SYSC 4806 Software Engineering Laboratory

Dependency Injection and the Spring Framework

Java Components, a.k.a JavaBeans

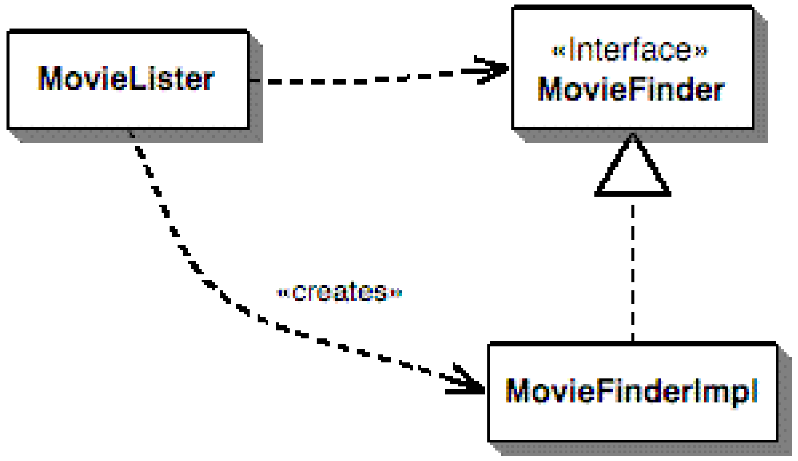
- JavaBeans are basically Plain Old Java Objects (POJOs)
  - they just follow a few conventions, so they can be easily initialized and assembled by 3rd party tools:
    - provide a default constructor
    - getter/setter method naming
- As standalone components they could be assembled in a “BeanBox”, although the original BeanBox implementation didn’t last long…
  - GUI design tools perpetuate the idea (most Swing classes are JavaBeans)
- Enterprise JavaBeans (EJBs) extended the idea for web app development (with support for threading, transactions, data access, etc.), but got too complex:
  - introduced many constraints and *dependencies*
  - backlash against EJBs led to a return to the POJO model

Dependency Injection

- Another term and pattern coined by Martin Fowler
  - see the blog post where it came from
  - also described in PEAA as the Plugin pattern
- It's about decoupling pushed to the extreme, to help with "componentization" and ease of unit testing
  - components and services are supposed to be even more reusable than typical classes
  - this means the component shouldn't be dependent on the implementations of other classes unless they are part of the service that it is supposed to provide
  - the component can be tested in isolation, using stubs when appropriate

