PROGETTO SAGRADA

Autore: Daniele Comi
Data stesura: 12 aprile 2018
Ultima revisione:

1 Regole per lo sviluppo in team delle procedure Java

1.1 Introduzione

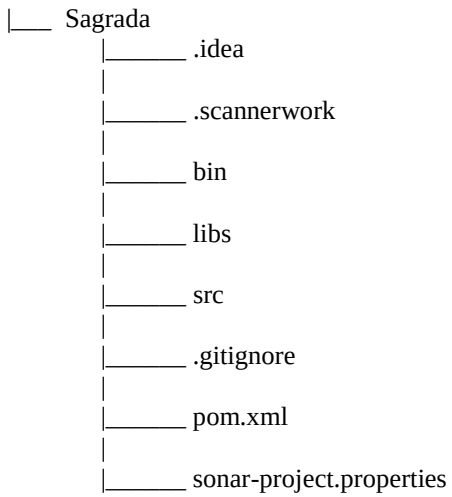
Scopo del presente documento è quello di riassumere le regole generali da adottare nello sviluppo delle procedure Java. L'obiettivo è quello di identificare delle procedure standard che rendano uniforme la procedura di sviluppo e consentano la cooperazione tra diversi sviluppatori sullo stesso progetto.

1.2 Organizzazione del codice

Il codice Java prodotto deve essere ordinato utilizzando in maniera opportuna la strutturazione a *package* tipica del linguaggio Java. La dislocazione del codice è prefissata per consentire l'uniformità tra i vari ambienti di sviluppo e per permettere l'utilizzo di uno strumento comune di *versioning*.

1.2.1 Struttura generale dell'ambiente operativo Java

Tutto lo sviluppo Java relativo ai progetti in corso d'opera deve essere svolto all'interno di una struttura a directory ben definita. Più precisamente, l'ambiente di sviluppo deve avere la seguente struttura:



In dettaglio:

- **.idea:** directory contente le varie personalizzazioni e impostazione relative all'IDE in utilizzo: IntelliJ IDEA 2018.
- **.scannerwork:** directory contente impostazione e log relative alle precedenti e correnti revisioni automatizzate dei sorgenti del progetto Sagrada attraverso il software SonarQube.
- **bin:** directory che raccoglie le compilazioni, cioè tutti i file *.class generati dalla compilazione dei sorgenti. Unitamente ai compilati deve contenere tutto il necessario al corretto funzionamento del software sviluppato. Questa directory sarà la base per il rilascio della release candidate ed eventualmente la final-stable release.
- Questa directory **non** contiene sorgenti quindi non è sottoposta a controllo di versione.
- **libs:** directory che raccoglie tutte le librerie necessarie allo sviluppo Java. Le librerie devono preferibilmente essere in formato compresso *.jar o *.zip; evitare il più possibile l'utilizzo di librerie basate su file system. Unitamente alle librerie questa directory include anche un file di testo UTF-8 che elenca le librerie contenute nella directory e alcune informazioni legate alle versioni e alla data di aggiornamento. Ogni qual volta si aggiunge una nuova libreria è necessario modificare questo file di testo ed aggiungere le informazioni relative alla nuova libreria introdotta.
- **src:** directory in cui sono presenti tutti i sorgenti del progetto Sagrada, raggruppati in diversi package secondo dei criteri che verranno dopo definiti. Ulteriormente ai sorgenti relativi al progetto, in questa directory sarà contenuta un secondo package radice relativo al testing che verrà descritto nei prossimi paragrafi. Entrambi i package radice e tutto il loro contenuto verranno sottoposti a intensivo testing e versioning.

