UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

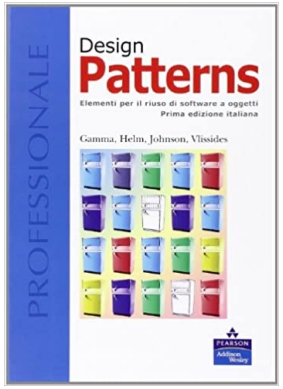
Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
http://jdirocco.github.io

Sommario

- Cosa è un design pattern
- Template
- Catalogo
- Pattern Creazionali
  - Singleton
  - Abstract Factory

Risorse

- Libro riferimento
  - Titolo: Design Patterns (Elementi per il riuso di software a oggetti)
  - Autori: Gamma, Helm, Johnson, Vlissides (GoF)
  - Lingua: Italiana
  - ISBN: 887192150
- Versione inglese
  - Titolo: Design Patterns: Elements of Reusable Object-Oriented Software
  - Autori: Gamma, Helm, Johnson and Vlissides
  - Casa Editrice: Addison-Wesley (1995)
  - ISBN: 0201633612



Cosa è: Cristopher Alexander

- Ogni pattern descrive
  - Un problema che si ripete più è più volte nel nostro ambiente
  - Il **core** di una soluzione al problema, dove tale soluzione può essere utilizzata un milione di volte senza **mai applicarla** nella stessa maniera
- Proviene dal mondo dell'architettura però patterns possono essere applicati a differenti aree tra cui lo sviluppo di software
- Ogni pattern è una regola di tre parti il quale esprime una relazione tra un certo **contesto**, un **problema** e una **soluzione**

Cosa è: GoF (1)

- Nome
  - Costituisce un nome simbolico per descrivere il pattern
  - Aiuta la comprensione poiché permette di ragionare ad un più alto livello di astrazione
  - Migliora la comunicazione tra sviluppatori poiché si ha un unico vocabolario
- Problema
  - Spiega il problema e il contesto
  - Descrive quando applicare il pattern
  - Potrebbero descrivere classi o strutture di oggetti che sono sintomatiche di un design inflessibile
  - Può includere lista di condizioni che devono essere rispettate per poter applicare il pattern

Cosa è: GoF (2)

- Soluzione
  - Descrive gli elementi facenti parte del pattern, le relazioni, responsabilità e collaborazioni
  - Non descrive una particolare soluzione poiché il pattern è un template che può essere applicato in differenti situazioni
  - Fornisce una descrizione astratta degli elementi che costituiscono la soluzione e non un design o implementazione concreta
- Conseguenze
  - Risultati e trade-off (pro e contro) nell'applicare il pattern
  - Riguardano problemi con un linguaggio di programmazione o con l'implementazione
  - Include eventuali impatti su affidabilità, portabilità, estendibilità

Benefici

- Cattura delle expertise e le rende accessibili anche ai non esperti
- Facilita la comunicazione tra sviluppatore fornendo un linguaggio comune
- Rende più facile il riuso di design di successo ed elimina alternative che diminuiscono il riuso
- Facilità modifiche al design
- Migliora la documentazione e la comprensione del design

Catalogo (1)

- Scopo: Cosa fa il pattern
  - Creazionale
    - Riguarda processo di creazione di oggetti
  - Strutturale
    - Focalizza attenzione su composizione di classi e oggetti
  - Comportamentale
    - Caratterizzano il modo nel quale classi o oggetti interagiscono e distribuiscono responsabilità
- Raggio di azione
  - Pattern applicato a classi o ad oggetti
  - Classe
    - Considera relazioni tra classi e loro sottoclassi stabilite attraverso ereditarietà (statica)
  - Oggetto
    - Considera relazioni tra oggetti che sono modificate a run-time e sono più dinamiche

Catalogo (2)

		Scopo		
		Creazionale	Strutturale	Comportamentale
Raggio d'azione	Classe	Factory Method	Adapter	Interpreter Template Method
	Oggetto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsability Command Iterator Mediator Memento Flyweight Observable State Strategy Visitor

Catalogo (3)

- Pattern di classe **creazionali** delegano parte del processo di creazione di un oggetto a sottoclassi, mentre quelli ad oggetti lo delegano ad altri oggetti
- Pattern di classe **strutturali** utilizzano l'ereditarietà per comporre classi, mentre quelli ad oggetti descrivono modi per raggruppare oggetti
- Pattern di classe **comportamentali** utilizzano ereditarietà per descrivere algoritmi e flusso di controllo, mentre quelli ad oggetti descrivono come gruppi di oggetti cooperano per eseguire un compito che un singolo oggetto non potrebbe portare a termine da solo

