



PROGETTO
SPME

2022

Legacy System - Refactoring, Evoluzione ed Integrazione con un Servizio di CI/CD

Alessandro Quirile N97/402

Università degli Studi di Napoli Federico II
Laurea Magistrale in Informatica LM-18

Indice

1	Introduzione	5
2	Azioni preliminari	6
2.1	Git	6
2.2	Jenkins	6
2.2.1	Installazione	6
2.2.2	Configurazione	9
3	The Gilded Rose	17
3.1	Refactoring	20
3.1.1	Program comprehension	20
3.1.2	Analisi dei bad smell	22
3.2	Risolvere i bad smell	25
3.2.1	Il package <code>exceptions</code>	28
3.2.2	Il package <code>factories</code>	29
3.2.3	Il package <code>test.factories</code>	31
3.2.4	Il package <code>interfaces</code>	33
3.2.5	Il package <code>test.interfaces</code>	35
3.2.6	Il package <code>implementations</code>	37
3.2.7	Il package <code>test.implementations</code>	41
3.2.8	Il package <code>models</code>	48
3.2.9	Il package <code>utils</code>	49
3.3	Risultato finale	51
3.4	Evolution	52
3.4.1	Modellare oggetti Conjured	52
4	Visualizzazione dei risultati su Jenkins	54

<i>INDICE</i>	3
5 Considerazioni finali e soluzioni alternative	63

"Clean code always looks like it was written by someone who cares"
- Michael Feathers

Capitolo 1

Introduzione

L'obiettivo di questo documento è fornire una chiave di lettura per spiegare in che modo ho rifattorizzato il legacy system *The Gilded Rose*¹ – giustificando le scelte e illustrando approcci alternativi – e come ho integrato l'intero processo con un server di CI/CD che sfrutta Jenkins². La prima parte della relazione sarà incentrata sull'installazione e sul setup degli strumenti necessari, mentre la seconda parte verterà sul refactoring e sull'evoluzione del sistema in accordo ai requisiti funzionali che verranno illustrati più avanti. L'intero processo, inoltre, è stato gestito tramite un tool di build automatico come Gradle³ ed interamente versionato tramite Git⁴. I topic coinvolti in questo progetto sono molti di quelli visti a lezione durante l'A.A. 2021/2022:

1. Manutenzione di un sistema (legacy)
2. Evoluzione di un sistema (legacy)
3. Principi SOLID e Design Pattern
4. Unit testing: jUnit
5. Clean Code
6. Strumenti di build automatico: Gradle
7. Strumenti VCS: Git
8. Strumenti di CI/CD: Jenkins

Tutto il codice sorgente è accessibile liberamente a questo URL:
<https://github.com/alessandroquirile/The-Gilded-Rose.git>

¹<https://github.com/emilybache/GildedRose-Refactoring-Kata.git>

²<https://www.jenkins.io>

³<https://gradle.org>

⁴<https://git-scm.com>

Capitolo 2

Azioni preliminari

Iniziamo ad installare l'occorrente: Git e Jenkins.

2.1 Git

Per installare Git ho usato il package manager Homebrew¹ attraverso il comando `brew install git` che installerà la versione più recente di questo tool, al momento la 2.38.1. Per verificare che Git sia stato installato correttamente è sufficiente lanciare dalla shell il comando `git --version`.

2.2 Jenkins

Si tratta di un sistema distribuito per implementare la *continuous integration* (o estensioni) in esecuzione su un server web che gestisce automaticamente, previa configurazione, la pipeline del progetto orchestrandone l'intera catena di CI/CD. Al momento Jenkins richiede Java 11 o Java 17 per poter funzionare.

2.2.1 Installazione

Per installare la versione Long-Term Support di Jenkins si può usare il comando `brew install jenkins-lts`. Seguirà un wall-of-text di informazioni al seguito del quale potremmo digitare, sempre da shell, il comando `brew services start jenkins-lts` per lanciare il servizio sul server. La socket che viene configurata sarà in `localhost` in ascolto sulla porta 8080 per richieste `http` da parte di un client.

¹<https://brew.sh>

La restante parte della configurazione deve essere fatta dal browser: occorre visitare la pagina `http://localhost:8080` e ci troveremo davanti alla seguente schermata:

Getting Started


Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

```
/Users/administrator/.jenkins/secrets/initialAdminPassword
```

Please copy the password from either location and paste it below.

Administrator password



Continue

Sarà sufficiente lanciare dal terminale il comando

```
cat /Users/administrator/.jenkins/secrets/initialAdminPassword
```

per ottenere la password amministratore iniziale ed incollarla nel form. A questo punto potremo installare alcuni plugin suggeriti dal sistema e creare un account amministratore.



Welcome to Jenkins!

Username

Password

☐ Keep me signed in

Sign in

Sarà possibile interrompere, o riavviare, il servizio Jenkins sul server lanciando dalla shell il comando `brew services stop jenkins-lts` oppure `brew services restart jenkins-lts`. Tut-

ti questi comandi sono consultabili dalla documentazione ufficiale di Homebrew.

2.2.2 Configurazione

Sto usando la versione 19.0.1 di Java e la versione 7.6 del wrapper di Gradle, coerentemente alla matrice di compatibilità dei due tool².

Una volta che abbiamo clonato la repo originale di Emily Bache nel nostro workspace attraverso il comando `git clone`, bisogna fare il `git push` del nostro codebase su una repo di GitHub, oppure fare direttamente una `git fork`.

Collegare la repository con Jenkins

A questo punto possiamo fare il binding della nostra repository con Jenkins: dalla homepage del browser creiamo un nuovo progetto (*freestyle item*) e, poiché stiamo utilizzando Git come VCS, incolliamo l'URL GitHub del progetto nel relativo campo. In seguito possiamo scegliere il *Trigger* che Jenkins userà nella fase di build automatico: dobbiamo decidere cioè *cosa* farà sì che Jenkins lanci il thread relativo alla build del nostro progetto. Io ho scelto *Poll SCM* che consiste in un'interrogazione periodica da parte di Jenkins sulla repo di GitHub per rilevare se c'è stata una modifica rispetto allo snapshot precedente. Alternativamente sarebbe stato possibile lanciare script da remoto, ad esempio funzioni in Python oppure bash. Adesso dobbiamo decidere cosa fare quando il *Trigger* rileva una modifica: Jenkins ci consente di scegliere se eseguire un comando dalla shell, ad esempio `javac Hello.java` e `java Hello`, oppure affidarci ad uno strumento che automatizza questi processi (come Gradle, Maven o Ant) – di gran lunga preferiti per una serie di ragioni, ad esempio per evitare la cosiddetta *dependency hell* con i link, in fase di compilazione e di esecuzione, delle librerie necessarie al progetto. Ho scelto Gradle ed in particolare ho scelto di utilizzare il wrapper `gradlew` che lancerà il task `test` per compilare ed eseguire l'intera test suite ed infine, come *Post-build Action*, ho richiesto a Jenkins di mostrarmi il report dei test appena eseguiti che vengono generati da Gradle sotto forma di file `xml`. Questa sarà la nostra pipeline di Jenkins.

Alternativamente sarebbe stato possibile creare un file `Jenkinsfile` da inserire nella workspace del progetto che stabilisca, attraverso una sintassi *dichiarativa*, la pipeline di Jenkins; in questo caso è necessario che il file `Jenkinsfile` venga spinto, ad esempio tramite il comando `git push` nella repository remota ospitata, ad esempio, su GitHub. Questo implica che `Jenkinsfile` è un file versionabile e si può quindi tracciare ogni sua modifica come un qualsiasi altro artefatto. Usare un `Jenkinsfile` è assolutamente equivalente a configurare la pipeline dal browser, come è stato fatto nel nostro caso; in alcuni contesti, tuttavia, può essere più comodo usare il `Jenkinsfile` perché

²<https://docs.gradle.org/current/userguide/compatibility.html>

sfrutta la sintassi **groovy** (la stessa di Gradle) e, aprendo il file in un editor – come ad esempio Visual Studio Code – avremmo le parole-chiave evidenziate, permettendo quindi una scrittura (ed una lettura) più comoda. Infine usare un **Jenkinsfile** permette di avere un singolo *entry point*, nel senso che tutte le modifiche possono essere fatte scrivendo direttamente sul file, anziché interfacciarsi ad una pagina web per mezzo di un browser che funziona tramite script.

Effettuato il binding tra la repository e Jenkins possiamo forzare la build cliccando il tasto **Build Now** sull'interfaccia grafica oppure fare un **git push** ed aspettare i secondi necessari affinché Jenkins rilevi questa modifica. A questo punto potremmo vedere i risultati dei test di unità del nostro progetto. Mi aspetto che il test fallisca perché il progetto viene consegnato con una test suite che inizialmente fallisce:

 **Build #2 (Dec 2, 2022, 5:04:15 PM)**



No changes.



Started by user [admin](#)



Revision: bf9e927bc823fbdd462e7c4077d24a2ff1e2d961
Repository: <https://github.com/alessandroquirile/The-Gilded-Rose.git>

- origin/main



Test Result (1 failure)
[GildedRoseTest.foo\(\)](#)

Come is nota, `GildedRoseTest.foo()` ha fallito siccome la repository originale contiene un unico metodo che intenzionalmente *deve* fallire, allora possiamo vedere questo risultato da Jenkins:

Test Result

1 failures

1 tests

Took 9 ms.

Add description

All Failed Tests

Test Name	Duration	Age
+ GildedRoseTest.foo()	9 ms	1

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
(root)	9 ms	1	+1	0		0		1	+1

Se clicchiamo sul nome di un metodo di test, come ad esempio quello di inizializzazione chiamato `GildedRoseTest.foo()` ci viene addirittura illustrato cosa è andato storto e perché:

Failed

GildedRoseTest.foo()

Failing for the past 1 build (Since  #2)

Took 9 ms.

Add description

Error Message

```
org.opentest4j.AssertionFailedError: expected: <fixme> but was: <foo>
```

Questo completa la fase di setup del progetto e del server di CI/CD: siamo sicuri che essi comunichino correttamente. Ci resta soltanto da fare il refactoring del sistema.

La configurazione di Jenkins che ho impostato effettua quindi un controllo periodico per veri-

ficare se è stato fatto un push oppure no. In caso affermativo viene lanciato il comando `./gradlew test` per compilare ed eseguire la test suite ed infine mostra i risultati dei test sulla GUI. Sarebbe stato possibile fare tanto altro, ad esempio implementare un servizio di reportistica che notifica, tramite mail, il risultato di una build. Oppure sarebbe stato possibile lanciare script bash da remoto o addirittura costruire una vera e propria catena di task di gradle: per esempio sarebbe stato possibile, nel caso in cui i test fossero andati *tutti* a buon fine, costruire il file `jar` della mia applicazione a livello di produzione. I task in Gradle dipendono dal tipo (o dai tipi) di plugin associato al progetto: il mio è un progetto `java`, quindi i task che il tool mi mette a disposizione sono tutti e soli i seguenti:

```
Build:
  assemble
  build
  buildDependencies
  buildNeeded
  classes
  clean
  jar
  testClasses
Build Setup:
  init
  wrapper
Documentation:
  javadoc
Help:
  buildEnvironment
  dependencies
  dependenciesInsight
  help
  javaToolchains
  outgoingVariants
  projects
  resolvableConfigurations
  tasks
Other:
  compileJava
  compileTestJava
  components
  dependentComponents
  model
```

```
prepareKotlinBuildScriptModel
processResources
processTestResources
Verification:
  check
  test
```

Questi task sono tutti consultabili dalla shell attraverso il comando `./gradlew tasks` oppure dalla documentazione ufficiale del tool. I plugin del progetto sono dichiarabili nel file `build.gradle` del progetto:

```
plugins {
    id 'java'
}

group 'org.example'
version '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.9.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-api:5.9.0'
}

test {
    useJUnitPlatform()
}
```

Come si nota, il file `build.gradle` è costituito da più parti:

1. **plugins** - questo blocco specifica il tipo di applicazione, modificando il comportamento del processo di build *ad hoc*. Nel nostro caso si tratta di un progetto Java. Se volessimo specificare che è, in particolare, un'applicazione, sarà necessario aggiungere `id 'application'` alla lista dei **plugins**. In tal caso la lista dei task a disposizione sarà estesa con un comando **run** che manda in esecuzione l'applicazione. Combinazioni più o meno complesse di plugin permettono di combinare task piuttosto utili: si pensi, ad esempio, ad una pipeline Jenkins

in cui dapprima viene lanciato il task `./gradlew test` e poi, se tutte le test suite hanno successo, si lancia un processo di deployment: *automated delivery*

2. **group** e **version** - corrispondono al concetto di *coordinate* in Maven. Ad esempio considerando `org.example:my-project:1.0-SNAPSHOT`, allora `org.example` è il **group**, `my-project` è l'**artifact** e `1.0-SNAPSHOT` è la **version**
3. **dependencies** - Gradle mette a disposizione un sistema di *dependency management* dal momento che i progetti software (ben fatti) non reinventano mai la ruota: nella maggior parte dei casi, infatti, un software ci si affida a componenti *riutilizzabili* per rendere il sistema quanto più modulare e riusabile possibile – ricordando che la riusabilità è una delle qualità interne di un software. La Figura 2.1 illustra come funziona il sistema di dependency management di Gradle: come si può notare, il file `build.gradle` sfrutta una directory chiamata Gradle Cache per memorizzare e accedere ad artefatti che aveva precedentemente scaricato da repository remoti per mezzo di una rete internet. Similmente al protocollo DNS, sfrutta dei nomi *shorthand* – come ad esempio `google()` o `mavenCentral()` – per evitare di specificare l'intero URL. La configurazione `testImplementation` specifica che la dipendenza viene usata per compilare/implementare il codice di test; la configurazione `testRuntimeOnly`, invece, specifica che la dipendenza viene usata per eseguire la batteria di test. La Figura 2.2 illustra quanto detto.
4. **test** - il blocco specifica informazioni per il testing del software. Nel mio caso, `useJUnitPlatform()` è una configurazione che stabilisce che verrà utilizzato il framework JUnit.

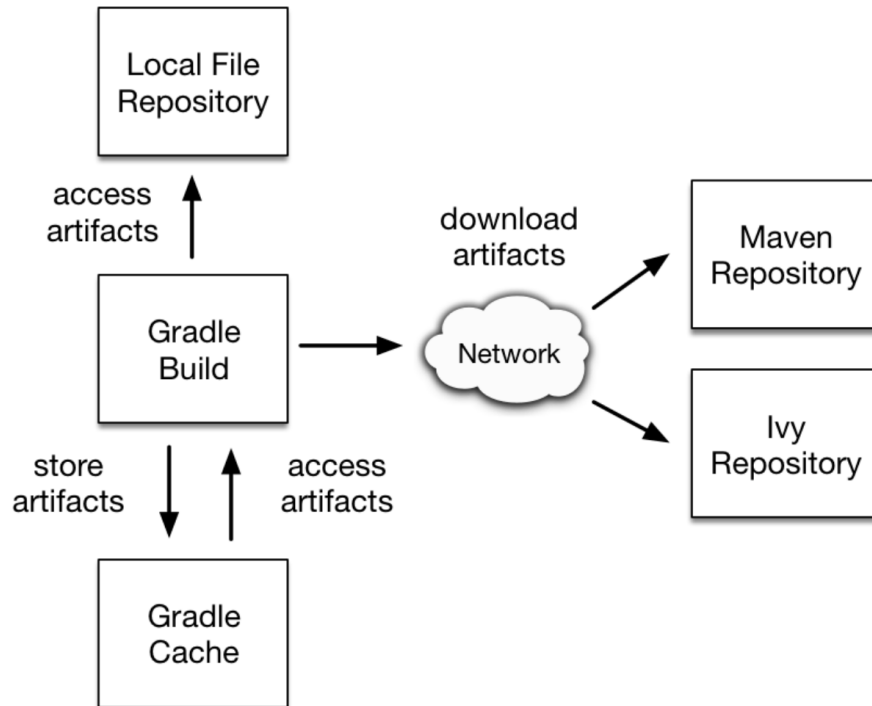
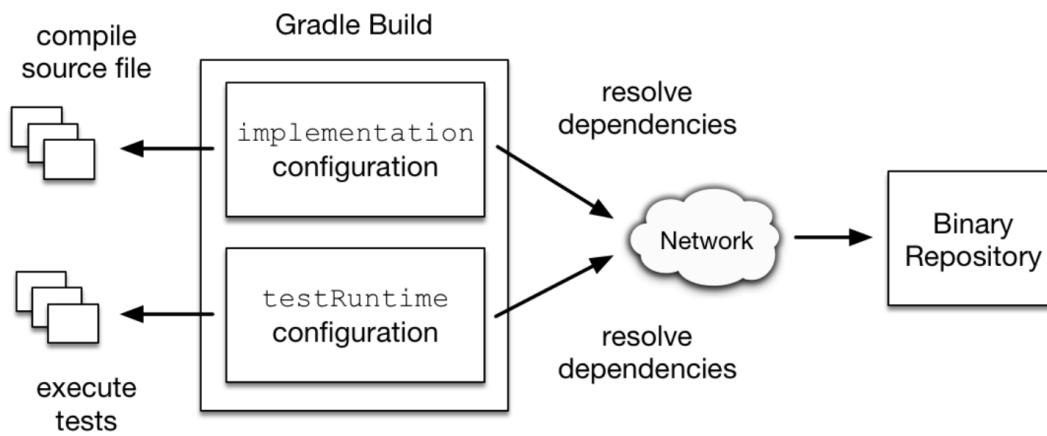


Figura 2.1: Il dependency management di Gradle

Figura 2.2: Configurare le dipenze, utilizzando ad esempio `implementation` o `testRuntime`, permette di specificarne il comportamento

Le dipendenze che specifichiamo nel file `build.gradle`, come visto, vengono scaricate in una directory chiamata Gradle Cache. Di default, la posizione sul disco di questa directory è al percorso `~/.gradle`, ma è possibile specificare un percorso alternativo utilizzando la proprietà `org.gradle.caching.local.directory` nel file `gradle.properties` del progetto. La directory in questione ha permessi `drwx-----@` il che significa che il proprietario ha diritto di lettura, scrittura ed esecuzione sui suoi file (`rw`) e tutti gli altri (gruppo e altri utenti) non hanno alcun diritto (`---` e `---`). Il carattere `@` stabilisce che ci sono estensioni di permessi (chiamate anche "ACL") attive su questa cartella. I file JAR scaricati dalle dipendenze di Gradle vengono salvati in una sottodirectory chiamata `jars` all'interno del repository. La posizione esatta del repository dipende dalla configurazione del progetto e dalla piattaforma su cui si sta eseguendo Gradle: di default, Gradle utilizza un repository locale per il download delle dipendenze. La posizione di questo repository dipende dalla piattaforma dove per Linux e MacOS è `~/.gradle/caches/modules-2/files-2.1`. La sottodirectory `jars` contiene le cartelle con i nomi delle dipendenze. Si può verificare la posizione esatta del repository locale aprendo il file `gradle.properties` del progetto e cercando la proprietà `org.gradle.caching.local.directory`. La cartella specificata in questa proprietà sarà la radice del repository locale. Si noti che Gradle *non* salva i file JAR a livello del singolo progetto: conservare i file JAR in una directory dedicata per ogni progetto potrebbe causare problemi di spazio disco a causa della duplicazione dei file JAR utilizzati da più progetti. Inoltre, potrebbe causare problemi di gestione dei file, ad esempio, quando si cerca di eliminare un progetto. Invece, utilizzando una directory cache unica per tutti i progetti, Gradle può condividere i file JAR tra i progetti, il che significa che non ci sono duplicazioni e il processo di compilazione è più veloce. Inoltre, se un progetto non è più necessario, è sufficiente eliminare solo la directory del progetto, senza doversi preoccupare di eliminare i file JAR condivisi. Infine, la posizione della cache può essere configurata utilizzando la proprietà `org.gradle.caching.local.directory` nel file `gradle.properties` del progetto, in questo modo si può specificare una cartella di cache personalizzata per il proprio progetto, così da non avere problemi di spazio disco e di gestione dei file. Si noti che `~` in Linux è la shorthand per il percorso home dell'utente corrente, nel mio caso `/Users/alessandroquirile`.

```
% ls ~/.gradle/caches/modules-2/files-2.1
```

```
com.gradle
org.jetbrains
org.opentest4j
org.apiguardian
org.junit.jupiter
org.hamcrest
org.junit.platform
```


Capitolo 3

The Gilded Rose

The Gilded Rose è una locanda che usa un'app per gestire i seguenti prodotti:

1. **Aged Brie** (brie invecchiato)
2. **Sulfuras** (mazza incantata)
3. **Backstage passes** (ticket per il backstage di un concerto)
4. **Oggetti Regular**

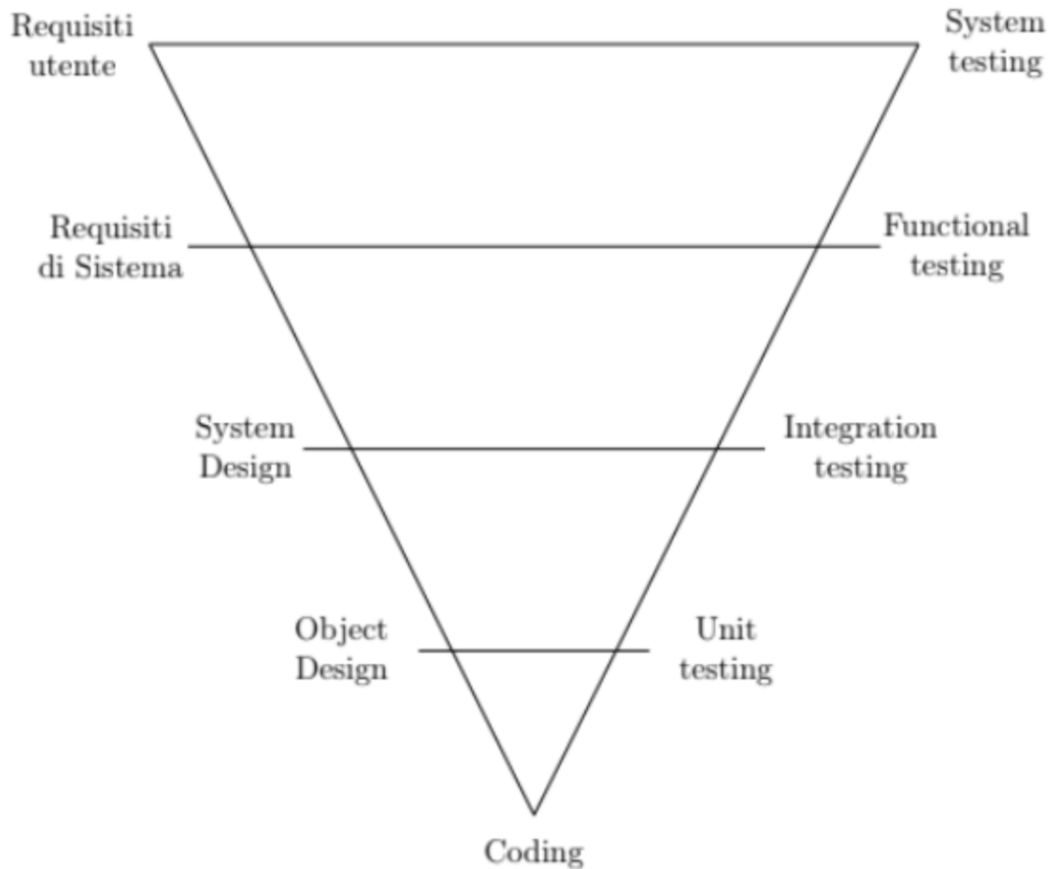
L'obiettivo è dapprima rendere *clean* il codice, successivamente evolvere il sistema aggiungendo nuovi tipi di oggetti chiamati **Conjured** (maledetti).

