

Rudimenti di Ingegneria del processo di produzione software

Programmazione ad Oggetti – Lab09

Docenti: Roberto **Casadei**, Danilo **Pianini**
Tutor: Luca **Deluigi**

C.D.S. Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Campus di Cesena

22 novembre 2022



- 1 DVCS Workflow
- 2 Rudimenti di Gradle (e strumenti di automazione della build)
 - Il problema della gestione delle dipendenze
 - Strumenti di automazione della build
 - Gradle in OOP
- 3 Preparazione al laboratorio

- 1 DVCS Workflow
- 2 Rudimenti di Gradle (e strumenti di automazione della build)
 - Il problema della gestione delle dipendenze
 - Strumenti di automazione della build
 - Gradle in OOP
- 3 Preparazione al laboratorio



Dalle puntate precedenti

DVCS

- DVCS sono strumenti potenti per tenere traccia in maniera efficiente della storia di un progetto
- Nascono in particolare come evoluzione dei tradizionali VCS (SVN, CVS ...)
- Enfasi su una **miglior gestione del lavoro di team**

DVCS e teamwork

- “La potenza è nulla senza controllo!”
- Ovvero ... la mancanza di un metodo chiaro e condiviso per utilizzarli può portare a risultati **DEVASTANTI**
 - ▶ effort necessario per la parte di gestione diventa presto preponderante e insostenibile
- Ecco perché è bene adottare un **workflow collaborativo**
 - ▶ i vostri progetti e i vostri partner di progetto vi ringrazieranno!

Quale workflow

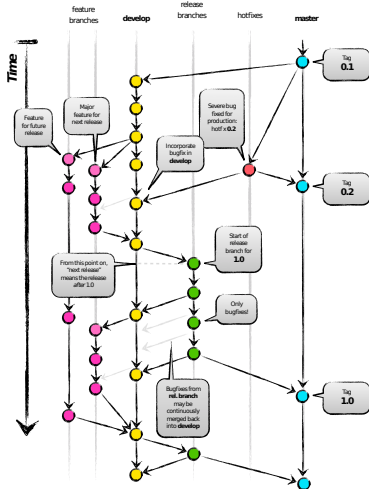
- Come si sceglie un workflow?
- Abbiamo parlato di semplicità...
- In realtà è più corretto parlare di giusto **trade-off** tra semplicità ed esigenze



Lo stato dell'arte: **git-flow** I

Definito da *Vincent Driessen* e spiegato in

<http://nvie.com/posts/a-successful-git-branching-model/>



Lo stato dell'arte: **git-flow** II

Alcune considerazioni

- Non lo useremo
 - ▶ troppo complicato per i nostri scopi
- Comunque molto interessante perché racchiude tutti gli aspetti di un DVCS workflow

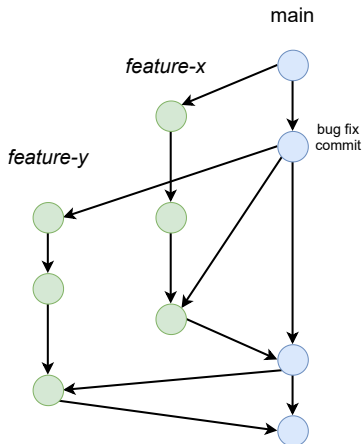
I branch

- Sono il supporto fondamentale alle fasi del ciclo di vita del software
- Ogni fase ha il proprio branch!
- Branching e merging all'ordine del giorno!

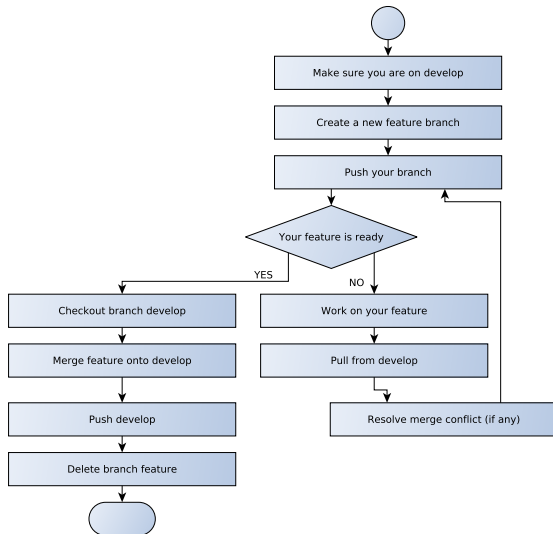


Un modello più semplice

- Un branch principale
- Feature branch diversi e indipendenti, per sviluppi diversi
 - ▶ da sincronizzare attraverso il branch principale (frequentemente per ridurre possibilità di conflitti complessi)



