Dependency Injection

Ingegneria Del Software

Università degli Studi di Padova
Dipartimento di Matematica
Corso di Laurea in Informatica

rcardin@math.unipd.it

Dependency Injection

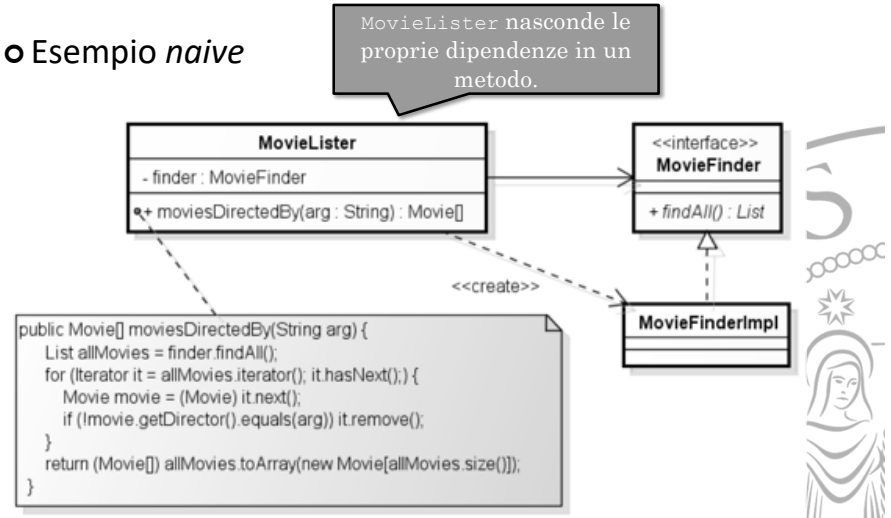
Relazioni tra		Campo di applicazione		
		Creational (5)	Structural (7)	Behavioral (11)
	Class	Factory method	Adapter (Class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter(Object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor
Architetturali		Dependency injection, Model view controller		

Introduzione e Contesto

- Scopo
  - Separazione del comportamento di una componente dalla risoluzione delle sue dipendenze
- Motivazione
  - Collegare due componenti in modo esplicito ne aumenta l'accoppiamento
    - Progettazione *unit-test* difficoltosa
    - Riutilizzo scarso della componente
    - Scarsa manutenibilità
  - Le dipendenze vanno minimizzate!!!

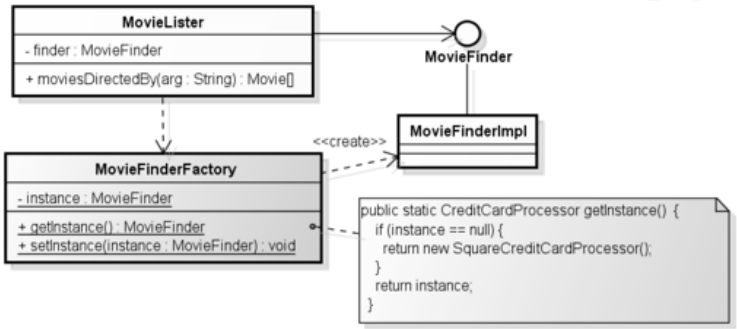
Problema

- Esempio *naive*



PROBLEMA

- Utilizzo di classi *factory*
  - Utilizza metodi statici nel recupero di istanze concrete ad interfacce
    - Si può fare di meglio...

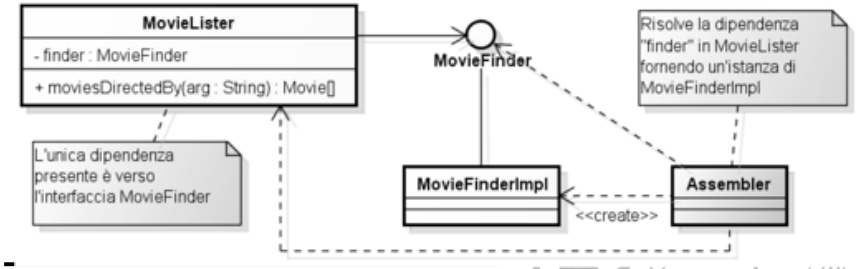


STRUTTURA

- *Inversion of Control*
  - Il ciclo di vita degli oggetti è gestito da una entità esterna (container)
    - Tutti i *framework* moderni implementano IoC
- *Dependency Injection* con IoC
  - Le dipendenze sono "iniettate" dal *container*
  - La componente dichiara solo le sue dipendenze
    - Minimizzazione del livello di accoppiamento
  - Diversi tipi di *injection*
    - *Constructor injection*
    - *Setter injection*