Il problema è che il codice originario è praticamente illegibile (*spaghetti code*), i requisiti sono intenzionalmente misleading e mancano dei test di unità che possano aiutarci nell'impresa di rifattorizzare il sistema. Prima di evolvere il sistema aggiungendo nuove funzionalità è necessario dapprima fare un buon refactoring scrivendo una test suite opportuna che garantisca la correttezza del sistema anche a seguito delle modifiche:

"Before you start refactoring, check that you have a solid suite of self-checking tests"

– Martin Fowler

Come mostrato nella seguente figura, è importante parallelizzare la fase costruttiva con quella distruttiva in primis per ragioni di tempo (sarebbe impensabile spendere sei mesi, ad esempio, per la fase costruttiva ed altri sei per la fase distruttiva) ma anche perché questo ci consente di accorgerci di errori il prima possibile (*early faults detection*).



Una test suite, infatti, dovrebbe essere tanto importante quanto il codice di produzione ed essere *molto* clean facendo sì che sia altamente *leggibile*:

*"What makes a clean test? Readability, readability, readability.
What makes tests readable? Clarity, simplicity, density of expression"*
- Robert Martin

E di qualità:

"Tests must be kept to the same level of code quality as the production code"
- Robert Martin

Per costruire le test suite dobbiamo rifarci ai requisiti funzionali del sistema scritti nel file `requirements.txt` e che ho riportato in seguito:

- Ciascun `Item` è caratterizzato da tre attributi:
 1. `name` - il nome del prodotto
 2. `sellIn` - la data entro cui il prodotto andrebbe venduto (una sorta di scadenza espressa in giorni rimanenti)
 3. `quality` - la qualità del prodotto
- Alla fine di ogni giornata un oste lancia l'app che decrementa i valori degli attributi `sellIn` e `quality`. In particolare:
 1. Una volta che è passata la vendita, ovvero quanto `sellIn < 0`, un oggetto degrada due volte più velocemente
 2. La qualità di un prodotto non è mai negativa né superiore a 50
 3. `Aged Brie` più invecchia più acquista valore
 4. `Sulfuras` è un prodotto leggendario: non degrada e non va venduto. Inoltre la sua qualità è fissa ad 80. Nulla vieta in futuro, comunque, che possa essere venduto.
 5. `Backstage passes`, come `Aged Brie`, acquista valore col passare del tempo: la sua qualità aumenta di 2 quando mancano al più 10 giorni al concerto, mentre aumenta di 3 quando ne mancano al più 5, ma la qualità diventa 0 subito dopo il concerto
 6. Il sistema va esteso aggiungendo prodotti `Conjured` che degradano due volte più velocemente dei più generici prodotti `Regular`

Le uniche classi che ci vengono consegnate dalla repository originale sono una classe che rappresenta la logica di business del sistema, chiamata `GildedRose`, con un unico metodo chiamato `updateQuality` ed una classe `Item` con le proprietà già descritte. È richiesto di *non modificare* in nessun caso il file `Item` poiché appartiene ad un Goblin che "non crede alla condivisione di codice".

3.1 Refactoring

Per prima cosa bisogna fare *program comprehension* per capire cosa fa il metodo e *come* lo fa. È una fase piuttosto delicata e dispendiosa di energie:

"Programmers have become part historians, part detectives and part clairvoyants"

- T.A. Corbi

3.1.1 Program comprehension

Il metodo da rifattorizzare si chiama `updateQuality()` ed è definito nella classe `GildedRose`. Dopo una fase di *reverse engineering* si nota che questa classe contiene unicamente 0 o più riferimenti ad istanze di `Item` le cui proprietà vengono aggiornate in base alla seguente logica:

```
public void updateQuality() {
    for (int i = 0; i < items.length; i++) {
        if (!items[i].name.equals("Aged Brie") &&
            !items[i].name.equals("Backstage passes to a TAFKAL80ETC
                concert")) {
            if (items[i].quality > 0) {
                if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
                    items[i].quality = items[i].quality - 1;
                }
            }
        } else {
            if (items[i].quality < 50) {
                items[i].quality = items[i].quality + 1;

                if (items[i].name.equals("Backstage passes to a TAFKAL80ETC
                    concert")) {
                    if (items[i].sellIn < 11) {
                        if (items[i].quality < 50) {
                            items[i].quality = items[i].quality + 1;
                        }
                    }
                }

                if (items[i].sellIn < 6) {
                    if (items[i].quality < 50) {
                        items[i].quality = items[i].quality + 1;
                    }
                }
            }
        }
    }
}
```

```
        }
    }
}

if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
    items[i].sellIn = items[i].sellIn - 1;
}

if (items[i].sellIn < 0) {
    if (!items[i].name.equals("Aged Brie")) {
        if (!items[i].name.equals("Backstage passes to a
            TAFKAL80ETC concert")) {
            if (items[i].quality > 0) {
                if (!items[i].name.equals("Sulfuras, Hand of
                    Ragnaros")) {
                    items[i].quality = items[i].quality - 1;
                }
            }
        } else {
            items[i].quality = items[i].quality - items[i].quality;
        }
    } else {
        if (items[i].quality < 50) {
            items[i].quality = items[i].quality + 1;
        }
    }
}
}
```

Listing 3.1: Metodo da rifattorizzare

3.1.2 Analisi dei bad smell

La complessità ciclomatica di McCabe, che misura il numero di cammini indipendenti in un programma attraverso il control flow graph, è pari a 19 e questo ci suggerisce che la funzione è particolarmente complessa e quindi difficile da mantenere. Inoltre tenendo conto del numero di linee di codice sorgente (SLOC) e l'indice di Halstead – che misura la complessità in base al numero di operatori e operandi intesi come keyword del linguaggio o letterali definiti dal programmatore, questi ultimi più difficili da comprendere rispetto operatori – ci si rende immediatamente conto che il metodo ha un indice di manutenibilità (MI) molto basso e occorre rimediare.

I principali bad smell che ho individuato sono i seguenti:

1. Violazione di alcuni principi SOLID:

- (a) Single Responsibility Principle: *"A module should have only one reason to change"*, mentre il metodo `updateQuality()`, oltre ad aggiornare la qualità dei prodotti, aggiorna anche l'attributo `sellIn`;
- (b) Open-Closed Principle: *"A module should be open for extension but closed for modification"*, mentre il metodo `updateQuality()` verrebbe modificato se aggiungiamo nuove feature al sistema;
- (c) Liskov-Substitution Principle: *"Subclasses should be substitutable for their base class"* questo principio non viene violato perché non ci sono gerarchie. Tuttavia è un campanello d'allarme perché le misure di CK sulla *depth of inheritance* (DIT) dipendono proprio dalla distribuzione di ereditarietà dell'intero software. Un software senza gerarchie è un software che sfrutta poco il paradigma Object-Oriented;
- (d) Interface Segregation Principle: *"A module should not be forced to depend upon interfaces they do not use"* questo principio non viene violato perché il software non usa interfacce;
- (e) Dependency Inversion Principle: *"High level modules should not depend on low level modules, both should depend on abstraction"*: questo sistema non prevede alcuna forma di astrazione.

2. *Bloater method* - `updateQuality()` è un metodo troppo lungo ed è difficile da leggere. Se non rifattorizzato, aumenta di dimensione ad ogni nuova funzionalità del sistema. Robert C. Martin nel suo libro *Clean Code* afferma che i moduli dovrebbero essere *small* e poi resi ancora più small, altamente coesi e disaccoppiati. Per risolvere questo problema, oltre ad un'attenta fase di program comprehension e una test suite adeguata (che sia leggibile!), occorre fare uso di tecniche come *method extraction* e *conditional decomposing*. In generale metodi troppo

lunghe nascondono sempre del *duplicated code*, altro bad smell. Anche il nome è misleading: `updateQuality()` non aggiorna solo la qualità di un prodotto, ma anche l'attributo `sellIn`: è più opportuno chiamarlo `updateState` oppure `updateItemProperties`.

3. *Change preventer* - le clausole `if` predicano su variabili d'istanza di oggetti `Item`: ad esempio si verifica che il nome *non* è `Aged Brie` e *non* è `Backstage Passes`, poi si verifica che la qualità sia positiva e poi viene fatto un altro controllo sul nome dell'oggetto che *non* deve essere `Sulfuras` per decrementare la qualità di 1. Una logica mostruosamente esagerata, se pensiamo che è la corrispondenza in linguaggio naturale delle sole prime 4 righe! Anziché usare gli `if-else/switch`, usiamo un fondamentale della programmazione OOP: il *polimorfismo*. Questo consente di ridurre drasticamente il numero di righe del metodo e di semplificare di moltissimo la sua logica. Costrutti `if-else/switch` dovrebbero essere usati soltanto quando la complessità ciclotomica rimane bassa (pochi percorsi indipendenti) e quando si utilizzano design pattern creazionali quali *Factory Method*. In generale costrutti `if-else/switch` danno luogo ad n alternative: se n cresce, anche la lunghezza del metodo crescerà. Un altro change preventer è dovuto al fatto che il codice non dipende da alcuna astrazione, ma direttamente da implementazioni: bisognerebbe sempre progettare per anticipare il cambiamento.
4. *Dispensable: duplicate code* - osserviamo il metodo. Quante volte l'istruzione `item[i].quality = item[i].quality + 1` viene ripetuta? E quante volte viene fatto il controllo per verificare se il nome dell'oggetto corrente non è "Backstage Passes"?
5. Nessun livello di astrazione - sarebbe buona norma leggere il codice di un metodo in maniera *top-down* seguendo la *stepdown rule*. Qui, invece, vengono mischiate istruzioni ad alto livello con istruzioni a basso livello.
6. *Formatting hell* - problemi di formattazione a livello di spaziatura che rendono difficoltosa la fase di program comprehension.
7. *No OOP* - mancano alcuni fondamenti della programmazione a oggetti quali encapsulation, inheritance e polymorphism.

I tipi di cambiamento che farò al sistema saranno di tipo perfettivo-evolutivo, aggiungendo funzionalità al sistema e applicando la *re-engineering* per migliorare la struttura del sistema e le sue prestazioni anche in termini di qualità percepita dagli altri sviluppatori.

