Elaborato di Ingegneria del Software

Claudia Manfredi
Mattia Pavlovic
Elena Tonini

Università degli Studi di Brescia

A.A. 2021/2022

Total slides: 27

Elaborato di Ingegneria del Software

Claudia Manfredi
Mattia Pavlovic
Elena Tonini

Università degli Studi di Brescia

A.A. 2021/2022

Sommario I

Sommario I

1. Testing della Versione 5 2
2. Pattern MVC 9
3. Pattern GRASP 10
4. Principi SOLID 11
5. Pattern della Gang of Four 12
6. Refactoring 13
7. Testing finale 20

Testing della Versione 5

Total slides: 27

Testing della Versione 5

Test Classe Categoria

Test di classe

Per il test di classe della versione 5 abbiamo selezionato la classe Categoria in quanto permetteva di effettuare in modo efficiente test su numerosi metodi.

Sarebbe diversamente stato complesso testare classi che richiedessero l'input da parte dell'utente.

Test Classe Categoria
Test di classe

Per il test di classe della versione 5 abbiamo selezionato la classe Categoria in quanto permetteva di effettuare in modo efficiente test su numerosi metodi.

Sarebbe diversamente stato complesso testare classi che richiedessero l'input da parte dell'utente.

Test Classe Categoria

Test di classe


```java
class CategoriaTest {

    12 usages
    Nodo n = new Nodo( _nome: "Nodo",  _descrizione: "categoria nodo");
    5 usages
    Foglia f = new Foglia( _nome: "Foglia",  _descrizione: "categoria foglia");
    2 usages
    Foglia f1 = new Foglia( _nome: "Foglia",  _descrizione: "categoria foglia");
    1 usage
    Foglia f2 = new Foglia( _nome: "Foglia",  _descrizione: "categoria foglia");


    eletoo
    @Test
    void isNameTaken() {
        n.addChild(f);
        assertTrue(n.isNameTaken("Foglia"));
    }

    eletoo
    @Test
    void nodeStructureIsInvalidIfHasNoChildren() { assertFalse(n.isStructureValid()); }
```


Test Classe Categoria

Test di classe

Disclaimer: il codice della versione 5 era particolarmente disorganizzato e non rispettava alcun pattern.
Con la versione finale post-refactoring è chiaramente più facile testare in modo più ordinato non solo questa classe ma anche tutte le altre.

I test effettuati hanno nomi esplicativi relativamente al fine del test stesso.

Test Classe Categoria


```
@Test
void nodeStructureIsInvalidIfHasOneChild(){
    n.addChild(f);
    assertFalse(n.isStructureValid());
}

eletoo
@Test
void nodeStructureIsValidIfHasTwoChildren(){
    n.addChild(f);
    n.addChild(f1);
    assertTrue(n.isStructureValid());
}

eletoo
@Test
void nodeStructureIsValidIfHasMoreThanTwoChildren(){
    n.addChild(f);
    n.addChild(f1);
    n.addChild(f2);
    assertTrue(n.isStructureValid());
}
```

```
@Test
void leafStructureIsValid() { assertTrue(f.isStructureValid()); }
```


Test Classe Categoria

Test Classe Orario

Test di requisiti e funzionalità

Per il test delle funzionalità abbiamo selezionato la funzionalità:
"Gli orari per effettuare lo scambio possono essere solo allo scoccare dell'ora o alla mezza (ogni trenta minuti)"

Abbiamo applicato volutamente un testing "superficiale" sulla versione 5 in modo da poter mostrare l'evoluzione tra l'inizio e la fine del progetto della tecnica utilizzata per individuare i casi di test

Test Classe Orario
Test di requisiti e funzionalità

Per il test delle funzionalità abbiamo selezionato la funzionalità:
"Gli orari per effettuare lo scambio possono essere solo allo scoccare dell'ora o alla mezza (ogni trenta minuti)"

Abbiamo applicato volutamente un testing "superficiale" sulla versione 5 in modo da poter mostrare l'evoluzione tra l'inizio e la fine del progetto della tecnica utilizzata per individuare i casi di test

Test Classe Orario

Test di requisiti e funzionalità


```java
package tests;

import ...

1 usage  eletoo
class OrarioTest {
    3 usages
    Orario o1 = new Orario( hour: 10,  minutes: 00);
    3 usages
    Orario o2 = new Orario( hour: 10,  minutes: 10);
    3 usages
    Orario o3 = new Orario( hour: 10,  minutes: 30);
    3 usages
    Orario o4 = new Orario( hour: 24,  minutes: 00);
    3 usages
    Orario o5 = new Orario( hour: 00,  minutes: 00);

    eletoo
    @Test
    void tenIsvalidTime() { assertTrue(o1.isValid(o1.getHour(), o1.getMinutes())); }

    eletoo
    @Test
    void tenPastTenIsInvalidTime() { assertFalse(o2.isValid(o2.getHour(), o2.getMinutes())); }
```


Test Classe Orario
Test di requisiti e funzionalità

Inizialmente i casi di test individuati erano in numero ridotto e poco approfonditi, mentre il testing della versione finale post-refactoring, come si vedrà nella sezione finale della presentazione, è stato effettuato in maniera più precisa e tenendo conto delle linee guida per l'individuazione dei casi di test black box.