STRUTTURA

- *Dependency Injection* con IoC
  - `Assembler` risolve le dipendenze delle componenti
    - Il ciclo di vita degli oggetti è deciso dal *container*
    - Configurazione: meta-linguaggio (XML) o programmatica
      - Cambio di *movie finder* → Cambio di configurazione



SOLUZIONE

- *Constructor injection*
  - Dipendenze dichiarate come parametri del costruttore

```
public class MovieLister {
   private MovieFinder finder;
   public MovieLister(MovieFinder finder) { // Dichiarazioe dip.
      this.finder = finder;
   } // ...
}
```

  - Vantaggi
    - Permette di costruire oggetti validi sin dalla loro istanziazione
    - Favorisce la costruzione di oggetti immutabili
  - Svantaggi
    - *Telescoping*
    - A volte è difficile comprendere il significato dei parametri

SOLUZIONE

- *Setter injection*
  - Dipendenze dichiarate come metodi *setter*

```
public class MovieLister {
   private MovieFinder finder;
   public void setFinder(MovieFinder finder) {
      this.finder = finder;
   } // ...
}
```

  - Vantaggi
    - Individuazione precisa delle dipendenze e dei loro nomi
    - Lavora meglio con le gerarchie di classi
  - Svantaggi
    - La componente è costruita per passi
    - Non è abilitante all'immutabilità

GOOGLE GUICE

- *Lightweight DI framework*
  - *Class injection* (`@Inject` – JSR-330)
  - Configurazione Java estendendo `AbstractModule`

```
public class MovieModule extends AbstractModule {
   @Override
   protected void configure() {
      // Istruisce il container su come risolvere la dipendenza
      bind(MovieFinder.class).to(ColonMovieFinder.class);
   }
}

public class MovieLister {
   // L'annotazione istruisce il container di risolvere
   // automaticamente le dipendenze dichiarate
   @Inject
   public MovieLister(MovieFinder finder) {
      this.finder = finder;
   } // ...
}
```


GOOGLE GUICE

- *Lightweight DI framework*
  - *Method (setter) injection*

```
public class MovieLister {
   // L'annotazione istruisce il container su come risolvere
   // automaticamente le dipendenze dichiarate
   @Inject
   public setFinder(MovieFinder finder) {
      this.finder = finder;
   } // ...
}
```

  - *Field injection*

```
public class MovieLister {
   // Risolta automaticamente dall'injector. Piu' conciso ma
   // meno verificabile
   @Inject MovieFinder finder;
}
```


GOOGLE GUICE

- *Lightweight DI framework*
  - *Cointainer* è una struttura `Map<Class<?>, Object>`
  - Non è possibile avere due oggetti dello stesso tipo
    - È possibile utilizzare qualificatori ulteriori per questo caso
  - Si integra con AOP, Servlet spec., DAO, ...
    - @Named, @Singleton, @SessionScoperd, @RequestScoped...
      - https://github.com/google/guice/wiki/BindingAnnotations

```
public static void main(String[] args) {
   // Guice.createInjector() takes your Modules, and returns a new
   // Injector instance. Most applications will call this method
   // exactly once, in their main() method.
   Injector injector = Guice.createInjector(new MovieModule());
   // Now that we've got the injector, we can build objects.
   MovieLister lister = injector.getInstance(MovieLister.class);
}
```

SPRING

"Spring is a «lightweight» inversion of control and aspect oriented container framework"

- *Lightweight*
  - Non è intrusivo, gli oggetti sviluppati non dipendono da classi del *framework*
- *Framework/Container*
  - La configurazione utilizza XML, o annotazioni, o Java
    - Lo sviluppatore può concentrarsi sulla logica di *business*
  - *Cointainer* è una struttura `Map<String, Object>`
- Utilizza semplici POJO: *Bean*
  - Supporta sia *constructor,* che *setter injection*
- *JEE replacement*
  - *MVC, Front controller, DI, AOP, JDBC Template, Security ...*