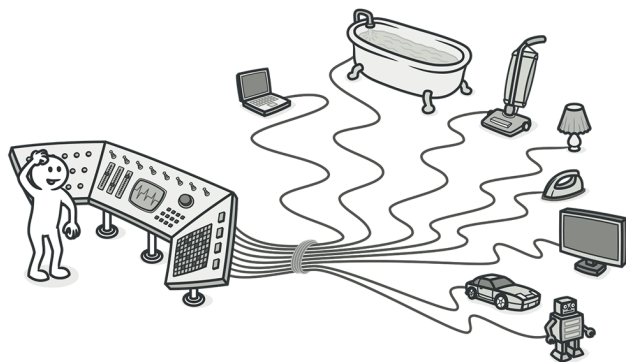


Figura 3.1: Abuso di costrutti if-else/switch

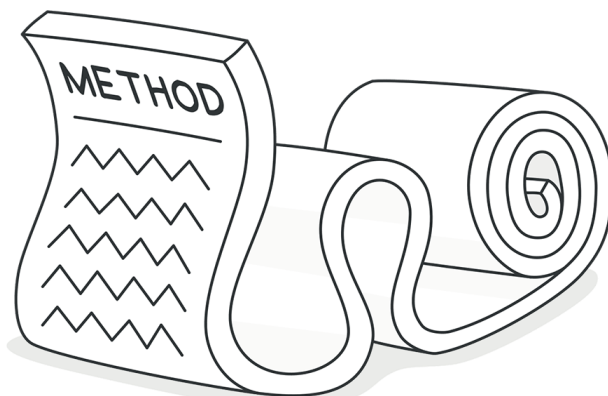


Figura 3.2: Un metodo troppo lungo



Figura 3.3: Dispensables

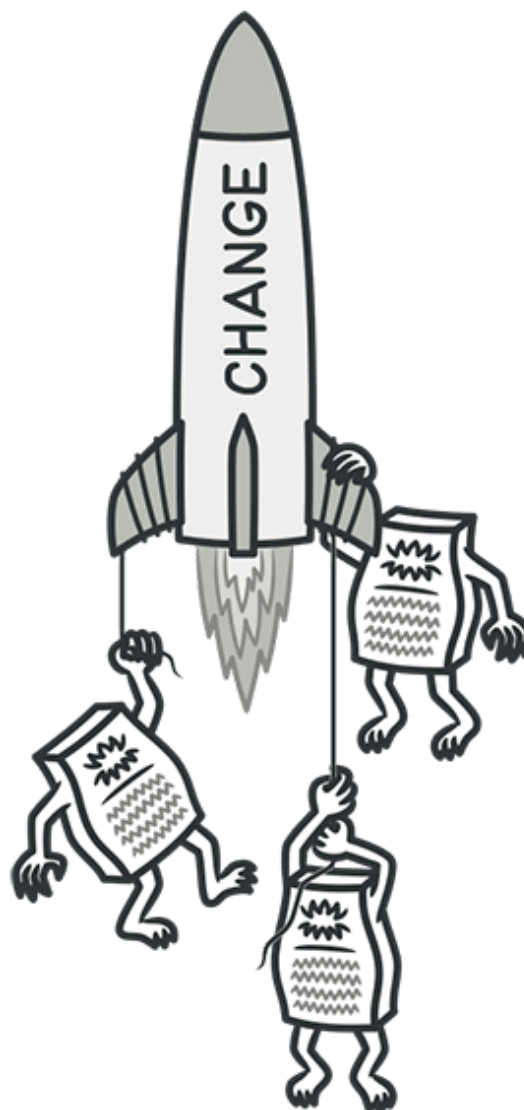
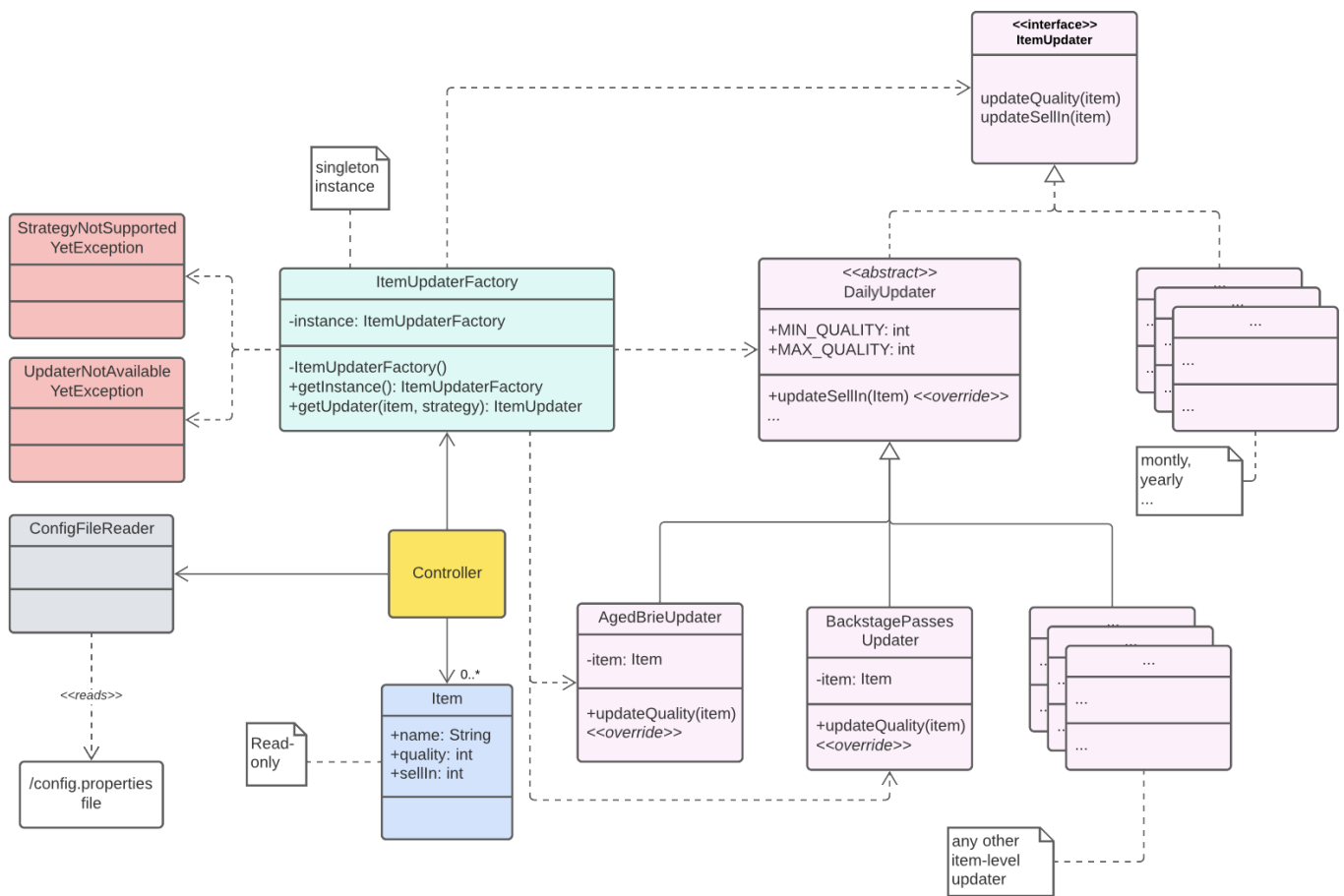


Figura 3.4: Change preventers

3.2 Risolvere i bad smell

Per risolvere i bad smell ho creato alcune classi e delle test suite che verranno descritte in questo paragrafo. Per ciascuna classe specifico testualmente qual è la sua responsabilità (unica!). In accordo alle leggi di Lehman, ha senso fare uno sforzo adesso per preservare e semplificare la struttura del sistema che evolve continuamente col passare del tempo.

Il seguente class diagram orientativo mostra, utilizzando il formalismo UML, la soluzione proposta. Il controller, che rappresenta la logica di business del sistema, avrà 0 o più riferimenti verso **Item**, classe che astrae le proprietà di un generico oggetto in dispensa e viene fornito dal collega Goblin ed è pertanto read-only.



Inoltre, attraverso un riferimento ad un'istanza singleton di **ItemUpdaterFactory**, il controller aggiornerà lo stato di ciascun item, inteso come coppia **quality** e **sellIn**, coerentemente ai requisiti

funzionali del sistema già descritti. E esso, inoltre, non è al corrente della reale natura dell'item (può essere uno qualsiasi tra Aged Brie, Backstage Passes e così via) perché si utilizzerà il design pattern State/Strategy congiuntamente al Factory Method + Singleton per rendere il comportamento dinamico istanziando l'opportuno updater per ciascun item. Si noti che lo strato di interfacce e/o classi *astratte* consente di implementare la *dependency inversion* siccome spezza la logica delle entità più in basso livello: le astrazioni non dovrebbero dipendere dai dettagli implementativi. Così facendo rendo il prodotto software più disaccoppiato possibile e seguo la filosofia del *design by contract*, "promettendo" cosa farà l'interfaccia attraverso la firma dei suoi metodi. Qualsiasi cambiamento futuro – ed il software evolve in continuazione – sarà più facile da gestire, rendendo la fase di manutenzione più semplice: bisognerebbe sempre progettare per anticipare il cambiamento, ma trovando il giusto trade-off perché aumentare di troppo il livello di astrazione del sistema potrebbe portare al problema dello *speculative generality* secondo cui il software diventa troppo complesso e quindi difficile da capire: in ingegneria del software non esiste mai la soluzione perfetta ma bisogna trovare un buon compromesso: ad esempio questo fenomeno può essere tranquillamente ignorato quando si sta progettando un framework. Dipende tutto dal contesto!

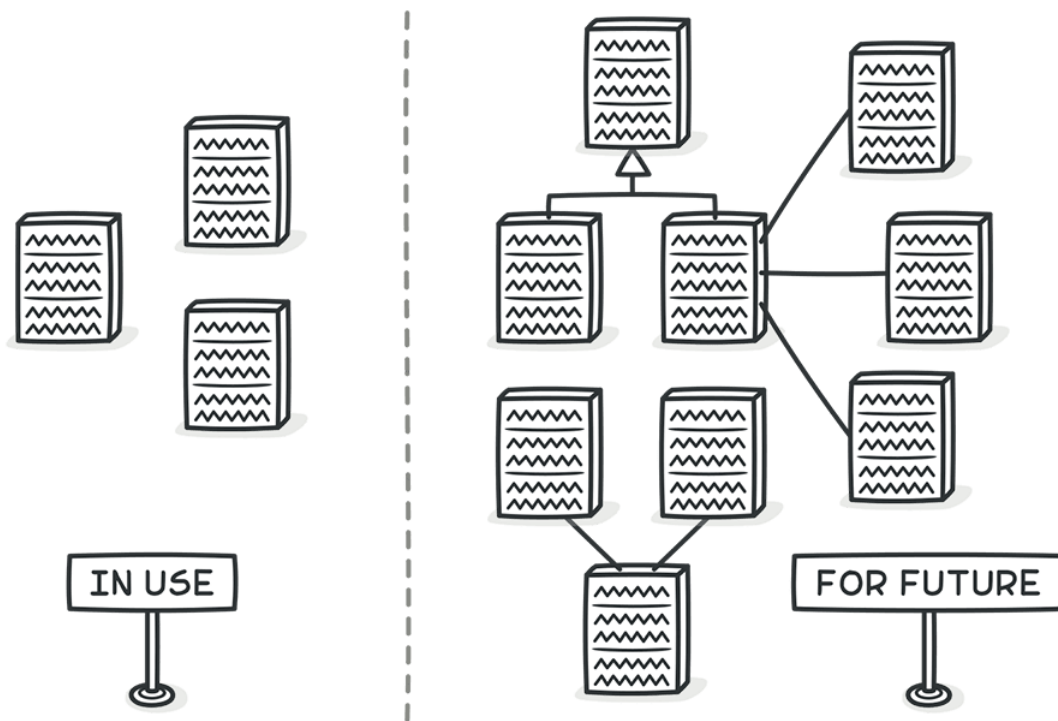


Figura 3.5: Speculative generality

Supponendo che l'*i*-esimo item da aggiornare sia Aged Brie, il controller dapprima chiederà alla classe `ConfigFileReader` di leggere il file `config.properties`¹ per ottenere la modalità di aggiornamento preferita (quotidiana, mensile, annuale...) e, successivamente, chiederà all'`ItemUpdaterFactory` di istanziare un oggetto `AgedBrieUpdater` restituendolo sotto forma di interfaccia `ItemUpdater`. Il casting da classe a interfaccia è reso possibile siccome la classe `AgedBrieUpdater` implementa `ItemUpdater`.

Il design del sistema è sufficientemente generalizzabile: se in futuro si vorrà dare la possibilità di vendere un nuovo item, ad esempio `MagicSword`, basterà scrivere una nuova classe `MagicSwordUpdater` con la sua logica a basso livello per aggiornare l'attributo `quality`. Similmente se dovesse essere necessario aggiornare lo stato degli item a cadenza non giornaliera ma, ad esempio, mensile, basterà scrivere una nuova classe astratta `MonthlyUpdater` che implementa `ItemUpdater` e decrementa l'attributo `item.sellIn` in base al numero di giorni in quel mese. Se si dovesse richiedere una strategia non ancora implementata, come ad esempio un aggiornamento mensile o annuale, sarà lanciata l'eccezione `StrategyNotSupportedYetException`; similmente se venisse richiesto un updater per un prodotto che non esiste in dispensa, verrà lanciata l'eccezione `UpdaterNotAvailableYetException` che illustreremo in dettaglio più avanti. In generale, il sistema deve essere

"Open for extension but closed for modification"
- Open/Closed Principle

Tutte le classi del class diagram di cui sopra, e non solo, saranno descritte nel dettaglio a breve, tenendo in considerazione anche l'aspetto implementativo in Java.

Infine per quanto riguarda la naming convention ho seguito quella standard di Java e quella suggerita da Robert Martin in *Clean Code*; in più se in futuro si vorrà aggiungere un nuovo updater a livello di tempo, suggerisco di seguire il pattern `<TimeLevel>Updater`, ad esempio `DailyUpdater`, `MonthlyUpdater`, `YearlyUpdater`... similmente se vorrò aggiungere un updater a livello del singolo item, suggerisco di seguire il pattern `<ItemName>Updater`, ad esempio `AgedBrieUpdater`, `MagicSwordUpdater`, `SteelShieldUpdater` e così via.