Test Classe Orario


```
eletoo
@Test
void tenThirtyIsValidTime() { assertTrue(o3.isValid(o3.getHour(), o3.getMinutes())); }

eletoo
@Test
void midnightAtTwentyfourIsValidTime() { assertTrue(o4.isValid(o4.getHour(), o4.getMinutes())); }

eletoo
@Test
void midnightIsValidTime() { assertTrue(o5.isValid(o5.getHour(), o5.getMinutes())); }
```


Test Classe Orario

Pattern MVC

Total slides: 27

Pattern GRASP

Principi SOLID

Total slides: 27

Pattern della Gang of Four

Pattern della Gang of Four

Refactoring

Pattern di Refactoring

Tra i numerossissimi pattern di refactoring utilizzati nello sviluppo dell'applicazione abbiamo selezionato solo i più rilevanti (sia dal punto di vista della frequenza di utilizzo sia dal punto di vista degli effetti positivi che la loro applicazione ha sul codice).

Tali pattern sono:

- Extract Class
- Extract Method
- Replace Temp with Query
- Extract Constant

Mostreremo nel dettaglio solo alcuni di essi.

Pattern di Refactoring

Tra i numerossissimi pattern di refactoring utilizzati nello sviluppo dell'applicazione abbiamo selezionato solo i più rilevanti (sia dal punto di vista della frequenza di utilizzo sia dal punto di vista degli effetti positivi che la loro applicazione ha sul codice).

Tali pattern sono:
- Extract Class
- Extract Method
- Replace Temp with Query
- Extract Constant

Mostreremo nel dettaglio solo alcuni di essi.

Pattern di Refactoring

Pattern Extract Class


```java
public static void main(String args[]) throws IOException {
    Controller controller = new Controller();
    View view = new View();

    Path path1 = Paths.get( first: System.getProperty("user.dir") + "/db/conf");
    Path path2 = Paths.get( first: System.getProperty("user.dir") + "/db/jsonFiles");

    if (!Files.isDirectory(path1) || !Files.isDirectory(path2)) {
        Files.createDirectories(path1);
        Files.createDirectories(path2);
    }

    String val;
    do {
        val = view.in( prompt: "Seleziona un'opzione: \n1. Accedi\n2. Registrati\n3. Esci");
        switch (val) {
            case "1": {...}
            break;
            case "2": {...}
            break;
            case "3": {...}
            break;
            default:
                view.errorMessage(View.ErrorMessage.E_UNAUTHORIZED_CHOICE);
        }
    } while (!val.contentEquals( cs: "3"));
}
```


Pattern di Refactoring

Pattern Extract Class

Il nostro metodo `main` nella versione 5 aveva la struttura presentata nell'immagine.
Chiaramente esso violava svariati principi di buona programmazione (primo tra tutti SRP, in quanto si occupava sia di creare le directories che di permettere all'utente di effettuare la scelta di un'azione da eseguire).

Abbiamo suddiviso le responsabilità creando una classe `LocalPath` che si occupa della creazione delle directories in cui creare e recuperare i file `json` di configurazione e delle gerarchie.

Pattern di Refactoring

Pattern Extract Class


```java
public class LocalPath {

    public static void createLocalDirectories() throws IOException {
        Path path1 = Paths.get( first: System.getProperty("user.dir") + "/db/conf");
        Path path2 = Paths.get( first: System.getProperty("user.dir") + "/db/jsonFiles");

        if (!Files.isDirectory(path1) || !Files.isDirectory(path2)) {
            Files.createDirectories(path1);
            Files.createDirectories(path2);
        }
    }
}
```


Pattern di Refactoring

Pattern Extract Class

Nella classe `LocalPath` abbiamo creato un metodo statico contenente il codice necessario per creare le directories. In questo modo il metodo potrà essere chiamato direttamente nel `main` invocandolo dalla classe.

