1. Introduzione all'Ingegneria e all'Astrazione Orientata agli Oggetti

Questo capitolo introduce i concetti fondamentali che stanno alla base della progettazione e dello sviluppo del software moderno, con un focus particolare sull'approccio orientato agli oggetti. Verranno esplorate le differenze tra un semplice programma e un sistema software complesso, le fasi del processo di sviluppo, e l'importanza dell'astrazione e dell'incapsulamento per costruire software di qualità.

1.1 Obiettivo del Corso e Concetti Fondamentali del Software

1.1.1 Obiettivo del Corso

L'obiettivo principale di questo percorso di studi è insegnare tecniche moderne per la costruzione di sistemi software, utilizzando i concetti dei sistemi ad oggetti, che rappresentano l'approccio "mainstream" nell'industria (da `01-oo-abstraction_slides.pdf`, Pag. 3). Questo significa che ci si concentrerà su metodologie e strumenti che consentono di creare software robusto, manutenibile ed estendibile.

1.1.2 Programma vs. Sistema Software

È fondamentale distinguere tra un "programma" e un "sistema software", poiché rappresentano livelli di complessità e integrazione molto diversi:

Programma

- Insieme di istruzioni che automatizzano la soluzione di una classe di problemi.
- Spesso è associato a una visione algoritmica della computazione, dove un input viene trasformato in un output (Input→Output)
- Un esempio semplice potrebbe essere un algoritmo per calcolare la somma di due numeri.

Sistema Software

- Un sistema software è un'aggregazione di componenti di varia natura che cooperano per fornire una funzionalità computazionale
- Queste componenti possono includere programmi, librerie, parti del sistema operativo, basi di dati, interfacce grafiche, servizi Web, elementi di rete e dispositivi hardware
- La sua complessità è significativamente maggiore rispetto a un singolo programma.

Introduzione ai Sistemi di Controllo di Versione (VCS)

Questo documento ti guiderà attraverso i concetti fondamentali dei Sistemi di Controllo di Versione (VCS), spiegando perché sono essenziali nello sviluppo software e come funzionano, con un focus sul sistema distribuito Git.

<text/markdown id="vcs_intro" title="Introduzione ai Sistemi di Controllo di Versione (VCS)">

1.1 Motivazioni e Problemi nello Sviluppo Software

Lo sviluppo di software, sia individuale che in team, presenta sfide significative che i Sistemi di Controllo di Versione (VCS) mirano a risolvere.

1.1.1 Tracciare le Modifiche

La prima motivazione fondamentale per l'uso di un VCS è la necessità di **tenere traccia delle modifiche** apportate al codice e ai file di un progetto durante il suo sviluppo. Senza un sistema adeguato, si possono verificare problemi come:

- **Perdita di progressi**: Se non si salvano regolarmente le versioni, un errore o un problema tecnico può causare la perdita di ore di lavoro.
- **Difficoltà nel tornare indietro**: Potrebbe essere necessario ripristinare il progetto a una versione precedente funzionante per risolvere un bug o esplorare un'alternativa di design. Senza un VCS, questo processo è manuale, inefficiente e incline all'errore.
- **Mancanza di documentazione**: È difficile capire cosa è cambiato tra una versione e l'altra, chi ha fatto la modifica e perché.

Il "metodo classico" di tracciamento delle modifiche, che consiste nel creare copie di file o cartelle con nomi diversi (es. `progetto_finale_v1.zip`, `progetto_finale_v2_con_correzioni.zip`), è **inefficace** perché:

- È inefficiente in termini di risorse (spazio su disco).
- Richiede tempo e operazioni manuali, aumentando la probabilità di errori.
- Rende difficile l'analisi delle differenze e la selezione di modifiche specifiche.

1.1.2 Supportare la Collaborazione

La seconda motivazione cruciale è il **supporto alla collaborazione** tra più sviluppatori di software. Nello sviluppo di team, sorgono problemi come:

- **Lavoro concorrente**: Più persone devono lavorare contemporaneamente sugli stessi file o sezioni di codice.
- **Integrazione delle modifiche**: Le modifiche apportate da sviluppatori diversi devono essere combinate in un'unica versione coerente del progetto.
- **Risoluzione dei conflitti**: Quando due sviluppatori modificano la stessa parte di un file in modi diversi, si verifica un conflitto che deve essere risolto.