- **.gitignore:** file descrittore per l'applicativo git utilizzato per il versioning in team, esso conterrà una descrizione rigorosa di quali file non devono essere soggetti al versioning, motivo di ciò è per garantire una maggiore portabilità, modularità e leggerezza al progetto Sagrada in modalità versioned.
- **pom.xml:** file descrittore del progetto Sagrada utilizzante il framework Maven 3, esso conterrà, oltre ad una struttura descrittiva del progetto Sagrada in XML per garantire la portabilità tra i diversi ambienti Maven, la lista di dipendenze utilizzate e necessarie per la corretta build del progetto insieme al successo nell'esecuzione di ogni Maven test.
- **sonar-project.properties:** file descrittore della revisione automatizzata del progetto Sagrada attraverso il software SonarQube. Esso conterrà la locazione dei file .class contente Java bytecode e la locazione dei sorgenti .java.

1.2.2 Organizzazione di massima del codice sorgente

All'interno della directory src che racchiude tutto il codice è necessario organizzare i file secondo una struttura ben precisa che evidenzi tutti i moduli operativi che compongono lo sviluppo in Java.

L'organizzazione del codice si attesta su una struttura a più livelli, implementata attraverso l'utilizzo dei costrutti di packaging tipici del linguaggio Java.

La struttura generale dei sorgenti può essere riassunta con questo primo schema:

src……. Directory che raccoglie tutti i sorgenti. Vedi punto precedente
|
|_____**main/java**……Non contiene sorgenti, serve solo per raggruppare i package
|
|___ **interfaces**…… **(M)** package contente interfacce **base**
|
|___ **abstracts**……. **(M)** package contente classi astratte **base**
|
|___ **network**……. **(M)** package contente la struttura delle comunicazioni di rete
|
|___ **animations**…. **(M)** package contente le classi costituenti decorator per oggetti animabili
|
|___ **ui**…… **(M)** package contente classi per la principale struttura d'interfaccia grafica.
|…….|
|___ **fx**…….. **(M)** package contente classi costituenti oggetti grafici animabili attraverso JavaFX

Per ogni MP (eventualmente, come sopra mostrato):
……|
|___ **nome sotto package**…. **(SP)** Sub package per migliore organizzazione di MP

(MP) = main package (**SP)** = sub package

Si noti in particolare:

- Ogni main package fa riferimento a un preciso modulo del progetto ed ha lo scopo di fornire un contenitore comune a tutti i sorgenti sviluppati nel contesto di un preciso progetto.
- I nomi dei package sono costruiti da una radice identificante in modo chiaro il ruolo del package all'interno del progetto, se è eventualmente presente un sub package viene su di esso applicato lo stesso tipo di regola del main package. La creazione di un nuovo package deve essere valutata con attenzione e deve essere decisa di comune accordo tra gli sviluppatori. In generale vale la regola che ogni package raggruppa un preciso filone dello sviluppo Java e quindi è frutto di una precisa attività di analisi.
- L'alfabeto utilizzabile nella denominazione dei package è quello inglese di soli caratteri minuscoli

1.3 Convenzioni relative al linguaggio Java

Allo scopo di migliorare la comprensibilità e l'interscambio del codice sviluppato è opportuno adottare nella scrittura dei sorgenti alcune regole di stile.

1. **Nome delle classi:** il nome delle classi deve essere il più possibile significativo e deve dare un'idea di massima della funzione svolta dalla classe stessa.
   Attenersi pertanto alle seguenti regole:

   - Evitare nomi troppo lunghi, senza significato o contenenti caratteri speciali che potrebbero dare problemi.
   - Il nome della classe deve iniziare con una maiuscola così come ogni parola che compone il nome. Un esempio un nome corretto può essere **ObjectFieldListener** mentre indubbiamente la forma **objectfieldlistener** è meno leggibile e quindi preferibilmente da evitare.
   - Il nome della classe deve possibilmente contenere anche un riferimento alla sua funzione all'interno del linguaggio: ad esempio, se una classe definisce un listener di un oggetto è opportuno che il suo nome finisca con la locuzione **Listener**; altri oggetti per cui può valere questa regola sono le Interface, gli Event, le Exception. Esempio: la classe che definisce una eccezione avrà un nome del tipo **PippoException**. Viene lasciato come discussione nel team la scelta dell'aggiunta del prefisso o postfisso **Abstract** nel caso di denominazione di classi astratte. Altri esempi: NetworkController, NetworkService, NetworkManager ecc…