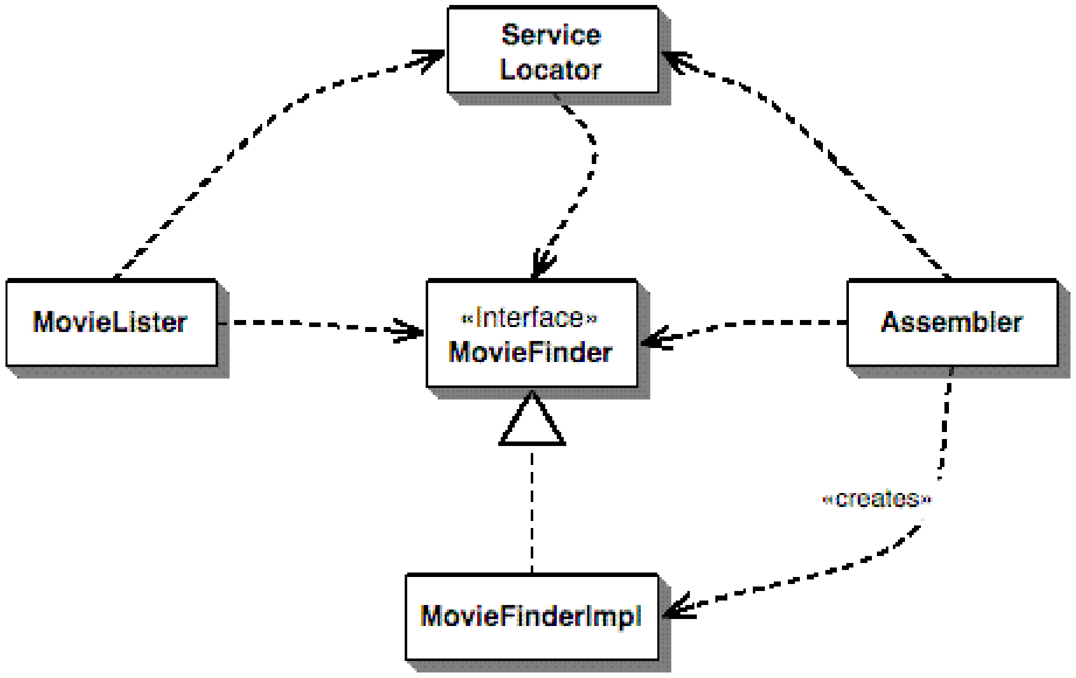
The dependency problem by example

(from Fowler’s blog post)



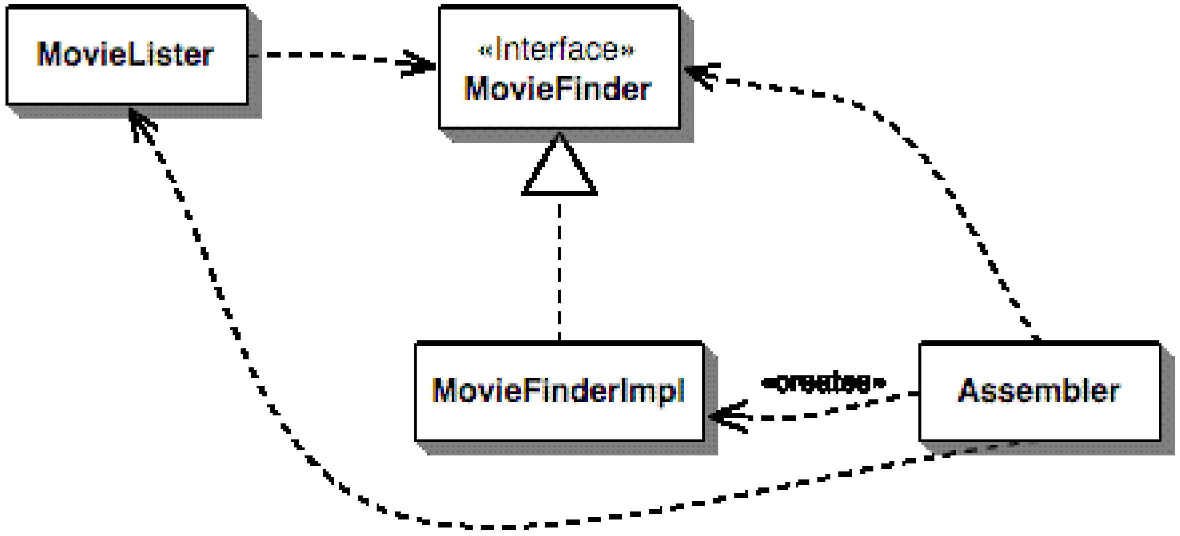
Dependency Lookup using a Service Locator

Used by EJBs via JNDI
(fig. from Fowler's blog post)



Dependency Injection with Inversion of Control

(from Fowler's blog post)



Benefits of Dependency Injection

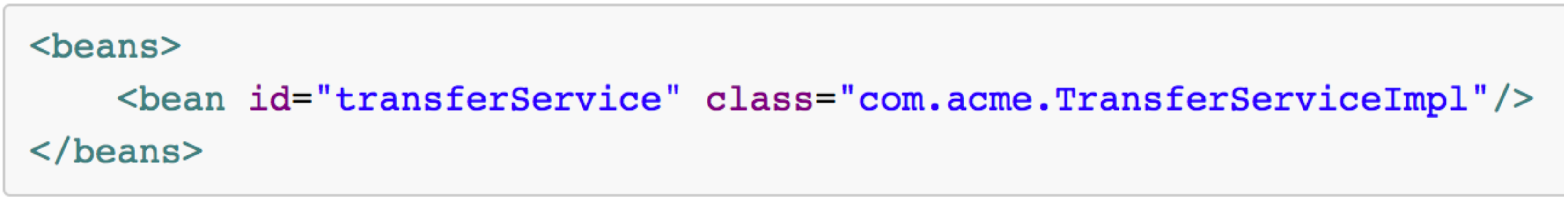
- Ease of unit testing: the dependency can be mocked by the container, no change to the unit required
- No boilerplate code to hook up components together
- No dependency on a framework on the container. It calls *you*.