Pattern di Refactoring

Pattern Extract Method e Replace Temp with Query


```java
public static void createLocalDirectories() throws IOException {

    if (!Files.isDirectory(getConfigurationFileDirectory()) || !Files.isDirectory(getJsonFilesDirectory())) {
        Files.createDirectories(getConfigurationFileDirectory());
        Files.createDirectories(getJsonFilesDirectory());
    }
}

private static @NotNull Path getConfigurationFileDirectory() {
    return Paths.get( first: System.getProperty("user.dir") + "/db/conf");
}

private static @NotNull Path getJsonFilesDirectory() {
    return Paths.get( first: System.getProperty("user.dir") + "/db/jsonFiles");
}
```


Pattern di Refactoring

Pattern Extract Method e Replace Temp with Query

Abbiamo poi estratto i metodi per individuare il *path* delle directories, sempre all'interno della classe `LocalPath`.
Si tratta di un'applicazione del pattern di refactoring "Extract Method".

Inoltre, anziché creare delle variabili temporanee per salvare localmente i *path* trovati, abbiamo applicato la tecnica di refactoring "Replace temp with Query".

Ciò è stato possibile in virtù del fatto che i metodi per ricavare i *path* sono idempotenti, pertanto la loro invocazione ripetuta non ha effetti collaterali sul sistema.

Pattern di Refactoring

Pattern Extract Constant


```java
public class LocalPath {

    private static final String DB_CONF = "/db/conf";
    private static final String DB_JSON_FILES = "/db/jsonFiles";

    private static @NotNull Path getConfigurationFileDirectory() {
        return Paths.get( first: System.getProperty("user.dir") + DB_CONF);
    }

    private static @NotNull Path getJsonFilesDirectory() {
        return Paths.get( first: System.getProperty("user.dir") + DB_JSON_FILES);
    }

    public static void createLocalDirectories() throws IOException {
        if (!Files.isDirectory(getConfigurationFileDirectory()) || !Files.isDirectory(getJsonFilesDirectory())) {
            Files.createDirectories(getJsonFilesDirectory());
            Files.createDirectories(getConfigurationFileDirectory());
        }
    }
```


Pattern di Refactoring

Pattern Extract Constant

Abbiamo poi rapidamente estratto le costanti `final` per indicare i *path* in modo da poter in futuro consentire modifiche ai *path* stessi accedendo a un solo punto dell'applicazione.
In questo caso è stato applicato il pattern di refactoring "Extract constant".

La classe LocalPath è così ultimata (a meno di modifiche introdotte solo successivamente per la gestione della creazione delle liste di *path* delle directories dei file `json` da caricare).

Pattern di Refactoring


```java
public static void main(String args[]) throws IOException {
    Controller controller = new Controller();
    View view = new View();

    LocalPath.createLocalDirectories();

    String val;
    do {
        val = view.in( prompt: "Seleziona un'opzione: \n1. Accedi\n2. Registrati\n3. Esci");
        switch (val) {
            case "1": {...}
            break;
            case "2": {...}
            break;
            case "3": {...}
            break;
            default:
                view.errorMessage(View.ErrorMessage.E_UNAUTHORIZED_CHOICE);
        }
    } while (!val.contentEquals( cs: "3"));
}
```


Pattern di Refactoring

Infine abbiamo richiamato il metodo statico della classe appena creata nel metodo `main`.

Testing finale

Total slides: 27

Testing finale

Testing finale

Abbiamo scelto di effettuare a seguito del refactoring gli stessi test effettuati sulla versione 5, in modo da verificare che le funzionalità corrette dell'applicazione rimanessero tali anche dopo il refactoring.

In particolare, i casi di test per la verifica dei requisiti e delle funzionalità sono stati scelti basandosi in modo più rigoroso rispetto al testing iniziale sulle linee guida che suggeriscono di effettuare test su:

- Classi di equivalenza
- Casi limite

Testing finale

Abbiamo scelto di effettuare a seguito del refactoring gli stessi test effettuati sulla versione 5, in modo da verificare che le funzionalità corrette dell'applicazione rimanessero tali anche dopo il refactoring.