Pattern Creazionali

- Forniscono un'astrazione del processo di istanziazione degli oggetti e rendono il sistema indipendente da tale modalità
- Pattern creazionale su **classi** utilizza **ereditarietà** per scegliere la particolare classe da istanziare
- Pattern creazionale su **oggetti** delega **l'istanziazione** ad un altro oggetto
- Rendono il sistema maggiormente flessibile poiché conosce soltanto le interfacce degli oggetti definite mediante classi astratte

Singleton (1)

- Intento (Scopo)
  - Assicurare che una classe abbia una sola istanza e fornire un punto d'accesso globale a tale istanza
- Motivazione
  - Esempi
    - Diverse stampanti ma una sola coda di stampa
    - Unico window manager
  - Come si assicura l'univocità dell'istanza?
    - Stessa classe ha la **responsabilità** di creare le istanze

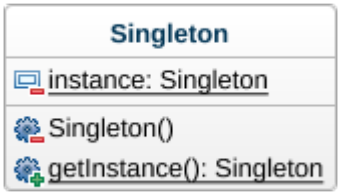
Singleton (2)

- Applicabilità
  - Quando deve esistere esattamente un'istanza di una classe e tale istanza deve essere accessibile ai *client* attraverso un punto di accesso noto a tutti gli utilizzatori
  - Quando l'unica istanza deve poter essere estesa attraverso la definizione di sottoclassi e i client devono essere in grado di utilizzare le istanze estese senza dover modificare il proprio codice

Singleton (3)

- Partecipanti
  - Singleton
    - Definisce un'operazione `getInstance` che consente ai client di accedere all'unica istanza esistente della classe
    - `getInstance` deve essere un'operazione di classe
    - Può essere responsabile della creazione della sua unica istanza
- Collaborazioni
  - *Client* possono accedere a un'istanza di un Singleton soltanto attraverso l'operazione `getInstance`

Struttura



Singleton (4)

- Implementazione
  - Assicurare l'esistenza di un'unica istanza
    - Costruttore privato
    - Variabile di classe privata
    - Metodo statico che restituisce la variabile di classe

Singleton: Soluzione (5)

```
public final class Singleton {
    private static Singleton instance = new Singleton( 10 );
    private int singletonData;
    private Singleton( int data ) {
        singletonData = data;
    }
    public void singletonOperation() {
        singletonData += 10;
    }
    public int getSingletonData() {
        return singletonData;
    }
    public static Singleton getInstance() {
        return instance;
    }
}
```

Non fa parte del pattern
Mostra che è comunque una qualsiasi classe con variabili e metodi

Singleton: Soluzione (6)

```
public class TestSingleton {

   public static void main( String[] args ) {
     Singleton s = Singleton.getInstance();

     s.singletonOperation();
     System.out.println( "Prima reference: " + s );
     System.out.println( "Valore del singleton data: " + s.getSingletonData() );

     s = null;
     s = Singleton.getInstance();
     System.out.println( "\nSeconda reference: " + s );
     System.out.println( "Valore del singleton data: " + s.getSingletonData() );
    }
}
```

Singleton: Soluzione (7)