Bean Configuration in Spring

- three possibilities: XML, Java code, Java annotations,
- XML is the old way, Java annotations are probably the best way now
- the beans we want to configure are the ones whose dependencies we don’t want to hard code: data mapper, various services, etc.

XML-based Configuration

- <beans>, <bean> tags



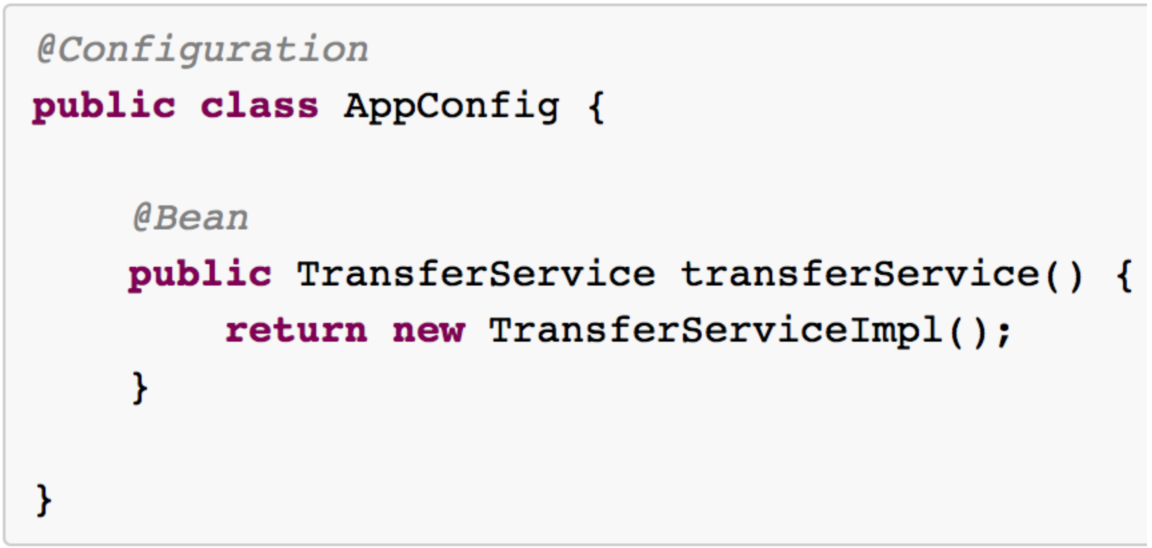
```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

- To load:

```
ApplicationContext context = new
        ClassPathXMLApplicationContext("/path/to/config.xml");
```

Java-based Configuration

- @Configuration, @Bean annotations
  - corresponding respectively to <beans> and <bean> tags



```java
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

}
```

- To load:

```java
ApplicationContext context = new
        JavaConfigApplicationContext(AppConfig.class);
```

Annotation-based Configuration

- My preferred way! Works especially well for *Spring Boot*
- @Component, @Service, @Repository to declare beans

```
@Service
public class TransferServiceImpl implements TransferService
```

- @Autowired to declare injection point
- @ComponentScan to do away with config files

Spring Boot

- Takes an “opinionated view” to help do away with most configuration. Only need to:
  - use annotation-based configuration for your beans
  - declare Spring Boot starter kit dependencies in Maven or Gradle
  - “properly” annotate the main class so that it auto-configures, scans for beans, and launch SpringApplication in the main() method
- Best understood by checking out the docs

Spring MVC annotations

- @Controller, @RestController, @Service
- @Repository, @CRUDRepository: for DAO and Models
- @Resource