Feature branch I



Feature branch II

```
1 # Create a new feature branch from develop and share it
2 git checkout develop
3 git checkout -b feature-mynewfeaturename
4 git push origin feature-mynewfeaturename
5
6 # WHILE your_feature_is_unfinished
7 # work on your feature:
8 git add mynewfiles
9 git add mymodifiedfiles
10 git add mydeletedfiles
11 git commit -m "my commit message"
12 # Merge develop in to prevent big merge conflicts!
13 git pull origin/develop
14 # You may need to solve a merge conflict here!
15 # Share and save your work
16 git push
17 # END-WHILE
18
19 # Merge feature onto develop
20 git checkout develop
21 git merge feature-mynewfeaturename
22
23 # Push develop
24 git push
25
26 # Delete feature branch
27 git branch -d feature-mynewfeaturename
```

Il repo ufficiale del vostro progetto I

Approccio 1: workflow semplice

- Qualcuno di voi agirà come “repo maintainer”
- Creerà quindi il repository su GitHub
- Gli altri membri del team faranno la `clone`
- Ciascuno lavorerà parallelamente sul proprio repository locale (working copy), condividendo tramite `push` e `pull` il proprio lavoro con gli altri



Il repo ufficiale del vostro progetto II

Approccio 2: workflow avanzato

Ottimo per progetti di grosse dimensioni e/o per team molto eterogenei, dove qualcuno deve assicurarsi della qualità del codice prodotto da altri.

- Il maintainer crea il repository, ed è l'unico col diritto di scrittura
- Gli altri membri del team hanno una **fork** a testa
- Ciascuno lavora su una working copy, facendo pull dal repository "centrale" e push sulla propria fork
- Quando una feature è completa, o si arriva ad un buon grado di sviluppo, si apre una **pull request**
- Il maintainer revisiona il codice, assegna eventuali modifiche, e quando è soddisfatto accetta la pull request facendo il merge del codice nel repository principale

Questo workflow è un overkill per il progetto di OOP

- Ma è possibile che vi chiederemo di lavorare così, se farete tesi o un tirocini relativi ad alcuni nostri software

- 1 DVCS Workflow
- 2 Rudimenti di Gradle (e strumenti di automazione della build)
 - Il problema della gestione delle dipendenze
 - Strumenti di automazione della build
 - Gradle in OOP
- 3 Preparazione al laboratorio



- 1 DVCS Workflow
- 2 Rudimenti di Gradle (e strumenti di automazione della build)
 - Il problema della gestione delle dipendenze
 - Strumenti di automazione della build
 - Gradle in OOP
- 3 Preparazione al laboratorio



Mini challenge¹

Scrivere un programma che va su Internet, scarica da TheTVDB i titoli degli episodi di Breaking Bad in italiano, e poi li stampa.

```
1 package it.unibo.ci;
2 import java.io.IOException;
3 import org.apache.commons.io.IOUtils;
4 import com.omertron.thetvdbapi.TheTVDBApi;
5 import com.omertron.thetvdbapi.model.Episode;
6 import com.omertron.thetvdbapi.model.Series;
7 import static java.nio.charset.StandardCharsets.UTF_8;
8
9 public final class PrintBreakingBad {
10     private static final String LANG = "it";
11     private static final String SERIE = "Breaking Bad";
12     public static void main(String... args) throws ClassNotFoundException, IOException {
13         final var externalKeyFile = ClassLoader.getResourceAsStream("TheTVDBAPIKey")
14         final var key = IOUtils.toString(externalKeyFile, UTF_8);
15         final var api = new TheTVDBApi(key);
16         api.searchSeries(SERIE, LANG).stream()
17             .filter(s -> s.getSeriesName().equals(SERIE))
18             .map(Series::getId)
19             .flatMap(s -> api.getAllEpisodes(s, LANG).stream())
20             .map(Episode::getEpisodeName)
21             .forEach(System.out::println);
22     }
23 }
```