In particolare, i casi di test per la verifica dei requisiti e delle funzionalità sono stati scelti basandosi in modo più rigoroso rispetto al testing iniziale sulle linee guida che suggeriscono di effettuare test su:

- Classi di equivalenza
- Casi limite

Test Classe Category

Test di classe


```
class CategoryTest {

    12 usages
    Node n = new Node( _nome: "Nodo",  _descrizione: "categoria nodo");
    5 usages
    Leaf f = new Leaf( _nome: "Foglia",  _descrizione: "categoria foglia");
    2 usages
    Leaf f1 = new Leaf( _nome: "Foglia",  _descrizione: "categoria foglia");
    1 usage
    Leaf f2 = new Leaf( _nome: "Foglia",  _descrizione: "categoria foglia");


    eletoo
    @Test
    void isNameTaken() {
        n.addChild(f);
        assertTrue(n.isNameTaken("Foglia"));
    }

    eletoo
    @Test
    void nodeStructureIsInvalidIfHasNoChildren() { assertFalse(n.isStructureValid()); }

    eletoo
    @Test
    void nodeStructureIsInvalidIfHasOneChild(){
        n.addChild(f);
        assertFalse(n.isStructureValid());
    }
```


Test Classe Category

Test di classe

Il test sulla classe `Categoria` non necessitava di subire particolari variazioni dal momento che gli unici cambiamenti apportati al codice che influissero sui test sono stati relativi ai nomi della classe `Categoria` in `Category`, `Foglia` in `Leaf` e `Nodo` in `Node`.

Test Classe Category


```
@Test
void nodeStructureIsValidIfHasTwoChildren(){
    n.addChild(f);
    n.addChild(f1);
    assertTrue(n.isStructureValid());
}

 eletoo
@Test
void nodeStructureIsValidIfHasMoreThanTwoChildren(){
    n.addChild(f);
    n.addChild(f1);
    n.addChild(f2);
    assertTrue(n.isStructureValid());
}

 eletoo
@Test
void leafStructureIsValid() { assertTrue(f.isStructureValid()); }
```


Test Classe Category

Test Classe Time

Test di requisiti e funzionalità


```
class TimeTest {

    //ora valida, test sui minuti
    Time o1 = new Time( hour: 10,  minutes: 00);
    Time o2 = new Time( hour: 10,  minutes: 01);
    Time o3 = new Time( hour: 10,  minutes: 59);
    Time o4 = new Time( hour: 24,  minutes: 00);
    Time o5 = new Time( hour: 00,  minutes: 00);
    Time o13 = new Time( hour: 10,  minutes: -1);
    Time o14 = new Time( hour: 10,  minutes: 60);
    Time o6 = new Time( hour: 10,  minutes: 30);
    Time o7 = new Time( hour: 10,  minutes: 29);
    Time o8 = new Time( hour: 10,  minutes: 31);
```


Test Classe Time

Test di requisiti e funzionalità

Per il test delle funzionalità abbiamo mantenuto la scelta della funzionalità individuata per il testing iniziale:
"Gli orari per effettuare lo scambio possono essere solo allo scoccare dell'ora o alla mezza (ogni trenta minuti)"

La logica applicata per individuare i casi di test è quella delle classi di equivalenza e dei casi limite.

Nella slide a fianco sono mostrati i casi di test individuati considerando orario valido e minuti invalidi.

Test Classe Time


```
@Test
void tenIsvalidTime() { assertTrue(o1.isValid()); }

@Test
void onePastTenIsInvalidTime() { assertFalse(o2.isValid()); }

@Test
void tenFiftynineIsInvalidTime() { assertFalse(o3.isValid()); }

@Test
void midnightAtTwentyfourIsValidTime() { assertTrue(o4.isValid()); }

@Test
void midnightIsValidTime() { assertTrue(o5.isValid()); }

@Test
void tenAndMinusOneIsInvalid() { assertFalse(o13.isValid()); }

@Test
void tenSixtyIsInvalidTime() { assertFalse(o14.isValid()); }

@Test
void tenThirtyIsValidTime() { assertTrue(o6.isValid()); }

@Test
void tenTwentyNineIsInvalidTime() { assertFalse(o7.isValid()); }

@Test
void tenThirtyOneIsInvalidTime() { assertFalse(o8.isValid()); }
```


Test Classe Time

Nella slide a fianco sono mostrati i metodi che testano i casi individuati considerando orario valido e minuti invalidi.

Test Classe Time

Test di requisiti e funzionalità


```
//minuti validi, test sulle ore
Time o9 = new Time( hour: 01,  minutes: 00);
Time o10 = new Time( hour: 23,  minutes: 00);
Time o11 = new Time( hour: 25,  minutes: 00);
Time o12 = new Time( hour: -1,  minutes: 00);

@Test
void oneOClockIsValidTime() { assertTrue(o9.isValid()); }

@Test
void twentyThreeIsValidTime() { assertTrue(o10.isValid()); }

@Test
void twentyFiveIsInvalidTime() { assertFalse(o11.isValid()); }

@Test
void minusOneIsInvalidTime() { assertFalse(o12.isValid()); }
```


Test Classe Time
Test di requisiti e funzionalità

Nella slide a fianco sono mostrati i casi di test individuati considerando orario invalido e minuti validi.