¹Si tratta di un file `key=value` che può essere usato per memorizzare alcune preferenze come, ad esempio, la modalità di aggiornamento (`update=daily`), oppure la tecnologia di connessione al database (`storage=mongodb`), oppure protocolli di connessione (`protocol=https`) e così via

3.2.1 Il package exceptions

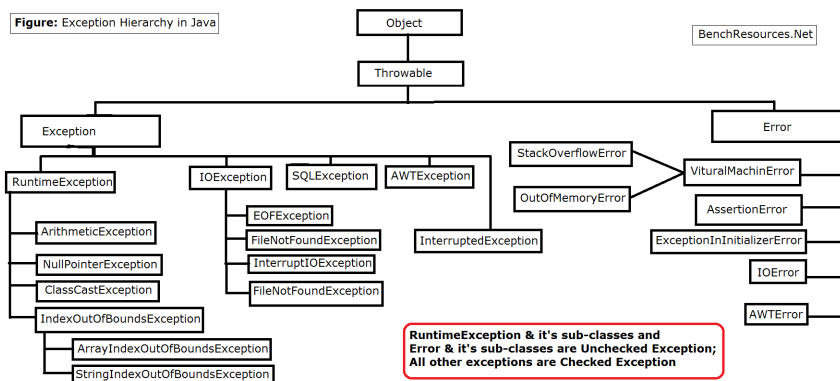
Questo package contiene delle eccezioni che estendono `IllegalArgumentException` e che potrebbero essere lanciate durante l'esecuzione dell'applicazione.

L'eccezione `StrategyNotSupportedYetException`, come visto, viene lanciata quando viene richiesta una strategia di aggiornamento che non è ancora disponibile: il sistema funziona per aggiornare lo stato di ciascun item quotidianamente e quindi se venisse richiesto di aggiornarne lo stato mensilmente, verrebbe lanciata la seguente eccezione:

```
public class StrategyNotSupportedYetException extends IllegalArgumentException {
    public StrategyNotSupportedYetException(String strategy) {
        super(strategy + " is not supported yet");
    }
}
```

L'eccezione `UpdaterNotAvailableYetException`, invece, viene lanciata quando viene richiesto un updater che non esiste ancora nel sistema, quindi per un prodotto che non è presente in dispensa.

```
public class UpdaterNotAvailableYetException extends IllegalArgumentException {
    public UpdaterNotAvailableYetException(Item item) {
        super("An updater is not available yet for " + item.name);
    }
}
```



3.2.2 Il package factories

La classe `ItemUpdaterFactory` ha la responsabilità di istanziare il giusto updater in base all'item e alla `strategy` passati in input, implementando il design pattern *Factory Method* e *Singleton*.

```
public class ItemUpdaterFactory {
    private static ItemUpdaterFactory instance;

    private ItemUpdaterFactory() {
        instance = null;
    }

    public static synchronized ItemUpdaterFactory getInstance() {
        if (instance == null)
            instance = new ItemUpdaterFactory();
        return instance;
    }

    public ItemUpdater getUpdater(Item item, String strategy) {
        if (!isDaily(strategy))
            throw new StrategyNotSupportedYetException(strategy);
        return getDailyUpdater(item);
    }

    private boolean isDaily(String strategy) {
        return strategy.equals("daily");
    }

    private static DailyUpdater getDailyUpdater(Item item) {
        if (isAgedBrie(item))
            return new AgedBrieUpdater(item);
        else if (isBackstagePasses(item))
            return new BackstagePassesUpdater(item);
        else if (isConjured(item))
            return new ConjuredItemUpdater(item);
        else if (isRegular(item))
            return new RegularItemUpdater(item);
        else if (isSulfuras(item))
            return new SulfurasUpdater(item);
        throw new UpdaterNotAvailableYetException(item);
    }
}
```

```
}
```

Ho usato il modificatore `synchronized` per gestire, eventualmente, accessi concorrenti alla risorsa garantendo che un solo thread alla volta, se l'applicazione migrasse in futuro verso una soluzione multi-thread, possa eseguire la funzione `getInstance()` per istanziare tale classe. Si noti, inoltre, che una catena di `if-else/switch` *non* è considerata un bad smell in una classe che implementa il design pattern Factory Method.

3.2.3 Il package test.factories

Questo package contiene la test suite per le classi Factory, nel nostro caso solo `ItemUpdaterFactory`. Per motivi di spazio in questo documento ho omesso alcuni attributi (`// ...`). Si noti che, poiché sto usando `jUnit 5.x`, la sintassi corretta per asserire un'eccezione è `assertThrows(.,.)` siccome l'attributo `expected=<nomeEccezione.class>` dell'annotazione `@Test` è esclusivo di `jUnit 4.x`. Si ricordi, infine, che la sintassi `() -> function(.)` è una *lambda expression*:

```
public class ItemUpdaterFactoryTest {
    private final String STRATEGY = "daily";
    // ...

    @BeforeEach
    void init() {
        factory = ItemUpdaterFactory.getInstance();
    }

    @Test
    void getProperUpdaterForDailyUpdate() {
        item = new Item(agedBrie, 0, 0);
        updater = factory.getUpdater(item, STRATEGY);
        assert (updater instanceof DailyUpdater);
    }

    @Test
    void shouldThrowTechnologyNotSupportedYetExceptionWhenStrategyIsNot -
        Daily() {
        item = new Item(agedBrie, 0, 0);
        assertThrows(
            StrategyNotSupportedYetException.class,
            () -> factory.getUpdater(item, "Any other strategy")
        );
    }

    @Test
    void shouldThrowIllegalArgumentExceptionWhenItemHasNoUpdater() {
        item = new Item("Any other item", 0, 0);
        assertThrows(
            UpdaterNotAvailableYetException.class,
            () -> factory.getUpdater(item, STRATEGY)
        );
    }
}
```

```
    );  
}  
  
@Test  
void getProperUpdaterForAgedBrie() {  
    item = new Item(agedBrie, 0, 0);  
    updater = factory.getUpdater(item, STRATEGY);  
    assert (updater instanceof AgedBrieUpdater);  
}  
  
@Test  
void getProperUpdaterForBackstagePasses() {  
    item = new Item(backstagePasses, 0, 0);  
    updater = factory.getUpdater(item, STRATEGY);  
    assert (updater instanceof BackstagePassesUpdater);  
}  
  
@Test  
void getProperUpdaterForConjuredItem() {  
    item = new Item(conjured, 0, 0);  
    updater = factory.getUpdater(item, STRATEGY);  
    assert (updater instanceof ConjuredItemUpdater);  
}  
  
@Test  
void getProperUpdaterForRegularItem() {  
    item = new Item(regular, 0, 0);  
    updater = factory.getUpdater(item, STRATEGY);  
    assert (updater instanceof RegularItemUpdater);  
}  
  
@Test  
void getProperUpdaterForSulfuras() {  
    item = new Item(sulfuras, 0, 80);  
    updater = factory.getUpdater(item, STRATEGY);  
    assert (updater instanceof SulfurasUpdater);  
}  
}
```


3.2.4 Il package interfaces

Questo package contiene interfacce e/o classi astratte che consentono di implementare il design pattern State/Strategy.

L'interfaccia `ItemUpdater` ha *la* responsabilità di fornire la firma di metodi per aggiornare lo stato di un `item`.

```
public interface ItemUpdater {  
    void updateQuality(Item item);  
    void updateSellIn(Item item);  
}
```

La classe astratta `DailyUpdater` ha *la* responsabilità di implementare parzialmente l'interfaccia descritta sopra, sovrascrivendo il metodo `updateSellIn(item)` – siccome la classe si riferisce a quella famiglia di `updater` che possono essere usati quotidianamente (perciò nel nome c'è il suffisso `Daily`).

```
public abstract class DailyUpdater implements ItemUpdater {  
    public static final int MIN_QUALITY = 0;  
    public static int MAX_QUALITY = 50;  
  
    public abstract void updateQuality(Item item);  
  
    @Override  
    public void updateSellIn(Item item) {  
        item.sellIn--;  
    }  
  
    public boolean isValid(int quality) {  
        return quality > MIN_QUALITY && quality < MAX_QUALITY;  
    }  
  
    public boolean hasExpired(Item item) {  
        return item.sellIn < 0;  
    }  
  
    public int getUpdateRate(Item item) {  
        return hasExpired(item) ? 2 : 1;  
    }  
}
```

}

3.2.5 Il package test.interfaces

Questo package contiene la test suite per le interfacce, nel nostro caso per la sola classe astratta `DailyUpdater`. Anche in questo caso ho omesso alcuni attributi per brevità.

```
class DailyUpdaterTest {
    private static final String DAILY = "daily";
    // ...

    @BeforeEach
    void init() {
        factory = ItemUpdaterFactory.getInstance();
    }

    @Test
    void sellInShouldDecreaseBy1EveryDay() {
        item = new Item(brie, 10, 30);
        updater = (DailyUpdater) factory.getUpdater(item, DAILY);
        updater.updateSellIn(item);
        assertEquals(9, item.sellIn);
    }

    @Test
    void itemShouldBeValidWhenQualityIsBetween0And50() {
        item = new Item(brie, 10, 30);
        updater = (DailyUpdater) factory.getUpdater(item, DAILY);
        assertTrue(updater.isValid(item.quality));
    }

    @Test
    void itemShouldNotBeValidWhenQualityIsLessThan0() {
        item = new Item(brie, 10, -1);
        updater = (DailyUpdater) factory.getUpdater(item, DAILY);
        assertFalse(updater.isValid(item.quality));
    }

    @Test
    void itemShouldNotBeValidWhenQualityIsMoreThan50() {
        item = new Item(brie, 10, 51);
        updater = (DailyUpdater) factory.getUpdater(item, DAILY);
        assertFalse(updater.isValid(item.quality));
    }
}
```

```
}

@Test
void itemShouldExpireWhenSellInIsNegative() {
    item = new Item(brie, -1, 30);
    updater = (DailyUpdater) factory.getUpdater(item, DAILY);
    assertTrue(updater.hasExpired(item));
}

@Test
void itemShouldNotExpireWhenSellInIsZero() {
    item = new Item(brie, 10, 0);
    updater = (DailyUpdater) factory.getUpdater(item, DAILY);
    assertFalse(updater.hasExpired(item));
}

@Test
void itemShouldNotExpireWhenSellInIsPositive() {
    item = new Item(brie, 10, 1);
    updater = (DailyUpdater) factory.getUpdater(item, DAILY);
    assertFalse(updater.hasExpired(item));
}

@Test
void updateRateShouldBe2IfExpired() {
    item = new Item(brie, -1, 30);
    updater = (DailyUpdater) factory.getUpdater(item, DAILY);
    assertEquals(2, updater.getUpdateRate(item));
}

@Test
void updateRateShouldBe1IfNotExpired() {
    item = new Item(brie, 1, 1);
    updater = (DailyUpdater) factory.getUpdater(item, DAILY);
    assertEquals(1, updater.getUpdateRate(item));
}
}
```

3.2.6 Il package implementations

Questo package contiene le classi che implementano le interfacce e/o le classi astratte illustrate nel paragrafo precedente a seconda del tipo specifico di `item`.