¹Si veda anche: <https://github.com/APICe-at-DISI/sample-gradle-project>

Sono state usate due librerie:

- Apache Commons IO (`commons-io:commons-io`)
- Omertron's TheTVDBApi (`com.omertron:thetvdbapi`)

...solo che per far funzionare TheTVDBApi servono altre due librerie...

- SLF4J (`org.slf4j:slf4j-api`)
- YAMJ (`org.yamj:api-common`)

...solo che per far funzionare YAMJ servono altre sei librerie...

...e per far funzionare alcune di queste servono altre librerie ancora



Albero delle dipendenze

Si crea un *albero* delle dipendenze!

```
1 +--- commons-io:commons-io:2.4
2 \--- com.omertron:thetvdbapi:1.7
3     +--- org.slf4j:slf4j-api:1.7.9                (A1)
4     \--- org.yamj:api-common:1.2
5         +--- org.apache.commons:commons-lang3:3.3.2
6         +--- commons-dbcp:commons-dbcp:1.4
7         |   \--- commons-pool:commons-pool:1.6    (B1)
8         +--- commons-pool:commons-pool:1.6        (B2)
9         +--- commons-codec:commons-codec:1.10     (C1)
10        +--- org.apache.httpcomponents:httpclient:4.3.6
11        |   +--- org.apache.httpcomponents:httpcore:4.3.3
12        |   +--- commons-logging:commons-logging:1.1.3
13        |   \--- commons-codec:commons-codec:1.10 (C2)
14        \--- org.slf4j:slf4j-api:1.7.9            (A2)
```

- La gestione manuale può diventare dispendiosa
 - ▶ Ad ogni aggiornamento bisogna ricontrollare tutto il sotto-albero
- Le dipendenze indirette (*transitive*) potrebbero avere versioni confliggenti!



Contesti delle dipendenze

Non è il solo problema

- Alcune dipendenze ci servono solo a runtime, non a compile time
 - ▶ ad esempio, l'implementazione SLF4J da usare per il logging
- Alcune dipendenze ci servono solo per il testing (JUnit ad esempio)

Andrebbero creati diversi classpath, uno per “scope”:

- implementazione (compilazione ed esecuzione)
 - ▶ per il prodotto
 - ▶ per i test
- solo compilazione (no runtime)
 - ▶ per il prodotto
 - ▶ per i test
- solo esecuzione (no compilazione)
 - ▶ per il prodotto
 - ▶ per i test

...i problemi cominciano a sommarsi...



- 1 DVCS Workflow
- 2 Rudimenti di Gradle (e strumenti di automazione della build)
 - Il problema della gestione delle dipendenze
 - Strumenti di automazione della build
 - Gradle in OOP
- 3 Preparazione al laboratorio



Strumenti di build automation: COSA

Build (automation) tool: automatizza la gestione del progetto

- Configurazione progetto
- Gestione delle dipendenze
 - ▶ Cerca le librerie
 - ▶ Le scarica
 - ▶ Prepara il classpath
- Compilazione e generazione artefatti
 - ▶ Compila il codice di produzione e il codice di test
 - ▶ Genera i JAR
 - ▶ Genera la documentazione (JavaDoc)
- Controllo qualità
 - ▶ Esegue i test
 - ▶ Verifica la qualità del codice
- Deployment
 - ▶ Ad esempio, rilascio su un host di librerie

In pratica, un build tool è uno strumento che cattura le funzionalità che spesso si trovano negli IDE



Strumenti di build automation: COME

- Esistono vari strumenti di build (anche all'interno di stesse comunità di linguaggi)—ad esempio ant, Maven, npm, sbt, ... e **Gradle**
- Uno strumento di build è un **programma**, da **installare** localmente²
- Uno strumento di build si può utilizzare da **linea di comando** o attraverso l'integrazione in un IDE (se supportato ad es. mediante plugin)
 - ▶ Un progetto Gradle può essere importato in Eclipse, ma anche in Netbeans o IntelliJ Idea

```
1 $ gradle --version
```