2. **Nome dei metodi.**
   - Evitare nomi lunghi e con caratteri speciali.
   - Il nome del metodo deve iniziare con una lettera minuscola. Se il nome è composto da più parole le parole intermedie devono però iniziare con una maiuscola. Ad esempio, un nome valido per un metodo può essere **printObjectList.**
   - Seguire il più possibile lo standard dei JavaBean, in modo da facilitare l'eventuale utilizzo di una classe all'interno di sistemi introspettivi (si rimanda alla documentazione dei ***Java Beans***):
     - I nomi dei metodi che assegnano il valore di una variabile di classe sono del tipo **set + Nome variabile**: ad esempio setCode è il metodo che assegna un valore all'attributo codice. Da notare che, se presente, la prima lettera delle variabili viene ignorata perché relativa al contesto della variabile (vedi sezione sulle variabili per maggiori dettagli).
     - I nomi dei metodi che leggono il valore di una variabile di classe sono del tipo **get + Nome variabile**: stesso principio per il caso dei metodi setter.
     - I nomi dei metodi che leggono il valore di una variabile logica sono del tipo **is + NomeVariabile**: ad esempio, il metodo che torna lo stato del flag iAllocated sarà del tipo isAllocated(). I metodi di questo tipo devono sempre tornare un valore di tipo Boolean.
   - Se il metodo prevede più combinazioni nei parametri di ingresso ripetere n volte la dichiarazione del metodo, ogni volta con il numero minimo di parametri di ingresso. Evitare la scrittura di metodi con molti parametri se si sa che gran parte di quei parametri si useranno solo in casi particolari: meglio scrivere una nuova dichiarazione del metodo da usarsi solo in quei casi.

3. **Nome delle variabili e degli attributi.**
   - Al solito, evitare nomi troppo lunghi.
   - Il nome deve iniziare con una lettera minuscola. Se il nome è composto da più parole le parole intermedie devono però iniziare con una maiuscola. È possibile utilizzare in certi casi solamente il carattere '_'.
   - Il nome assegnato alla variabile/attributo deve dare un'idea di dove sia dislocata la variabile/attributo stessa all'interno della classe. Quindi:
     - **Attributi di classe:** sono le variabili definite a livello di classe e che sono pertanto disponibili in ogni punto del sorgente. Il loro nome può avere una lettera iniziale per una maggiore localizzazione.
     - **Variabili locali:** sono le variabili definite all'interno di un metodo e valide solo all'interno del metodo in cui sono definite. Il loro nome può avere una lettera iniziale per una maggiore localizzazione.
     - **Attributi passati ad un metodo:** sono le variabili che definiscono i campi di input previsti da un metodo. Il linguaggio Java prevede che valgano solo all'interno del metodo in cui sono definite. Ad esempio:

       public void printHash(String hash)

     - **Costanti:** sono variabili dichiarate di tipo static e final. In questo caso il nome è tutto maiuscolo. Esempio:

       public static final String FONT_LOCALE = "/fonts"
       private static final String FONT_LOCALE "/fonts"

È da considerate la stessa politica di scrittura in maiuscolo il nome delle constanti nelle enumerazioni.

- **Contatori:** le regole precedenti trovano eccezione nel caso di variabili usate per definire i contatori all'interno di costrutti di loop. In questo caso è conveniente utilizzare nomi brevi ma significativi, come **index**, **counter**, **key**…
  Nel caso di loop annidati utilizzare contatori diversi. Ad esempio:

```
for (int counter = 0; counter < 10; counter ++)
{
        for (int _counter = 10; _counter > 0; _counter--)
        {
                …..
        }
}
```

  Si può eventualmente utilizzare nomi brevi di una lettera quali i, j, k… nel caso si venga alla decisione garantisca una maggiore leggibilità al codice oltre al fatto della non importanza della denominazione di un certo indice.