I "metodi classici" per la collaborazione, come il lavoro su uno schermo condiviso o l'assegnazione di "blocchi" di codice a singoli sviluppatori, sono altamente problematici:

- **"Uno schermo, molte teste"**: Inefficiente, poiché solo una persona può lavorare attivamente alla volta.
- **"Blocchi"**: Porta a un'alta probabilità di conflitti e rallenta lo sviluppo.
- **Condivisione in tempo reale (es. Google Docs)**: Accettabile per documenti di testo, ma "dirompente" per il codice, dove anche piccole incoerenze possono causare errori gravi.

1.2 Sistemi di Controllo di Versione (VCS)

I Sistemi di Controllo di Versione (VCS), anche noti come Sistemi di Gestione del Contenuto Sorgente (SGCS), sono strumenti progettati per affrontare le sfide sopra menzionate. Essi supportano lo sviluppo di progetti attraverso:

- Il **tracciamento della storia** del progetto.
- La possibilità di effettuare **rollback** (tornare a versioni precedenti).
- La raccolta di **meta-informazioni** sulle modifiche (autori, date, note).
- Il **merge** (unione) delle informazioni prodotte in diverse fasi o da diverse fonti.
- La **facilitazione di flussi di lavoro paralleli**.

1.2.1 Tipi di VCS: Locale, Centralizzato, Distribuito

Esistono tre tipi principali di Sistemi di Controllo di Versione, che si distinguono per come gestiscono l'archivio delle versioni del progetto:

1.2.1.1 VCS Locale

In un VCS locale, il database delle versioni è memorizzato direttamente sul computer dello sviluppatore. Questo è il metodo più semplice, ma offre limitazioni significative per la collaborazione e la sicurezza dei dati.

Vantaggi:

- Semplice da configurare e usare per progetti individuali.
- Accesso rapido alla cronologia delle versioni.

Svantaggi:

- Nessun supporto per la collaborazione in team.
- Rischio di perdita di dati in caso di guasto del computer locale.

1.2.1.2 VCS Centralizzato (CVCS)

Nei sistemi centralizzati, esiste una **copia di riferimento del repository su un server centrale** che contiene tutta la storia del progetto. Gli sviluppatori lavorano su un sottoinsieme di tale storia, scaricando le modifiche dal server e caricando le proprie. Esempi storici includono CVS e Subversion (SVN).

Vantaggi:

- Facile gestione dei permessi e controllo centralizzato.

Svantaggi:

- Semplice per i team, poiché tutti accedono a un'unica fonte di verità.

- **Punto singolo di fallimento**: Se il server centrale si guasta, il lavoro di tutti è bloccato e si rischia la perdita della cronologia.
- **Dipendenza dalla rete**: È necessario essere connessi al server per la maggior parte delle operazioni.
- **Meno flessibilità** per i flussi di lavoro paralleli complessi.

1.2.1.3 VCS Distribuito (DVCS)

Nei sistemi distribuiti, **ogni copia del repository contiene l'intera storia del progetto** (cioè, ogni sviluppatore ne ha una localmente). Questo significa che ogni clone del repository è un backup completo. Git e Mercurial sono esempi di DVCS.

Vantaggi:

- **Resilienza**: Nessun singolo punto di fallimento. Se un server si guasta, altre copie complete del repository esistono.
- **Lavoro offline**: Gli sviluppatori possono lavorare e committare modifiche localmente senza una connessione di rete.
- **Flessibilità nei flussi di lavoro**: Favorisce il lavoro parallelo e la creazione/fusione di rami (branch) in modo agevole.
- **Velocità**: Molte operazioni sono locali e quindi molto più rapide.

Svantaggi:

- Richiede una maggiore comprensione dei concetti di base (ad es. merge, rebase).
- La dimensione del repository può essere maggiore localmente, poiché include tutta la cronologia.