› OUTPUT

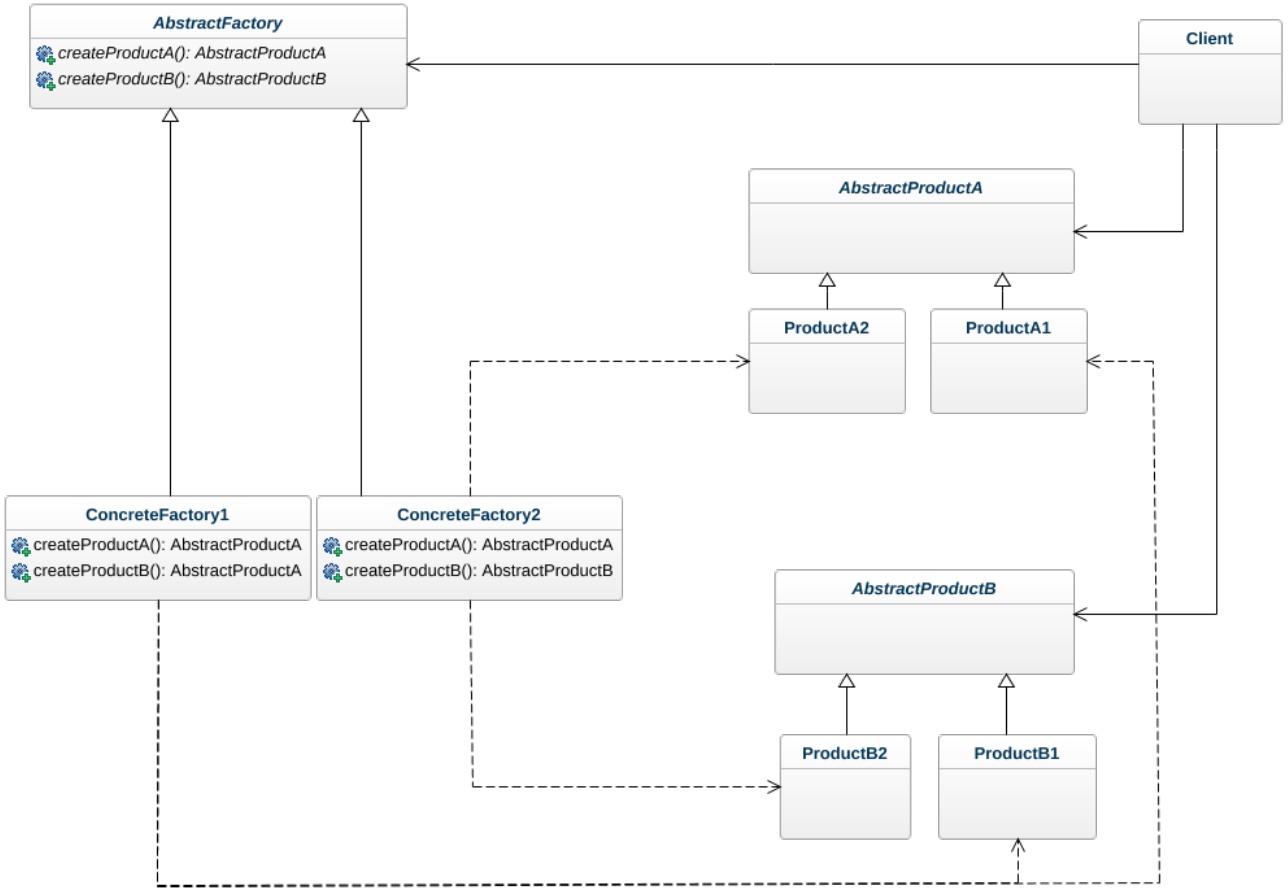
› Primo reference: `it.lab.pattern.singleton.Singleton@182f0db`
› Valore del singleton data: 20

› Seconda reference: `it.lab.pattern.singleton.Singleton@182f0db`
› Valore del singleton data: 20

Abstract Factory (1)

- Scopo
  - Fornire un'interfaccia per la creazione di **famiglie** di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete
- Motivazione
  - GUI toolkit che supporta diversi standard di look-and-feel (motif, presentation manager)
  - Diverse modalità di presentazione e comportamento per gli elementi (widget)
  - Per garantire portabilità gli elementi grafici di un look-and-feel non devono essere cablati nel codice

Abstract Factory (2)



Abstract Factory (3)

- Applicabilità
  - Sistema deve essere indipendente dalle modalità di creazione, composizione e rappresentazione dei suoi prodotti
  - Sistema deve poter essere configurato scegliendo una tra più famiglie di prodotti
  - Esistono diverse famiglie di prodotti che devono essere insieme
  - Si vuole libreria di classi che espone soltanto l'interfaccia e non l'implementazione

Abstract Factory (4)

- Partecipanti
  - AbstractFactory
    - Dichiara un'interfaccia per le operazioni di creazione di oggetti prodotto astratti
  - ConcreteFactory
    - Implementa le creazioni degli oggetti prodotto concreti
  - AbstractProduct
    - Dichiara un'interfaccia per una tipologia di oggetti prodotto
  - ConcreteProduct
    - Definisce un oggetto prodotto che dovrà essere creato dalla corrispondente factory concreta
    - Implementa l'interfaccia `AbstractProduct`
  - Client
    - Utilizza soltanto le interfacce dichiarate dalle classi `AbstractFactory` e `AbstractProduct`

Abstract Factory (5)