- Dichiarare preferibilmente variabili protette o private e definire gli opportuni metodi di set o di get se si vuole rendere pubblico l'accesso alle variabili da classi esterne. Nel codice scritto evitare se possibile l'accesso diretto alle variabili di una classe esterna, cioè assegnamenti del tipo Classe.variabile = valore, perché non compatibile con le specifiche JavaBeans (anche se sintatticamente corretto).
  Ad esempio, il codice:

```
MyClass myClass = new myClass();
myClass.myAttribute = "Java";
```

  è sintatticamente corretto ma non viene riconosciuto da un sistema di introspezione che si aspetta la presenza di un metodo setMyAttribute(String attributeValue) definito all'interno della classe MyClass. In questo caso, il codice corretto per l'assegnamento di una variabile diventa allora del tipo:

```
MyClass myClass = new MyClass();
myClass.setMyAttribute("Java");
```

4. Scrittura e semantica dei sorgenti

- Evitare l'implementazione di metodi troppo lunghi, si cerca di spingere ad un'implementazione di progetto costituita da molte classi con poco codice, e di conseguenza metodi piccoli che implementano la minima funzionalità necessaria
- Non scrive metodi con molti parametri, se la firma del metodo contiene più di 4/5 parametri è necessario valutare se è il caso di incapsulare questi parametri in un'apposita classe (DTO/Model)
- Ritornare oggetti strutturati ove necessario
- Utilizzare le parentesi graffe dove necessario il loro utilizzo deve essere affiancato durante l'apertura alla linea di codice corrente, esempio:

```
try {
        //codice
}
catch(MyException exc) {
        //altro codice
}
```

```
class MyClass {

        //codice
}
```

-Viene lasciato almeno uno spazio tra i vari metodi dichiarati e una linea vuota iniziale per ogni tipo di dichiarazione o espressione d'utilizzo, nessuno spazio tra gli attributi tranne uno spazio divisorio per gruppi di attributi diverso tipo. Esempio: attributi e valori costanti.
-Viene lasciato uno spazio bianco per ogni cambio di parametro ed espressione e sempre uno dopo una virgola ma non prima di essa.

- Nel caso dei costruttori : non verrà mai lasciato il costruttore di default, verrà sempre implementato un proprio costruttore, anche nel caso in cui funge da inizializzatore di attributi con nessun parametro formale richiesto.
- Rendere quanto più possibili funzionale il codice attraverso l'utilizzo delle nuove funzionalità introdotte dal JDK 8
- Favorire l'utilizzo di Enumeration al posto di Boolean quando si hanno espressioni multi valore, esempio:

  String getDate(Date date, Boolean beforeMidnight);

  deve diventare:

  ```
  enum TimeBound {
        MIDNIGHT,
        BEFORE_MIDNIGHT
  }
  String getDate(Date date, TimeBound timeBound);
  stringDateFormat = getDate(myDate, TimeBound.BEFORE_MIDNIGHT);
  ```

- Condurre le espressioni all'interno di variabili booleane: cercare di lasciare le condizioni di valutazione booleane il più pulite e chiare possibili, esempio:

  ```
  booleanCondition = evaluate();
  if (booleanCondition)
        doSomething();
  ```

- Le righe del codice devono essere **sempre** indentate attraverso l'utilizzo di un tabulatore.
  Nonostante molti IDE utilizzino anche la politica di indentazione a 4 spazi, nel suddetto progetto verrà ammessa solamente l'indentazione attraverso il tabulatore.