La classe `AgedBrieUpdater` ha *la* responsabilità di aggiornare lo stato di un Aged Brie.

```
public class AgedBrieUpdater extends DailyUpdater {
    public Item item;

    public AgedBrieUpdater(Item item) {
        this.item = item;
    }

    @Override
    public void updateQuality(Item item) {
        if (isValid(item.quality)) {
            int improvementRate = getUpdateRate(item);
            item.quality += improvementRate;
        }
    }
}
```

La classe `BackstagePassesUpdater` ha *la* responsabilità di aggiornare lo stato di un `Backstage Passes`.

```
public class BackstagePassesUpdater extends DailyUpdater {
    public Item item;

    public BackstagePassesUpdater(Item item) {
        this.item = item;
    }

    @Override
    public void updateQuality(Item item) {
        if (isValid(item.quality)) {
            if (hasExpired(item))
                item.quality = 0;
            else
                item.quality += getUpdateRate(item);
        }
    }

    @Override
    public int getUpdateRate(Item item) {
        int daysToConcert = item.sellIn;
        if (daysToConcert <= 5)
            return 3;
        else if (daysToConcert <= 10)
            return 2;
        else
            return 1;
    }
}
```

La classe `RegularItemUpdater` ha *la* responsabilità di aggiornare lo stato di un item `Regular`.

```
public class RegularItemUpdater extends DailyUpdater {
    public Item item;

    public RegularItemUpdater(Item item) {
        this.item = item;
    }

    @Override
    public void updateQuality(Item item) {
        if (isValid(item.quality)) {
            int decayRate = getUpdateRate(item);
            item.quality -= decayRate;
        }
    }
}
```

La classe `SulfurasUpdater` ha *la* responsabilità di aggiornare lo stato di un item `Sulfuras`. Nota che ho usato `// do nothing` come *explanation of intent* per ricordare agli altri sviluppatori i requisiti funzionali.

```
public class SulfurasUpdater extends DailyUpdater {
    private static final int MAX_QUALITY = 80;
    public Item item;

    public SulfurasUpdater(Item item) {
        this.item = item;
    }

    @Override
    public void updateQuality(Item item) {
        item.quality = MAX_QUALITY;
    }

    @Override
    public void updateSellIn(Item item) {
        // do nothing
    }
}
```


3.2.7 Il package test.implementations

Questo package contiene le test suite delle classi implementations.

```
public class AgedBrieUpdaterTest {
    private final String brie = InStock.AGED_BRIE.getFullName();
    private Item item;
    private AgedBrieUpdater updater;

    @Test
    void qualityShouldIncreaseBy1WhenSellInIsPositive() {
        item = new Item(brie, 30, 40);
        updater = new AgedBrieUpdater(item);
        updater.updateQuality(item);
        assertEquals(41, updater.item.quality);
    }

    @Test
    void qualityShouldIncreaseBy1WhenSellInIsZero() {
        item = new Item(brie, 0, 40);
        updater = new AgedBrieUpdater(item);
        updater.updateQuality(item);
        assertEquals(41, updater.item.quality);
    }

    @Test
    void qualityShouldIncreaseBy2WhenSellIsNegative() {
        item = new Item(brie, -1, 40);
        updater = new AgedBrieUpdater(item);
        updater.updateQuality(item);
        assertEquals(42, updater.item.quality);
    }

    @Test
    void qualityIsAtLeast0() {
        item = new Item(brie, 30, 0);
        updater = new AgedBrieUpdater(item);
        updater.updateQuality(item);
        assertEquals(0, updater.item.quality);
    }
}
```

```
@Test
void qualityIsAtMost50() {
    item = new Item(brie, 30, 50);
    updater = new AgedBrieUpdater(item);
    updater.updateQuality(item);
    assertEquals(50, updater.item.quality);
}
}
```

```
public class BackstagePassesUpdaterTest {
    private final String pass = InStock.BACKSTAGE_PASSES.getFullName();
    private Item item;
    private BackstagePassesUpdater updater;

    @Test
    void qualityShouldIncreaseBy1WhenSellInIsMoreThan10() {
        item = new Item(pass, 11, 30);
        updater = new BackstagePassesUpdater(item);
        updater.updateQuality(item);
        assertEquals(31, updater.item.quality);
    }

    @Test
    void qualityShouldIncreaseBy2WhenSellInIs10_orLess() {
        item = new Item(pass, 10, 30);
        updater = new BackstagePassesUpdater(item);
        updater.updateQuality(item);
        assertEquals(32, updater.item.quality);
    }

    @Test
    void qualityShouldIncreaseBy3WhenSellInIs5_orLess() {
        item = new Item(pass, 5, 30);
        updater = new BackstagePassesUpdater(item);
        updater.updateQuality(item);
        assertEquals(33, updater.item.quality);
    }

    @Test
    void qualityIsAtLeast0() {
        item = new Item(pass, 30, 0);
        updater = new BackstagePassesUpdater(item);
        updater.updateQuality(item);
        assertEquals(0, updater.item.quality);
    }

    @Test
    void qualityIsAtMost50() {
        item = new Item(pass, 30, 50);
```

```
        updater = new BackstagePassesUpdater(item);
        updater.updateQuality(item);
        assertEquals(50, updater.item.quality);
    }
}
```

```
public class RegularItemUpdaterTest {
    private final String regular = InStock.REGULAR.getFullName();
    private Item item;
    private RegularItemUpdater updater;

    @Test
    void qualityShouldDecreaseBy1WhenSellInIsPositive() {
        item = new Item(regular, 30, 40);
        updater = new RegularItemUpdater(item);
        updater.updateQuality(item);
        assertEquals(39, updater.item.quality);
    }

    @Test
    void qualityShouldDecreaseBy1WhenSellInIsZero() {
        item = new Item(regular, 0, 40);
        updater = new RegularItemUpdater(item);
        updater.updateQuality(item);
        assertEquals(39, updater.item.quality);
    }

    @Test
    void qualityShouldDecreaseBy2WhenSellInIsNegative() {
        item = new Item(regular, -1, 40);
        updater = new RegularItemUpdater(item);
        updater.updateQuality(item);
        assertEquals(38, updater.item.quality);
    }

    @Test
    void qualityIsAtLeast0() {
        item = new Item(regular, 30, 0);
        updater = new RegularItemUpdater(item);
        updater.updateQuality(item);
        assertEquals(0, updater.item.quality);
    }

    @Test
```

```
void qualityIsAtMost50() {  
    item = new Item(regular, 30, 50);  
    updater = new RegularItemUpdater(item);  
    updater.updateQuality(item);  
    assertEquals(50, updater.item.quality);  
}  
}
```

```
public class SulfurasUpdaterTest {
    private final String sulfuras = InStock.SULFURAS.getFullName();
    private Item item;
    private SulfurasUpdater updater;

    @BeforeEach
    void init() {
        item = new Item(sulfuras, 30, 80);
        updater = new SulfurasUpdater(item);
    }

    @Test
    void qualityShouldBe80() {
        updater.updateQuality(item);
        assertEquals(80, updater.item.quality);
    }

    @Test
    void sellInShouldNotDecrease() {
        updater.updateQuality(item);
        assertEquals(30, updater.item.sellIn);
    }
}
```

3.2.8 Il package models

Questo package contiene classi POJO²/Model.

La classe `Item` ha *la* responsabilità di modellare gli elementi in dispensa. Viene fornita dal Goblin ed è pertanto *read-only*. Si noti che gli attributi sono `public` e non vengono forniti metodi `getter` e `setter`: manca l'incapsulazione dei dati.

```
/**
 * This is a read-only file.
 * Do not alter the Item class or Item's property as those
 * belong to the goblin in the corner who will insta-rage and
 * one-shot you as he doesn't believe in shared code ownership
 *
 * @author Goblin
 */
public class Item {
    public String name;
    public int sellIn;
    public int quality;

    public Item(String name, int sellIn, int quality) {
        this.name = name;
        this.sellIn = sellIn;
        this.quality = quality;
    }

    @Override
    public String toString() {
        return this.name + ", " + this.sellIn + ", " + this.quality;
    }
}
```

²https://it.wikipedia.org/wiki/Plain_Old_Java_Object

3.2.9 Il package utils

Questo package contiene classi di generica utilità.

La classe `ConfigFileReader` ha la responsabilità di leggere il file di configurazione `config.properties` che contiene coppie `key=value` per ricavare una proprietà, ovvero una `value`, a partire da una `key`.

```
public class ConfigFileReader {
    public static String getProperty(String key) {
        Properties properties = new Properties();
        InputStream stream = ConfigFileReader.class.getResourceAsStream
            ("/config.properties");
        properties.load(stream);
        return properties.getProperty(key);
    }
}
```

La enum `InStock` ha la responsabilità di enumerare gli item in dispensa. Alcuni di essi, infatti, potrebbero avere un nome molto lungo (vd. "Sulfuras, Hand of Ragnaros") e questa stringa sarà il `fullName` dell'enum `SULFURAS`.

```
public enum InStock {
    AGED_BRIE("Aged Brie"),
    BACKSTAGE_PASSES("Backstage passes to a TAFKAL80ETC concert"),
    CONJURED("Conjured"),
    REGULAR("Regular"),
    SULFURAS("Sulfuras, Hand of Ragnaros");

    private final String fullName;

    InStock(String fullName) {
        this.fullName = fullName;
    }

    public String getFullName() {
        return fullName;
    }
}
```

La classe `ItemTypeChecker` ha *la* responsabilità di verificare se un `item` passato in input è uno tra quelli in dispensa.

```
public abstract class ItemTypeChecker {
    public static boolean isSulfuras(Item item) {
        return InStock.SULFURAS.getFullName().equals(item.name);
    }

    public static boolean isRegular(Item item) {
        return InStock.REGULAR.getFullName().equals(item.name);
    }

    public static boolean isConjured(Item item) {
        return InStock.CONJURED.getFullName().equals(item.name);
    }

    public static boolean isAgedBrie(Item item) {
        return InStock.AGED_BRIE.getFullName().equals(item.name);
    }

    public static boolean isBackstagePasses(Item item) {
        return InStock.BACKSTAGE_PASSES.getFullName().equals(item.name);
    }
}
```

3.3 Risultato finale

Il metodo `GildedRose.updateState()` è stato così rifattorizzato:

```
public void updateState(List<Item> items) {
    factory = ItemUpdaterFactory.getInstance();
    strategy = ConfigFileReader.getProperty("update_strategy");
    for (Item item : items)
        updateState(item);
}

private void updateState(Item item) {
    updater = factory.getUpdater(item, strategy);
    updater.updateQuality(item);
    updater.updateSellIn(item);
}
```

Il metodo ricava l'istanza di `ItemUpdaterFactory` e la `strategy` e aggiorna lo stato di ciascun `item` nella lista `items` passata in ingresso.

Si noti che ho sfruttato l'*overloading* di Java e ho aggiunto come parametro una lista di `Item`, così da rendere questo metodo riusabile e *self-contained*. Inoltre ho utilizzato i *generics*, la cui sintassi è `<.>`, per definire tipi di dati omogenei per una collezione di oggetti; nel mio caso `List<Item> items` definisce una lista di oggetti di tipo `Item`, così da avere maggiore sicurezza del codice: gli errori di tipo vengono individuati in fase di compilazione, il che rende il sistema più robusto e meno suscettibile a bug, più leggibile dal momento che i generics consentono di specificare il tipo di dati utilizzato all'interno di una classe o di un metodo, e più flessibile siccome consentono di creare classi e metodi generici che possono essere utilizzati con diversi tipi di dati, il che aumenta la riutilizzabilità del codice.

Si confronti il risultato ottenuto con il metodo prima del refactoring in Figura 3.1.

3.4 Evolution

Ci viene chiesto di dare la possibilità alla locanda di poter vendere anche prodotti **Conjured**, intesi come particolari oggetti **Regular** che degradano due volte più velocemente rispetto ad essi.

3.4.1 Modellare oggetti Conjured

Dopo aver eseguito il locale i test per assicurarci che il refactoring non abbia violato i requisiti funzionali del sistema, possiamo procedere con l'evoluzione dello stesso.