SPRING

- Configurazione XML
  - `org.springframework.beans.factory.BeanFactory`
    - Implementazione del pattern *factory,* costruisce e risolve le dipendenze
  - `org.springframework.context.ApplicationContext`
    - Costruita sulla bean factory, fornisce ulteriori funzionalità
      - AOP, transazionalità, ...

```
ApplicationContext ctx =
   new ClassPathXmlApplicationContext(
         "com/springinaction/springidol/filename.xml");
// ...
MovieLister ml = (MovieLister) ctx.getBean("movieLister");
```


SPRING

- Configurazione XML
  - *Dependency injection* attraverso proprietà
    - Utilizzo metodi setter e getter

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
   <bean id="movieLister" class="MovieLister">
      <!-- La classe ha come proprieta' finder... -->
      <property name="finder" ref="movieFinder" />
   </bean>
   <bean id="movieFinder" class="ColonMovieFinder">
      <property name="file" value="movies.csv" />
   </bean>
</beans>
```


SPRING

- Configurazione XML
  - *Dependency injection* attraverso costruttori
    - Spring risolve automaticamente la scelta del costruttore

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
   <bean id="movieLister" class="MovieLister">
      <constructor-arg ref="movieFinder"/>
   </bean >
   <bean id="movieFinder" class="ColonMovieFinder">
      <constructor-arg value="movies.csv"/>
   </bean>
</beans>
```

- Oppure *factory methods, init,destroy methods...*

SPRING

- Configurazione Java
  - `@Configuration`: dichiara una classe configurazione
  - `@Bean`: dichiara un *bean*

```
@Configuration
public class MovieConfig {
   @Bean
   public MovieLister lister(){
      return new MovieLister( finder() );
   }

   @Bean
   public MovieFinder finder(){
      return new ColonMovieFinder("movies.csv");
   }
}
ApplicationContext ctx =
   new AnnotationConfigApplicationContext(MovieConfig.class);
```

Constructor injection

bean id

SPRING

- *Wiring* utilizzando annotazioni
  - Permette una gestione migliore della configurazione in progetti grandi
  - Introduce una dipendenza da *framework* esterni
    - `@Autowired`
    - `@Inject`, annotazione JSR-330
    - `@Resource`, annotazione JSR-250

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
</beans>
   <context:annotation-config/>
   <context:component-scan base-package="org.example"/>
   [...]
```


SPRING

- *Wiring* utilizzando annotazioni
  - `@Inject/@Autowired`
    - Ricerca per tipo il *bean* della proprietà e lo usa come **id**
    - Si utilizza su costruttori, proprietà, ...
    - Il bean non deve essere ambiguo
      - Disambiguazione tramite `@Named` per l'ID

```
@Inject
private Instrument instrument;

@Inject
@Named("guitar")  // Fornisce nome differente dal tipo
private Instrument instrument2;
```

- *Autodiscovery*
  - `@Component`, `@Controller`, `@Service`, ...

JAVASCRIPT

- *Dependency injection* in *Javascript*
  - *Container* è una mappa `[string, function]`
    - Linguaggio non tipizzato staticamente
    - Non ha interfacce esplicite
  - JS è strutturato a moduli (*module pattern*)
    - *Injection* di oggetti o interi moduli
  - Esistono diverse librerie (API) per la DI
    - RequireJS/AMD - http://requirejs.org/
    - Inject.js - http://www.injectjs.com/
    - AngularJS (*framework*) - https://angularjs.org/

ANGULARJS 1.6

- Javascript *framework*
  - *Client-side*
  - *Model-View-Whatever*
    - MVC per alcuni aspetti (controller)…
    - …MVVM per altri (*two-way data binding*)
  - Utilizza HTML come linguaggio di *templating*
    - Non richiede operazioni di DOM *refresh*
      - Controlla attivamente le azioni utente, eventi del *browser*
  - *Dependence injection*
    - Fornisce ottimi strumenti di *test*

ANGULARJS 1.6

- *Dependency Injection*
  - `$provide`: risolve le dipendenze fra le componenti
    - Viene invocato direttamente da Angular al *bootstrap*

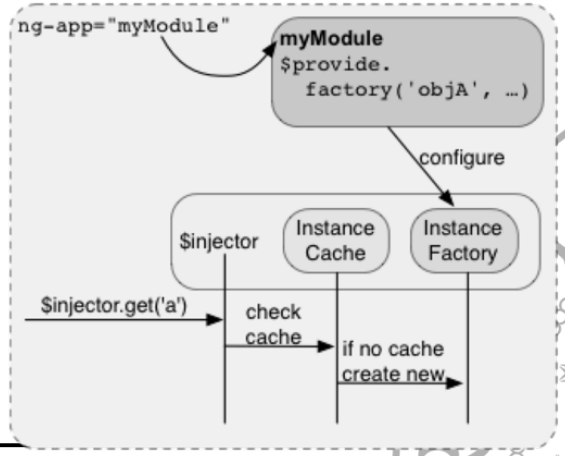
```
// Provide the wiring information in a module
angular.module('myModule', []).
  // Teach the injector how to build a 'greeter'
  // Notice that greeter itself is dependent on '$window'
  factory('greeter', function($window) {
    // This is a factory function, and is responsible for
    // creating the 'greet' service.
    return {
      greet: function(text) {
        $window.alert(text);
      }
    };
  });