5. Implementazione

- È sempre e comunque necessario ricordare di rispettare una certa complessità temporale, sebbene si possa utilizzare in pre RC durante il versioning e quindi temporaneamente una complessità temporale pari a $O(N^2)$, la complessità temporale a cui attenersi nella RC è pari a O(NlogN).
- È sempre necessario un buon utilizzo di Java Collections che sfruttino inoltre il polimorfismo dovuto alla loro implementazione gerarchica, ovvero come esempio: se si volesse utilizzare un ArrayList come tipo di attributo o tipo di ritorno o qualsiasi altro valido utilizzo è necessario dichiarare questo attributo o variabile come List e non ArrayList, segue lo stesso ragionamento per tutti gli altri tipi di strutture dati.
- È necessario un buon utilizzo dei tipi generici in Java introdotti con il JDK 5 e inoltre il loro miglioramento con le wildcard introdotto con il JDK 7.
- Si richiede una forte modularizzazione e indipendenza tra le varie classi del progetto, questo necessario per avere un'ottima e semplicistica implementazione del progetto a moduli intercambiabili.
- È necessario l'utilizzo di vari design pattern per poter andare a compimento del punto precedentemente descritto.
- È **assolutamente vietato** il non rispetto delle caratteristiche della programmazione a oggetti riguardante la visibilità degli attributi di classe e riguardante i tre principi fondamentali: ereditarietà, polimorfismo e incapsulamento.

6. Gestione delle eccezioni

- È necessario dichiarare le specifiche eccezioni che un metodo può lanciare. **Assolutamente vietato** gestire la sola eccezione generica Exception.
- Utilizzare il blocco finally per rilasciare le eventuali risorse acquisite
- È assolutamente vietato l'utilizzo di blocchi di catch senza alcun codice al loro interno
- Effettuare correttamente il wrap e log di un'eccezione, in modo da non perdere il suo stack trace.
  Se questo non è possibile o va contro l'implementazione in via di scrittura l'eccezione dovrà essere rilanciata al chiamante.
- È necessario dichiarare un nuovo tipo d'eccezione quando il suo eventuale lancio è previsto dall'implementazione.

1.4 Documentazione dei sorgenti

Al fine di migliorare la comprensibilità dei sorgenti è utile inserire all'interno del codice degli opportuni commenti che facilitino la comprensione delle procedure. Questo vale in particolar modo nel caso di procedure complesse o particolari.
La documentazione si sviluppa su due livelli distinti:

1.4.1 Documentazione a livello di classe

Il file sorgente di ogni classe deve riportare in testata una serie di informazioni che consentano di identificare le funzioni svolte da una singola classe. La scelta di associare o meno una descrizione alla classe deve essere valutata caso e per caso e tenere conto sia dell'importanza della classe stessa sia della riusabilità del codice prodotto. La descrizione deve essere posta in testa al sorgente e racchiusa all'interno di commenti Java attraverso l'utilizzo dei costrutti previsti dal linguaggio. Le informazioni contenute nella testata devono essere le seguenti:

1. Data di creazione della classe
2. Autore
3. Descrizione di massima della funzione svolta dalla classe. Il livello di meticolosità della descrizione deve essere proporzionato alla effettiva importanza della classe all'interno del progetto (ad esempio, evitare descrizioni dettagliate per classi di servizio o di scarsa riusabilità).
4. Se necessario, una descrizione dei parametri di ingresso previsti dalla classe e del loro effetto sul comportamento della classe stessa.
5. Se necessario, un esempio di utilizzo della classe. È sufficiente riportare qualche riga di codice che rappresenti un tipico caso di utilizzo della classe, possibilmente senza entrare troppo nel dettaglio.
6. Una lista delle modifiche maggiori che sono state apportate alla classe, con particolare attenzione alle modifiche che hanno introdotto modifiche sostanziali. Questa lista può integrare le descrizioni inserite nel server di gestione dei sorgenti al momento del rilascio di una classe.

1.4.2 Documentazione a livello di metodo

All'interno di una classe può essere utile inserire dei commenti a livello di singolo metodo. Questo diventa necessario soprattutto quando il metodo svolge funzioni importanti oppure prevede un gran numero di parametri di difficile comprensione. La documentazione a livello di metodo si deve limitare a fornire informazioni che rendano semplice l'utilizzo del metodo senza dover analizzare il codice in esso contenuto: deve quindi definire con buon dettaglio cosa fa un metodo, quali sono le informazioni da fornirgli e qual è l'effetto dell'esecuzione di un metodo sul resto della classe. La documentazione a livello di metodo è utile che abbia un formato di tipo **javadoc** (vedi documentazione Java per i dettagli)**,** in modo da facilitare una eventuale generazione automatica di documentazione operativa.