²<https://gradle.org/install/>



Concetti fondamentali di Gradle I

Uno strumento di build permette di eseguire **task** su un **progetto** (collezione di risorse) opportunamente configurato attraverso un **descrittore di build**

Progetto (modulo) e configurazione di build

Un **progetto Gradle** è l'insieme di cose che volete costruire o fare con Gradle, ed i file che consentono di farlo. Ogni progetto è costituito di **metadati/impostazioni** (ad es. nome e versione), **risorse** (ad es. sorgenti), e *task*

```
1 $ mkdir my-project && cd my-project
2 $ gradle init
```

- configurazione progetto nel descrittore di build `./build.gradle.kts`



Concetti fondamentali di Gradle II

Task

Operazione atomica eseguita sul progetto (ad esempio, compilazione di file Java, o impacchettamento di un jar). Può *dipendere* da altri task (ad esempio, la creazione di un jar eseguibile richiede che la compilazione dei sorgenti sia stata completata).

```
1 # Ottenere la lista dei task disponibili
2 gradle tasks --all
3 # Esecuzione del task di test
4 gradle test
```



Plugin

Componente che estende le funzionalità di Gradle, aggiungendo configurazioni e task

- Ad esempio il Java Plugin^a

^ahttps://docs.gradle.org/current/userguide/java_plugin.html



Configurazione per il nostro esperimento:

```
1 plugins {  
2     java  
3 }  
4 repositories {  
5     mavenCentral() // Where to look for jars  
6 }  
7 dependencies {  
8     implementation("commons-io:commons-io:2.4")  
9     implementation("com.omertron:thetvdbapi:1.7")  
10    testImplementation("junit:junit:4.12")  
11 }
```

- Questo file di build è sufficiente per supportare compilazione, testing, etc. per un progetto Java. Come è possibile?
 - ▶ Gradle sfrutta il principio **Convention over Configuration** (si veda il directory layout di default in slide successiva)
- Se importato in Eclipse come “Gradle Project” scarica automaticamente i jar necessari e li mette nel classpath!
- Visualizzazione grafo delle dipendenze: `$ gradle dependencies`



Directory layout di default in Gradle (Maven, sbt, ...)

Di default, Gradle si aspetta una **struttura di progetto in cartelle** diversa da quella di default in Eclipse

- Chi usasse Gradle, dovrebbe optare per questa convenzione, ed importare il progetto in Eclipse dopo averla applicata
- È la convenzione che si usa nei progetti “seri”
- Deriva da uno strumento di build automation antesignano di Gradle (Apache Maven), ed applicata da altri build tool (ad es. sbt)

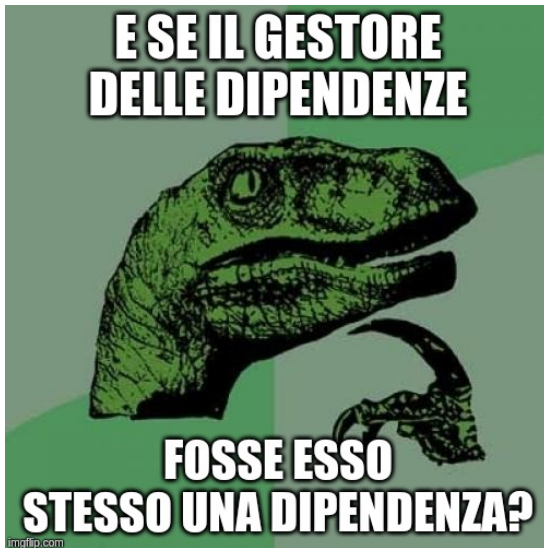
```
1 |-- src
2 |   |-- main
3 |   |   |-- java      Sorgenti java (ex: src)
4 |   |   \-- resources  Risorse del software (ex: res)
5 |   \-- test
6 |       |-- java      Sorgenti java per i test
7 |       \-- resources  Risorse per i test
8 |-- build.gradle.kts   Build file
9
```



Problema: ma se cambiassero il modo in cui Gradle risolve le dipendenze?