La classe `ConjuredItemUpdater` ha la responsabilità di aggiornare la qualità dei **Conjured**, sovrascrivendo opportunamente il metodo `getUpdateRate(item)`.

```
public class ConjuredItemUpdater extends RegularItemUpdater {

    public ConjuredItemUpdater(Item item) {
        super(item);
    }

    @Override
    public int getUpdateRate(Item item) {
        return 2 * super.getUpdateRate(item);
    }
}
```

E, al solito, una test suite:

```
public class ConjuredItemUpdaterTest {
    private final String conjured = InStock.CONJURED.getFullName();
    private Item item;
    private ConjuredItemUpdater updater;

    @Test
    void qualityShouldDecreaseBy2WhenSellInIsPositive() {
        item = new Item(conjured, 30, 40);
        updater = new ConjuredItemUpdater(item);
        updater.updateQuality(item);
        assertEquals(38, updater.item.quality);
    }
}
```

```
@Test
void qualityShouldDecreaseBy2WhenSellInIsZero() {
    item = new Item(conjured, 0, 40);
    updater = new ConjuredItemUpdater(item);
    updater.updateQuality(item);
    assertEquals(38, updater.item.quality);
}

@Test
void qualityShouldDecreaseBy4WhenSellInIsNegative() {
    item = new Item(conjured, -1, 40);
    updater = new ConjuredItemUpdater(item);
    updater.updateQuality(item);
    assertEquals(36, updater.item.quality);
}

@Test
void qualityIsAtLeast0() {
    item = new Item(conjured, 30, 0);
    updater = new ConjuredItemUpdater(item);
    updater.updateQuality(item);
    assertEquals(0, updater.item.quality);
}

@Test
void qualityIsAtMost50() {
    item = new Item(conjured, 30, 50);
    updater = new ConjuredItemUpdater(item);
    updater.updateQuality(item);
    assertEquals(50, updater.item.quality);
}
}
```

Capitolo 4

Visualizzazione dei risultati su Jenkins

A questo punto se raggiungiamo Jenkins ci viene mostrata la dashboard:

The screenshot shows the Jenkins dashboard interface. At the top is a black header with the Jenkins logo, a search bar, and user information (admin, log out). Below the header is a sidebar with navigation links: New Item, People, Build History, Manage Jenkins, My Views, and Open Blue Ocean. The main content area displays a table of build results for the project 'The-Gilded-Rose'. The table has columns for status (S, W), name, last success, last failure, last duration, and favorite status. The 'The-Gilded-Rose' project is shown with a green checkmark icon, indicating a successful build. Below the table, there are links for 'Icon legend', 'Atom feed for all', 'Atom feed for failures', and 'Atom feed for just latest builds'. On the left side of the dashboard, there are two expandable sections: 'Build Queue' and 'Build Executor Status'. The 'Build Queue' section shows 'No builds in the queue.' and the 'Build Executor Status' section shows '1 Idle' and '2 Idle'.

S	W	Name ↓	Last Success	Last Failure	Last Duration	Fav
✓	☀	The-Gilded-Rose	7 min 51 sec #44	2 mo 0 days #2	1.5 sec	▶ ☆

Icon: S M L

Icon legend

Atom feed for all

Atom feed for failures

Atom feed for just latest builds

Build Queue

No builds in the queue.

Build Executor Status

1 Idle

2 Idle

Come si nota dall'icona del sole, il progetto **The-Gilded-Rose** è *stabile*, nel senso che nessuna build recente ha fallito. Se clicchiamo sul progetto otteniamo informazioni più dettagliate sullo status:

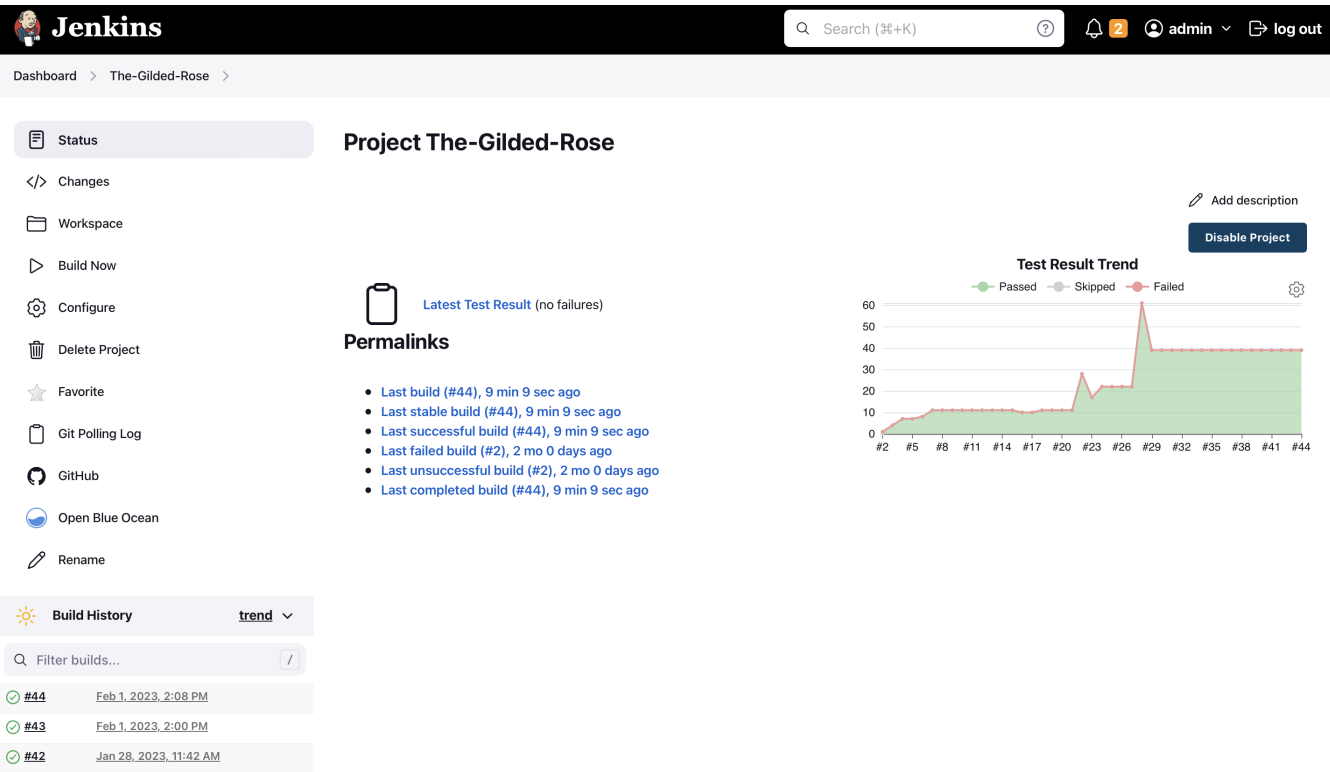
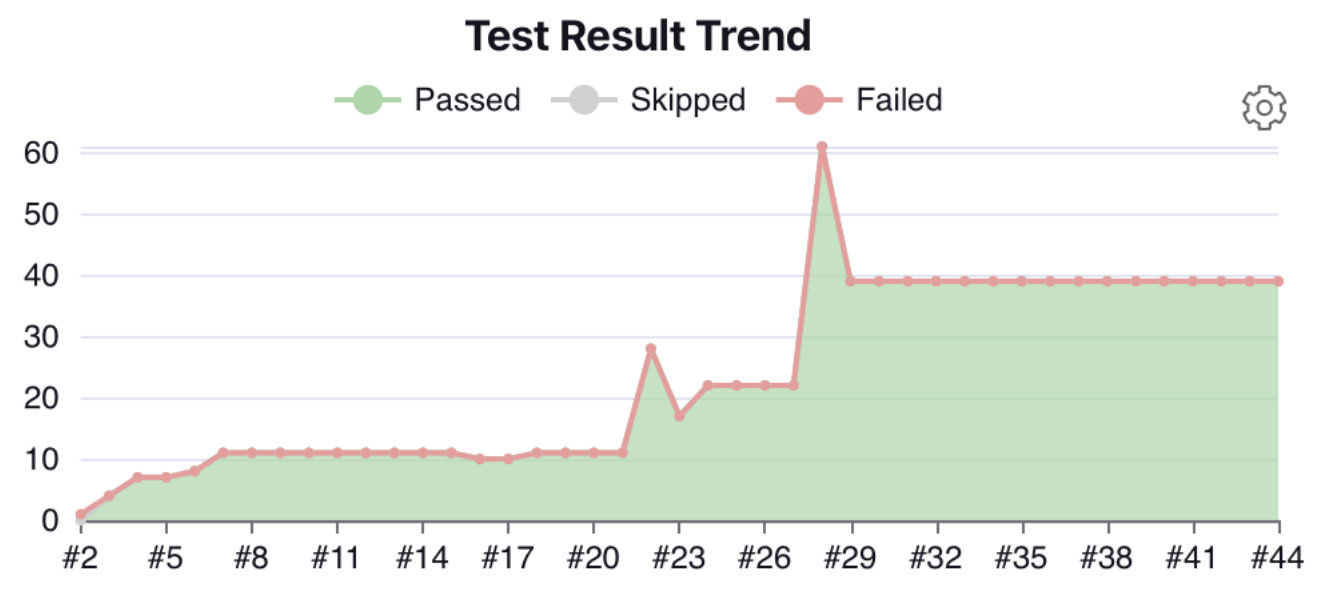


Figura 4.1: Vista del progetto

Come si nota, Jenkins genera alcuni link permanenti (*permalinks*) che puntano ad alcune build interessanti del progetto come, ad esempio, l'ultima build (*last build*), oppure l'ultima build ad aver fallito (*last failed build*). Sulla destra troviamo il trend dei risultati dei test: in verde quelli che hanno avuto successo, in grigio quelli saltati (nessuno) ed in rosso quelli falliti (uno solo all'inizio). Sulle ascisse c'è il progressivo relativo alla build, mentre sulle ordinate il numero di test lanciati. Coerentemente ad uno dei nove principi della *continuous integration*, i risultati delle build sono visibili a tutto il team e non solo allo sviluppatore sul suo IDE.



Se clicchiamo su *Latest Test Result* otteniamo una lista di tutti i test del progetto:

Jenkins

Search (#+K)

?

2

admin

log out

Dashboard

The-Gilded-Rose

#44

Test Results

Status

Changes

Console Output

Edit Build Information

History

Polling Log

Git Build Data

Test Result

Open Blue Ocean

Previous Build

Test Result

0 failures (±0)

39 tests (±0)
Took 32 ms.


Add description






All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
factories	13 ms	0		0		8		8	
implementations	6 ms	0		0		22		22	
interfaces	3 ms	0		0		9		9	


Per ciascuna classe di test, come si vede, ci vengono fornite informazioni interessanti come – ad esempio – la durata per l’esecuzione. L’informazione non è banale, dal momento che alcuni test


potrebbero richiedere molto tempo in altri contesti, riducendo di fatto la produttività del team. Se clicchiamo su un qualsiasi classe di test, come ad esempio `implementations/AgedBrieUpdaterTest`, otteniamo informazioni più specifiche:


 **Jenkins**


   admin   log out


[Dashboard](#) > [The-Gilded-Rose](#) > [#44](#) > [Test Results](#) > [implementations](#) > [AgedBrieUpdaterTest](#)


 Status


 Changes


 Console Output


 Edit Build Information


 History

 Polling Log

 Git Build Data

 Test Result


 Open Blue Ocean

 Previous Build

Test Result : AgedBrieUpdaterTest

0 failures (±0)

5 tests (±0)
Took 2 ms.

 Add description

All Tests

Test name	Duration	Status
qualityIsAtLeast0()	1 ms	Passed
qualityIsAtMost50()	1 ms	Passed
qualityShouldIncreaseBy1WhenSellIsPositive()	0 ms	Passed
qualityShouldIncreaseBy1WhenSellIsZero()	0 ms	Passed
qualityShouldIncreaseBy2WhenSellIsNegative()	0 ms	Passed

È possibile ottenere informazioni ancora più dettagliate come, ad esempio, il changelog di una certa build cliccando sul tasto *Changes* a sinistra:

Status

Changes

Console Output

Edit Build Information

Delete build '#44'

Polling Log

Git Build Data

Test Result

Open Blue Ocean

Previous Build

Changes

Summary

1. Made DailyUpdaterTest cleaner (commit: b73c6ed) (details)

Commit b73c6ed522238e28cb025c362df4b643da3ed14d by a.quirile

Made DailyUpdaterTest cleaner

(commit: b73c6ed)

.gradle/7.6/fileHashes/fileHashes.lock (diff)

.gradle/7.6/executionHistory/executionHistory.bin (diff)

.gradle/7.6/fileHashes/fileHashes.bin (diff)

.gradle/7.6/executionHistory/executionHistory.lock (diff)

src/test/java/interfaces/DailyUpdaterTest.java (diff)

.gradle/7.6/fileHashes/resourceHashesCache.bin (diff)

.gradle/buildOutputCleanup/buildOutputCleanup.lock (diff)

Oppure la *Console Output* per capire quali comandi Jenkins sta lanciando (ovviamente in maniera coerente rispetto al setup di quel progetto, come visto nel Capitolo 2):

Status

Changes

Console Output

Edit Build Information

Delete build '#44'

Polling Log

Git Build Data

Test Result

Open Blue Ocean

Previous Build

Console Output

Started by an SCM change

Running as SYSTEM

Building in workspace /Users/alessandroquirile/.jenkins/workspace/The-Gilded-Rose

The recommended git tool is: NONE

No credentials specified

> git rev-parse --resolve-git-dir /Users/alessandroquirile/.jenkins/workspace/The-Gilded-Rose/.git # timeout=10

Fetching changes from the remote Git repository

> git config remote.origin.url https://github.com/alessandroquirile/The-Gilded-Rose.git # timeout=10

Fetching upstream changes from https://github.com/alessandroquirile/The-Gilded-Rose.git

> git --version # timeout=10

> git --version # 'git version 2.37.1 (Apple Git-137.1)'

> git fetch --tags --force --progress -- https://github.com/alessandroquirile/The-Gilded-Rose.git +refs/heads/*:refs/remotes/origin/* # timeout=10

Seen branch in repository origin/main

Seen 1 remote branch

> git show-ref --tags -d # timeout=10

Checking out Revision b73c6ed522238e28cb025c362df4b643da3ed14d (origin/main)

> git config core.sparsecheckout # timeout=10

> git checkout -f b73c6ed522238e28cb025c362df4b643da3ed14d # timeout=10

Commit message: "Made DailyUpdaterTest cleaner"

> git rev-list --no-walk 4a5cd6c443ff099ac3b3f54438b99fc5d81df845b # timeout=10

[Gradle] - Launching build.

