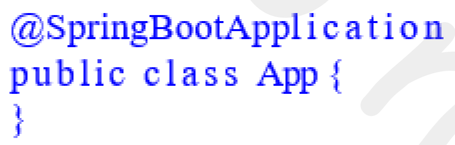
The Environment

```
public class Main implements AutoCloseable {
  private final ConfigurableApplicationContext appCtx;
  public Main() throws Exception {
    . . .
    HashMap<String, Object> props = new HashMap<>();
    props.put("optionA", "argA");
    props.put("optionB", "argB");
    app.setDefaultProperties(props);
    logger.info("My Spring Appenv: {}", props);

    this.appCtx = app.run();
    logger.info("My Spring App is running.");
    . . .
  }
  . . .
```

- Key/value pairs may be provided to initialize components in the application.
  - Instead of passing parameters to the constructor of the App class and letting it pass them to other components.

The App Class



```
@SpringBootApplication
public class App {
}
```

- The code appears that there is nothing in the App class, the `@SpringBootApplication` tells the Spring framework to
  - Configure the application using default settings.
  - Scan current and sub packages for components.
- You may also create components here if you choose to.

@Component and Singleton

```
@Component("beanA")
public class ComponentA {
  private final String name = "Bean-A";
  public ComponentA(Environment env) {
    logger.info("{}: created with {}.", name, env.getProperty("optionA"));
  }
  public String getName() {return name;}
  ...
}
```

- Use `@Component` to mark a class as a singleton.
  - There will be exactly one component of this class.
  - You may name the component if needed.
- The constructor needs the `Environment` object.
  - It will be provided by Spring (as you passed in `Main`, by `setDefaultProperties(props)`.
- Creating `@Component` may fail if
  - There is more than one constructor.
  - Spring does not know how to provide some parameters for the constructor.

Multiple Components of the Same Type

```
public class ComponentB {
  private final String name;
  private final ComponentA a;

  public ComponentB(Environment env, ComponentA a, String name) {
    logger.info("{}: created with {} and {}.",
      name, a.getName(), env.getProperty("optionB"));
    this.name = name; this.a = a;
  }

  public String getName() {return name;}
  ...
}
```

- We can't use `@Component` because we need two `ComponentB` objects, beanB1 and beanB2.
- For the constructor, while Spring may provide `env` and `a`, we need to provide `name` when creating `ComponentB`.

@Configuration and Factory

```
@Configuration
class FactoryB {
  @Bean
  public ComponentB beanB1(Environment env, ComponentA a) {
    return new ComponentB(env, a, "Bean-B1");
  }
  @Bean
  public ComponentB beanB2(Environment env, ComponentA a) {
    return new ComponentB(env, a, "Bean-B2");
  }
}
```

- Use `@Configuration` to mark a class as a factory of components (BeanFactory).
- **Beans** are objects of classes that are managed by Spring. Traditionally, objects are used to create their dependencies. But, Spring manages all dependencies of an object and instantiates the object after injecting the required dependencies.
- Each method with `@Bean` will be called once to generate one component with the same name.
- Spring needs to provide all the parameters.
- You have full control over how components are created.

Accessing Multiple Components of the Same Type

```
public interface InterfaceC {
  public String getName();
}

class ComponentC implements InterfaceC {
  public ComponentC(ComponentB b, String prefix) {
...

class ComponentCX implements InterfaceC {
  public ComponentCX(String name) {
...
```

- How can we use different `ComponentB` components to create different `ComponentC` components?

@Qualifier and Component Name

```
@Configuration
class FactoryC  {
   @Bean
   public InterfaceC beanC1(@Qualifier("beanB1") ComponentB b1) {
      return new ComponentC(b1, "Bean-C1");
   }
   @Bean
   public ComponentC beanC2(@Qualifier("beanB2") ComponentB b2) {
      return new ComponentC(b2, "Bean-C2");
   }
   @Bean
   public ComponentCX beanC3() {return new ComponentCX("Bean-C3");}
. . .
```

- Use `@Qualifier` to specify which component should be used if there could be more than one match.
  - Otherwise, Spring is not able to tell which `ComponentB` to use.
- Factory may create components of different types
- Components may be exposed via interfaces.

Type Matching

```
@Configuration
class FactoryC {
  @Bean
  public InterfaceC beanC1(...
  @Bean
  public ComponentC beanC2(...
  @Bean
  public ComponentCX beanC3(...
  @Bean
  public List<InterfaceC> listC(@Qualifier("beanC1") InterfaceC c1,
    @Qualifier("beanC2") InterfaceC c2, ComponentCX c3) {
    logger.info("List-C: [{}, {}, {}]",
      c1.getName(), c2.getName(), c3.getName());
    return Arrays.asList(c1, c2, c3);
  }
}
```

- Any parameter of type `InterfaceC` will match any of beanC1, beanC2, and beanC3.
  - Must use `@Qualifier` with `c1` and `c2`.
- `c3` will only match with beanC3.
  - No need to use `@Qualifier`.

@Autowired Members

```
class ComponentC implements InterfaceC {
  . . .
  @Autowired
  private ComponentA a;

  private ComponentD d;

  public ComponentC(ComponentB b, String prefix) {
    logger.info("{}/?/?: created with {}.", prefix, b.getName());
    this.prefix =prefix;
  }
. . .
```

- For members for association like a that are not set in the constructor, we may ask Spring to set it use `@Autowired`.
  - Spring automatically injects a bean of a compatible type
- Spring uses reflection to get around `private` so there is no need to write a setter – this may help to prevent someone mistakenly resetting a later.
- However, we cannot `@Autowired d` since its component name depends on the `prefix`.
- Be careful that both a and d are null in the constructor.

@Autowired Methods

```
class ComponentC implements InterfaceC {
  ...
  @Autowired
  public void onContext(ApplicationContext ctx) {
    d = (ComponentD)ctx.getBean("beanD"+prefix.substring(6));
  }
...
```

- Methods with `@Autowired` will be called by Spring to further setup the component.
- Use the `ApplicationContext` object to search for components by their names.
  - It will be provided by Spring.

Overview

- Component creation
  - Use `@Component` for singleton classes.
  - Use `@Configuration` and `@Bean` to fully control the creation.
- Additional component setup
  - Apply `@Autowired` to member variables.
  - Apply `@Autowired` to methods.
- Spring will match parameters/variables with components using their types.
  - Use `@Qualifier` to resolve multiple possible matches.
- Play with the code!
  - What additional functionality has been provided by Spring Boot to our application?