- Collaborazioni
  - Generalmente si crea una **singola** istanza di una classe `ConcreteFactory` durante l'esecuzione
  - Tale factory concreta gestisce la creazione di una famiglia di oggetti con un'implementazione specifica
  - `AbstractFactory` delega la creazione di oggetti prodotto alle sue sottoclassi `ConcreteFactory`
- Conseguenze
  - Isola classi concrete
  - Consente di **cambiare** in modo semplice famiglia di prodotti utilizzata
  - Promuove coerenza utilizzo di prodotti
  - Aggiunta del supporto di nuovi tipologie di prodotti difficile

Abstract Factory (6)

- Implementazione
  - Factory **come** Singleton
  - Creazioni dei prodotti
    - Ogni prodotto viene creato mediante un factory method
      - Diverse sottoclassi per ogni famiglia di prodotti
    - Pattern Prototype elimina necessità di diverse sottoclassi concrete di factory
    - Un unico metodo che restituisce un solo tipo di prodotto (sconsigliato e utilizzato per particolari linguaggi)
  - Definire factory estendibili ovvero un unico metodo con un parametro in ingresso che mi stabilisce il tipo del prodotto (meno sicuro)

Primo esempio abstract factory (1)

```
/*
 * GUIFactory example
 */
public abstract class GUIFactory {
	public static GUIFactory getFactory() {
		int sys = readFromConfigFile("OS_TYPE");
		if (sys == 0) {
			return new WinFactory();
		} else {
			return new OSXFactory();
		}
	}
	public abstract Button createButton();
}
```

```
class WinFactory extends GUIFactory {
	public Button createButton() {
		return new WinButton();
	}
}

class OSXFactory extends GUIFactory {
	public Button createButton() {
		return new OSXButton();
	}
}
```

Primo esempio abstract factory (2)

```
class WinFactory extends GUIFactory {
        public Button createButton() {
                return new WinButton();
        }
}
class OSXFactory extends GUIFactory {
        public Button createButton() {
                return new OSXButton();
        }
}
public abstract class Button {
        public abstract void paint();
}
```

```
class WinButton extends Button {
        public void paint() {
                System.out.println("Sono un
                        WinButton!");
        }
}
class OSXButton extends Button {
        public void paint() {
                System.out.println("Sono un
                        SXButton!");
        }
}
```

Primo esempio abstract factory (3)

```
public class Application {
        public static void main(String[]  args) {
                GUIFactory factory = GUIFactory.getFactory();
                Button button = factory.createButton();
                button.paint();
        }
        //  L'output sarà:
        //      "Sono un WinButton!"
        //  oppure:
        //      "Sono un OSXButton!"
}
```

Secondo esempio abstract factory (1)

```
public abstract class MyUnivaqBusinessFactory {


   private static MyUnivaqBusinessFactory factory =
                                        new RAMMyUnivaqBusinessFactoryImpl();

   //private static MyUnivaqBusinessFactory factory =
   //                                     new FileMyUnivaqBusinessFactoryImpl();


   public static MyUnivaqBusinessFactory getInstance() {

        return factory;

   }


   public abstract UtenteService getUtenteService();

   public abstract InsegnamentoService getInsegnamentoService();

   public abstract CorsoDiLaureaService getCorsoDiLaureaService();


}
```

Altra implementazione della factory

Prefisso `get` e non `create` per far intendere che anche i prodotti sono singleton

Secondo esempio abstract factory (2)

```
public interface UtenteService {

    Utente authenticate(String username, String password)
                    throws UtenteNotFoundException, BusinessException;

}

public interface InsegnamentoService {

    List<Insegnamento> findAllInsegnamenti(Docente docente)
                        throws BusinessException;

    List<Appello> findAllAppelli(Insegnamento insegnamento)
                        throws BusinessException;

    void createAppello(Appello appello) throws BusinessException;

    void updateAppello(Appello appello) throws BusinessException;

}

public interface CorsoDiLaureaService {

   List<CorsoDiLaurea> findAllCorsiDiLaurea() throws BusinessException;

  CorsoDiLaurea findCorsoDiLaureaById(int id) throws BusinessException;

}
```

Secondo esempio abstract factory (3)

```
public class RAMMyUnivaqBusinessFactoryImpl extends MyUnivaqBusinessFactory {
   private UtenteService utenteService;

   private InsegnamentoService insegnamentoService;

   private CorsoDiLaureaService corsoDiLaureaService;

   public RAMMyUnivaqBusinessFactoryImpl() {
      corsoDiLaureaService = new RAMCorsoDiLaureaServiceImpl();
      utenteService = new RAMUtenteServiceImpl(corsoDiLaureaService);
      insegnamentoService = new RAMInsegnamentoServiceImpl(corsoDiLaureaService);
   }
```