[The-Gilded-Rose] \$ /Users/alessandroquirile/.jenkins/workspace/The-Gilded-Rose/gradlew test

> Task :compileJava

> Task :processResources

> Task :classes

> Task :compileTestJava

> Task :processTestResources NO-SOURCE

> Task :testClasses

> Task :test

BUILD SUCCESSFUL in 799ms

4 actionable tasks: 4 executed

Build step 'Invoke Gradle script' changed build result to SUCCESS

Recording test results

[Checks API] No suitable checks publisher found.

Finished: SUCCESS

Executed Gradle Tasks

Task :compileJava

Task :processResources

Task :classes

Task :compileTestJava

Task :processTestResources

Task :testClasses

Task :test

Oppure, ancora, possiamo osservare il *Polling Log* che ha innescato quella specifica build:

The screenshot shows the Jenkins web interface. The top navigation bar includes the Jenkins logo, a search bar, and user information (admin). The breadcrumb trail is: Dashboard > The-Gilded-Rose > #44 > Polling Log. On the left sidebar, the 'Polling Log' option is selected. The main content area is titled 'Polling Log' and contains a message: 'This page captures the polling log that triggered this build.' Below this, a code block displays the following log output:

```
Started on Feb 1, 2023, 2:07:59 PM
Polling SCM changes on built-in
Using strategy: Default
[poll] Last Built Revision: Revision 4a5cd6c443ff098ac3b3f54430b9fc5d81df8450 (origin/main)
The recommended git tool is: NONE
No credentials specified
> git rev-parse --resolve-git-dir /Users/alessandroquirile/.jenkins/workspace/The-Gilded-Rose/.git # timeout=10
Fetching changes from the remote Git repositories
> git config remote.origin.url https://github.com/alessandroquirile/The-Gilded-Rose.git # timeout=10
Fetching upstream changes from https://github.com/alessandroquirile/The-Gilded-Rose.git
> git --version # timeout=10
> git --version # 'git version 2.37.1 (Apple Git-137.1)'
> git fetch --tags --force --progress -- https://github.com/alessandroquirile/The-Gilded-Rose.git
+refs/heads/*:refs/remotes/origin/* # timeout=10
Polling for changes in
Seen branch in repository origin/main
Seen 1 remote branch
> git show-ref --tags -d # timeout=10
> git log --full-history --no-abbrev --format=raw -M -m
4a5cd6c443ff098ac3b3f54430b9fc5d81df8450..b73c6ed522238e28cb025c362df4b643da3ed14d # timeout=10
Done. Took 2.6 sec
Changes found
```

Se torniamo alla pagine di vista del progetto come in Figura 4.1 possiamo consultare il *Workspace* che contiene tutti gli artefatti generati, come accade sulla pagina GitHub del progetto:

Jenkins Search (⌘+K) ? 2 admin log out

Dashboard > The-Gilded-Rose >

Status

Changes

Workspace

Wipe Out Current Workspace

Build Now

Configure

Delete Project

Favorite

Git Polling Log

GitHub

Open Blue Ocean

Rename

Workspace of The-Gilded-Rose on Built-In Node

The-Gilded-Rose /

- .git
- .gradle
- .idea
- build
- gradle/wrapper
- src
- build.gradle Dec 2, 2022, 5:03:10 PM 291 B
- gradlew Dec 2, 2022, 5:03:10 PM 7.88 KB
- gradlew.bat Dec 2, 2022, 5:03:10 PM 2.70 KB
- README.md Jan 16, 2023, 4:50:23 PM 404 B
- settings.gradle Dec 2, 2022, 5:03:10 PM 38 B

(all files in zip)

Se clicchiamo su *Open Blue Ocean* possiamo aprire una UI che mostra in maniera alternativa tutte le informazioni di cui sopra, come ad esempio la vista generale sul progetto:

Jenkins

PipelinesAdministration

The-Gilded-Rose ☆ ⚙️

ActivityBranchesPull Requests

Logout

Run

Disable

STATUS	RUN	COMMIT	MESSAGE	DURATION	COMPLETED
✓	42	—	Started by an SCM change	6s	4 days ago
✓	43	—	Made ItemUpdaterFactoryTest cleaner	2s	23 minutes ago
✓	44	—	Made DailyUpdaterTest cleaner	2s	15 minutes ago
✓	40	—	Added @BeforeEach annotation in SulfurasUpdaterTest	7s	7 days ago
✓	39	—	Item[] is now List<Item> in GildedRose class	2s	7 days ago
✓	38	—	Improved ItemUpdaterFactoryTest readability	2s	7 days ago
✓	37	—	Raise is now Increase and Degradate is now Decrease in test suites	7s	10 days ago
✓	36	—	Extracted method from ItemUpdaterFactory.getUpdater logic	2s	14 days ago
✓	35	—	Constants.java is now InStock.java	6s	15 days ago

Oppure i risultati dei test:

The-Gilded-Rose < 44

Pipeline
Changes
Tests
Artifacts
⚙️
📄
Logout
✕

Branch: — 2s Changes by a.quirile
Commit: — 15 minutes ago Started by an SCM change

All tests are passing

Nice one! All 39 tests for this pipeline are passing.

Passed - 39

✓	> getProperUpdaterForConjuredItem() - factories.ItemUpdaterFactoryTest	<1s
✓	> getProperUpdaterForBackstagePasses() - factories.ItemUpdaterFactoryTest	<1s
✓	> shouldThrowTechnologyNotSupportedYetExceptionWhenStrategyIsNotDaily() - factories.ItemUpdaterFactoryTest	<1s
✓	> getProperUpdaterForRegularItem() - factories.ItemUpdaterFactoryTest	<1s
✓	> getProperUpdaterForAgedBrie() - factories.ItemUpdaterFactoryTest	<1s
✓	> getProperUpdaterForDailyUpdate() - factories.ItemUpdaterFactoryTest	<1s
✓	> getProperUpdaterForSulfuras() - factories.ItemUpdaterFactoryTest	<1s
✓	> shouldThrowIllegalArgumentExceptionWhenItemHasNoUpdater() - factories.ItemUpdaterFactoryTest	<1s
✓	> qualityIsAtLeast0() - implementations.AgedBrieUpdaterTest	<1s
✓	> qualityIsAtMost500() - implementations.AgedBrieUpdaterTest	<1s
✓	> qualityShouldIncreaseBy1WhenSellInIsZero() - implementations.AgedBrieUpdaterTest	<1s
✓	> qualityShouldIncreaseBy2WhenSellInIsNegative() - implementations.AgedBrieUpdaterTest	<1s
✓	> qualityShouldIncreaseBy1WhenSellInIsPositive() - implementations.AgedBrieUpdaterTest	<1s
✓	> qualityShouldIncreaseBy1WhenSellInIsMoreThan10() - implementations.BackstagePassesUpdaterTest	<1s
✓	> qualityIsAtLeast0() - implementations.BackstagePassesUpdaterTest	<1s
✓	> qualityIsAtMost500() - implementations.BackstagePassesUpdaterTest	<1s

Nella sezione *Artifacts* in alto si può accedere ad una lista di artefatti costruiti dallo script gradle impostato precedentemente:

The-Gilded-Rose < 44

Pipeline
Changes
Tests
Artifacts
⚙️
📄
Logout
✕

Branch: — 2s Changes by a.quirile
Commit: — 16 minutes ago Started by an SCM change

NAME	SIZE	
pipeline.log	-	

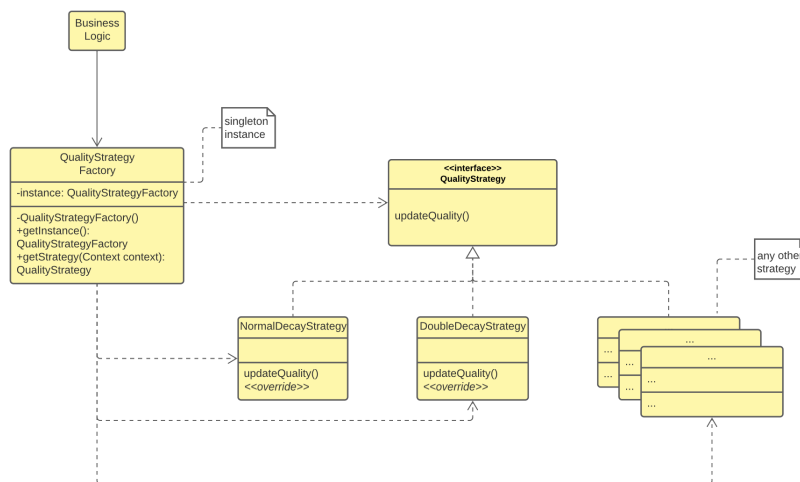
Se nelle impostazioni di Jenkins del Capitolo 2 scegliessimo oltre all'esecuzione dei test, attraverso il comando `./gradlew test`, anche la generazione di un file `.jar`, attraverso il comando `./gradlew jar`, l'eseguibile sarà visibile qui e scaricabile attraverso un semplice click.

Capitolo 5

Considerazioni finali e soluzioni alternative

Come visto, ho risolto i bad smell facendo uso del design pattern State/Strategy in sinergia con Factory Method e Singleton, sfruttando il polimorfismo tra oggetti e aumentando il livello di astrazione; di contro è aumentato il numero di classi nel package.

Una soluzione alternativa sarebbe consistita nel costruire a runtime oggetti `QualityStrategy` che specificano la strategia di aggiornamento della qualità di un qualsiasi `Item`: ad esempio `NormalDecayStrategy` il quale specifica che l'`Item` a cui è associato degrada di 1 unità alla volta – come accade per `RegularItem`; ritengo, tuttavia, che sarebbe stata una soluzione meno intuitiva dell'altra. Il seguente class diagram illustra la soluzione alternativa appena descritta:



Sono del parere che

"Simplicity is prerequisite for reliability"

– Edsger W. Dijkstra

Per cui ritengo ottimale la prima soluzione proposta. Certo, in ingegneria del software non esiste mai la *soluzione perfetta* e bisogna trovare un compromesso adeguato. Quindi la prima soluzione aumenta la qualità del software percepita dagli altri sviluppatori: rende la fase di manutenzione, in particolare, molto più lineare e indirizza più naturalmente il software verso quello che viene chiamato clean code attraverso astrazioni *semplici* che consentano di progettare per *anticipare il cambiamento*:

"Clean code is early building of simple abstractions"

– Ron Jeffries

Bibliografia

- [1] Robert C. Martin: Clean Code - A Handbook of Agile Software Craftsmanship
- [2] Martin Fowler: Refactoring - Improving the Design of Existing Code
- [3] Michael Feathers: Working effectively with Legacy Code
- [4] Paul Amann & Jeff Offutt: Introduction to Software Testing
- [5] Alexander Shvets: Dive into Design Patterns
- [6] Alexander Shvets: Refactoring Guru
- [7] Robert C. Martin: Agile Software Development - Principles, Patterns and Practises