- Il sistema potrebbe ad un certo punto cambiare delle dipendenze transitive
- Il classpath cambierebbe
- Potenzialmente il nostro software potrebbe smettere di funzionare!
- (nota: è successo davvero...)
 - ▶ (...svariate volte)





Gradle wrapper III

Gradle fornisce una funzionalità detta **wrapper**

- Un piccolo script, contenente solo le informazioni necessarie a scaricare e usare la corretta versione di Gradle
- generabile da Gradle tramite il task wrapper
 - ▶ `gradle wrapper --gradle-version=X.Y.Z`
- dal momento in cui è presente il wrapper, è possibile usare quello per eseguire la corretta versione di Gradle!
 - ▶ `./gradlew nomeTask` — su Unix o emulatore bash come git bash
 - ▶ `gradlew.bat nomeTask` — su Windows cmd o Powershell
- **È sempre meglio usare il wrapper!**
- Ovviamente è possibile generare il wrapper usando il wrapper:
 - ▶ `./gradlew wrapper --gradle-version=X.Y.Z`
- Grazie al wrapper, è possibile eseguire Gradle anche se non si ha Gradle installato!



Gradle e DVCS/git: cosa tracciare?

- I descrittori della build e del progetto vanno tracciati
 - ▶ `build.gradle.kts`, `settings.gradle.kts`
- I file relativi al gradle wrapper vanno tracciati
 - ▶ `gradlew`, `gradlew.bat`, e tutta cartella `gradle/`
- I file generati e generabili attraverso i task gradle **NON** vanno tracciati
 - ▶ cartella `build/` va in `.gitignore`
- La cache interna di Gradle **NON** va tracciata
 - ▶ cartella `.gradle/` va in `.gitignore`
 - ▶ NB: `.gradle/` e `gradle/` sono cartelle diverse



- 1 DVCS Workflow
- 2 Rudimenti di Gradle (e strumenti di automazione della build)
 - Il problema della gestione delle dipendenze
 - Strumenti di automazione della build
 - Gradle in OOP
- 3 Preparazione al laboratorio



Uso raccomandato

- La costruzione e automazione di un processo di build **non** è parte della programmazione ad oggetti in quanto tale
- È un tema legato all'ingegneria del processo di produzione del software
- Molto sfaccettato, fra gli altri include aspetti che vedrete in futuro (ultimo anno di magistrale, opzionale “Laboratorio di Sistemi Software”)
 - ▶ Tecniche agili (e.g. Scrum)
 - ▶ Versioning
 - ▶ Build automation
 - ▶ Continuous integration
 - ▶ Continuous delivery
 - ▶ Deploy automation
 - ▶ ...
- La competenza con Gradle acquisibile da OOP è rudimentale: va oltre agli obiettivi di questo corso, ed è ripreso in corsi della magistrale

Consiglio: usate Gradle come semplice gestore di dipendenze
(lo sfrutteremo per la configurazione di progetti JavaFX)

- 1 DVCS Workflow
- 2 Rudimenti di Gradle (e strumenti di automazione della build)
 - Il problema della gestione delle dipendenze
 - Strumenti di automazione della build
 - Gradle in OOP
- 3 Preparazione al laboratorio



Modalità di lavoro

1. Forkare e clonare il repository fornito
2. Seguire le istruzioni nel file `README.md` nella root del repository
3. Leggere attentamente le istruzioni e risolvere l'esercizio in autonomia
 - ▶ Contattare il docente se si rimane bloccati
4. Utilizzare i metodi `main` e/o i test per verificare la correttezza delle soluzioni
5. Cercare di risolvere autonomamente eventuali piccoli problemi che possono verificarsi durante lo svolgimento degli esercizi
 - ▶ Contattare il docente se, anche dopo aver usato il **debugger**, non si è riusciti a risalire all'origine del problema
6. Effettuare almeno un commit ad esercizio completato
 - ▶ Fatene pure quanti ne volete durante l'esercizio stesso
7. **A esercizio ultimato sottoporre la soluzione al docente/tutor**
8. Proseguire con l'esercizio seguente