// Request any dependency from the $injector
angular.injector(['myModule', 'ng']).get('greeter');
```

Ritorna il servizio `$provide`

"**Ricetta**" di come costruire `greeter`

ANGULARJS 1.6

- *Dependency Injection*



ANGULARJS 1.6

- *Dependency Injection*
  - *Factory methods*: costruiscono le componenti
    - Utilizzano *recipe* (ricette)

```
angular.module('myModule', []).
  config(['depProvider', function(depProvider){
    //...
  }]).
  factory('serviceId', ['depService', function(depService) {
    //...
  }]).
  directive('directiveName', ['depService', function(depService) {
    //...
  }]).
  filter('filterName', ['depService', function(depService) {
    //...
  }]).
  run(['depService', function(depService) {
    //...
  }]);
```

SCALA

- *Cake Pattern*
  - Utilizzo di una notazione dedicata *(self-type annotation)* e dei trait per dichiarazione delle dipendenze e loro risoluzione

```
trait FooAble {
  def foo() = "I am a foo!"
}
class BarAble { this: FooAble =>   // self-type annotation
  def bar = "I am a bar and " + foo()
}
val barWithFoo = new BarAble with FooAble   // mixin
```

  - Utilizzo di contenitori dedicati per risolvere le dipendenze
    - Divisione dei concetti di *business* dai dettagli tecnici

SCALA

```
// Declares a repository
trait UserRepositoryComponent {
  val userRepository: UserRepository
  class UserRepository {
    def findAll() = Array[User]()   // fake implementation
    def save(user: User) = "Saving a user..."   // fake implementation
  }
}
// Declares a service that needs a repository
trait UserServiceComponent { this: UserRepositoryComponent =>
  val userService: UserService
  class UserService {
    def findAll() = userRepository.findAll
    def save(user: User) = userRepository.save(user)
  }
}
// Configuration object that binds dependencies
object ComponentRegistry extends
  UserServiceComponent with UserRepositoryComponent {
  // Dependency injection
  val userRepository = new UserRepository
  val userService = new UserService
}
```


CONCLUSIONI

> *Dependency injection means giving an object its own instance variables. Really. That's it.*
>
> James Shore

- Vantaggi
  - Migliora la fase di *testing*
    - Isolamento delle componenti → Maggior riuso
  - Migliora la resilienza del codice
    - Manutenzione e correzioni componenti specifiche
  - Migliora la flessibilità
    - La disponibilità di più componenti simili permette di scegliere a *runtime* la migliore

RIFERIMENTI

- Design Patterns, Elements of Reusable Object Oriented Software, GoF, 1995, Addison-Wesley
- Inversion of Control Containers and the Dependency Injection pattern http://martinfowler.com/articles/injection.html
- Google Guice – Motivation https://github.com/google/guice/wiki/Motivation
- Spring vs Guice: The one critical difference that matters http://gekkio.fi/blog/2011-11-29-spring-vs-guice-the-one-critical-difference-that-matters.html
- Spring – The IoC Container http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html

RIFERIMENTI

- Angular Dependency Injection https://docs.angularjs.org/guide/di
- JavaScript Dependency Injection http://merrickchristensen.com/articles/javascript-dependency-injection.html
- Eat that cake! http://rcardin.github.io/design/2014/08/28/eat-that-cake.html
- Resolving your problems with Dependency Injection http://rcardin.github.io/programming/software-design/java/scala/di/2016/08/01/resolve-problems-dependency-injection.html

GITHUB REPOSITORY

https://github.com/rcardin/swe