1.4.3 Altre forme di documentazione

Oltre ai casi visti in precedenza, ci possono essere altri casi particolari in cui può essere utile introdurre dei commenti ad un sorgente.

- È buona norma commentare i passaggi del codice che potrebbero essere di difficile interpretazione. Questo non vuol dire inserire commenti per ogni cosa che si fa ma solo per quei casi che potrebbero generare confusione ad un osservatore esterno.
- Segnalare sempre le parti di codice non completate o da sviluppare ulteriormente. Lo scopo è quello di inserire nel codice delle note che segnalino allo sviluppatore i punti da sviluppare ulteriormente e che consentano ad un osservatore esterno di capire se una funzione è completamente implementata o se manca qualcosa. Un metodo semplice può essere quello di inserire prima della parte incompleta un commento del tipo:

//TODO: descrizione delle parti mancanti previa implementazione

1.5 IDE, Testing, Repository e software ausiliari

Al fine di migliorare il processo di sviluppo del progetto Sagrada è necessario aggiungere una serie di punti riguardanti gli strumenti per lo sviluppo.

1.5.1 IDE

Per il progetto Sagrada verrà utilizzato l'IDE IntelliJ IDEA 2018 Community Edition.
Per una più facile integrazione tra i vari host è vietato l'utilizzo di IDE alternativi.

1.5.2 Testing

Nel progetto Sagrada il testing verrà effettuato attraverso il supporto di Maven 3. Esso è parte integrante del progetto ed è assolutamente necessario ad ogni nuova implementazione o aggiunta di classe o metodo o della loro relativa implementazione, aggiungere o cambiare le modalità e i processi di testing, in modo adeguato, descritti dalle classi di testing nel package radice main/test. Il superamento dei test garantirà una buona implementazione dell'astrazione precedentemente pensata. Non è possibile aggiungere alla repository comune sorgenti che non supereranno i test definiti. L'aggiornamento verrà rifiutato in modo automatico e notificato via e-mail.

1.5.3 Repository

Il versioning e il repository della codebase del progetto Sagrata verrà effettuata attraverso l'utilizzo di git e messo disponibili in rete attraverso GitHub alla seguente repository privata.
È necessario che ad ogni cambiamento rilevante ed aggiunta venga aggiornata il repository comune.
La gestione dei vari comandi è sia utilizzabile in modalità integrata attraverso l'IDE sia via linea di comando.

1.5.4 Software ausiliari

Per lo sviluppo del progetto verranno utilizzati due servizi e software ausiliari.

1. **SonarQube**

   SonarQube verrà utilizzato attraverso un'integrazione nell'IDE utilizzato in collegamento su una sessione avviata localmente. Esso permetterà il miglioramento dell'intero insieme di sorgenti del progetto andando a identificare oltre ad eventuali falle presenti e che sono state o ignorate o sfuggite ad uno o più sviluppatori all'interno del team, anche i punti dove il sorgente permette un miglioramento a livello di complessità sia spaziale che temporale. I cambiamenti richiesti e suggeriti da SonarQube verranno resi visibili automaticamente ad ogni tentativo di commit, build e avvio del progetto facilitando le correzioni.

2. **Travis CI**
   Travis CI è un servizio remoto che permetterà il testing del progetto globalmente prima di far sì che un commit venga definitivamente alla codebase principale od altri branch. Se il testing fallirà il commit verrà rifiutato effettuando una rollback allo stato precedente della codebase e una notifica di errore verrà inviata via e-mail. Questo permetterà la non presenza di codice non funzionante, supponendo che il test sia stato correttamente scritto, nella codebase principale ed altri relativi branch.