Implementazioni dei prodotti (services) modificate ovvero costruttore prende in input il prodotto (service) necessario

Secondo esempio abstract factory (4)

```
    @Override
    public UtenteService getUtenteService() {
      return utenteService;
    }

    @Override
    public InsegnamentoService getInsegnamentoService() {
       return insegnamentoService;
    }

    @Override
    public CorsoDiLaureaService getCorsoDiLaureaService() {
       return corsoDiLaureaService;
    }
}
```

Secondo esempio abstract factory (5)

› Versione vecchia

```
public class RAMUtenteServiceImpl implements UtenteService {
   private CorsoDiLaureaService corsoDiLaureaService;
   public RAMUtenteServiceImpl() {
       this.corsoDiLaureaService = new RAMCorsoDiLaureaServiceImpl();
   }
}
```

› Versione nuova

```
public class RAMUtenteServiceImpl implements UtenteService {
   private CorsoDiLaureaService corsoDiLaureaService;
   public RAMUtenteServiceImpl(CorsoDiLaureaService corsoDiLaureaService) {
       this.corsoDiLaureaService = corsoDiLaureaService;
   }
```

Secondo esempio abstract factory (6)

› Versione vecchia

```
public class RAMInsegnamentoServiceImpl implements InsegnamentoService {
   private CorsoDiLaureaService corsoDiLaureaService;
   public RAMInsegnamentoServiceImpl() {
       this.corsoDiLaureaService = new RAMCorsoDiLaureaServiceImpl();
   }
}
```

› Versione nuova

```
public class RAMInsegnamentoServiceImpl implements InsegnamentoService {
   private CorsoDiLaureaService corsoDiLaureaService;
   public RAMInsegnamentoServiceImpl(CorsoDiLaureaService corsoDiLaureaService) {
       this.corsoDiLaureaService = corsoDiLaureaService;
   }
```

Secondo esempio abstract factory (7)

```
public class FileMyUnivaqBusinessFactoryImpl extends MyUnivaqBusinessFactory {

   private UtenteService utenteService;

   private InsegnamentoService insegnamentoService;

   private CorsoDiLaureaService corsoDiLaureaService;

   private static final String REPOSITORY_BASE = "src" + File.separator + "main"
                        + File.separator + "resources" + File.separator + "dati";

   private static final String UTENTI_FILE_NAME = REPOSITORY_BASE
                        + File.separator + "utenti.txt";

   private static final String INSEGNAMENTI_FILE_NAME = REPOSITORY_BASE
                        + File.separator + "insegnamenti.txt";

   private static final String APPELLI_FILE_NAME = REPOSITORY_BASE
                        + File.separator + "appelli.txt";

   private static final String CORSI_DI_LAUREA_FILE_NAME = REPOSITORY_BASE
                        + File.separator + "corsidilaurea.txt";
```

Secondo esempio abstract factory (8)

```
public FileMyUnivaqBusinessFactoryImpl() {

     corsoDiLaureaService =
                 new FileCorsoDiLaureaServiceImpl(CORSI_DI_LAUREA_FILE_NAME);

     utenteService = new FileUtenteServiceImpl(UTENTI_FILE_NAME,
                              corsoDiLaureaService);

     insegnamentoService = new FileInsegnamentoServiceImpl(
                                   INSEGNAMENTI_FILE_NAME, APPELLI_FILE_NAME,
                                   corsoDiLaureaService);

}

@Override

public UtenteService getUtenteService() {

    return utenteService;

}

@Override

public InsegnamentoService getInsegnamentoService() {

    return insegnamentoService;

}
```

Implementazioni dei prodotti (services) modificate ovvero costruttore prende in input il prodotto (service) necessario

Secondo esempio abstract factory (9)

```
    @Override
    public CorsoDiLaureaService getCorsoDiLaureaService() {
      return corsoDiLaureaService;
    }
}
```

Secondo esempio abstract factory (10)

- Ai costruttori delle varie implementazioni dei service
  - Viene passato come parametro i nomi dei files e non viene più definito dentro il singolo service
  - Il service (ovvero il `FileCorsoDiLaureaServiceImpl`) che viene utilizzato nell'implementazione (ovvero dentro `FileUtenteServiceImpl` e `FileInsegnamentoServiceImpl`) viene passato come parametro