1.2.2 Breve Storia dei VCS

La storia dei VCS mostra un'evoluzione verso sistemi sempre più robusti e flessibili:

- **Concurrent Versioning System (CVS) (1986)**: Un sistema client-server centralizzato, operava su singoli file o a livello di repository. Utilizzava la compressione delta per risparmiare spazio.
- **Apache Subversion (SVN) (2000)**: Successore di CVS, ancora utilizzato. Mantiene un modello centralizzato ma migliora la gestione dei file binari e la concorrenza delle operazioni, sebbene rimanga macchinoso per flussi di lavoro paralleli.
- **Mercurial e Git (entrambi aprile 2005)**: Rappresentano la generazione dei DVCS. Sono stati sviluppati quasi contemporaneamente e condividono molti concetti, dimostrando l'efficacia del modello distribuito. Git, in particolare, è diventato il sistema di controllo di versione distribuito *de facto*.

1.3 L'Evoluzione della Cronologia di un Progetto

Intuitivamente, la cronologia di un progetto potrebbe sembrare una semplice linea retta, dove le modifiche si susseguono in ordine temporale. Tuttavia, nel mondo reale, le cose non vanno sempre come previsto.

1.3.1 Ripristino delle Modifiche (Rollback)

Spesso è necessario tornare indietro nel tempo a uno stato precedente in cui le cose funzionavano. Questo può accadere a causa di bug, decisioni di design errate o semplicemente per esplorare alternative. Quando si considerano i rollback, la cronologia del progetto non è più una linea, ma un **albero.**

1.3.2 Collaborazione: Divergenza e Riconciliazione (Merge)

Nel lavoro di squadra, gli sviluppatori spesso lavorano in parallelo su diverse parti del progetto. Questo porta a una **cronologia divergente**, dove le modifiche di un team non sono immediatamente presenti nel lavoro dell'altro.

La capacità di **riconciliare gli sviluppi divergenti** è fondamentale e viene solitamente chiamata **merge** (unione). Quando si ha la possibilità di unire sviluppi divergenti, la cronologia del progetto diventa un **grafo.**

Questo è possibile nei DVCS perché ogni copia del repository contiene l'intera storia, permettendo agli sviluppatori di unire il proprio lavoro in modo indipendente e poi sincronizzarsi con il repository centrale o con i repository degli altri.

1.4 Concetti e Terminologia dei DVCS (con focus su Git)

Git è il sistema di controllo di versione distribuito *de facto*, creato nel 2005 da Linus Torvalds per la gestione del kernel Linux. È noto per la sua velocità e per essere orientato a Unix. Comprendere la terminologia di base è essenziale per utilizzarlo efficacemente.

1.4.1 Repository

Il **Repository** è il cuore di un DVCS. Include l'intero contenuto e la cronologia del progetto, insieme a tutti i metadati. Questo comprende:

- Tutte le modifiche, con i loro autori, date e descrizioni.
- Informazioni su come ripristinare le modifiche.
- Differenze tra diversi punti nel tempo.

Di solito, è memorizzato in una cartella nascosta (es. `.git` per Git) nella cartella principale del progetto. La posizione di questa cartella segna la radice del repository.

1.4.2 Working Tree (Working Directory)

Il **Working Tree** (o `worktree`, o `working directory`) è l'insieme di **file che costituiscono il progetto**, escludendo i metadati del repository. È la **copia** dei file con cui lo sviluppatore sta lavorando attivamente. Quando si apportano modifiche ai file, queste avvengono nel working tree.

1.4.3 Commit

Un **Commit** rappresenta uno stato salvato del progetto. È una "istantanea" (snapshot) dello stato del working tree in un dato momento. Ogni commit:

- Raccoglie le modifiche necessarie per trasformare il commit precedente (genitore) nel commit corrente (tracciamento differenziale).
- Registra metadati importanti: il commit genitore (o più genitori in caso di merge), l'autore, la data, un messaggio che riassume le modifiche e un identificatore univoco (UID), che è un hash crittografico.
- Un commit senza genitori è un **commit iniziale**.
- Un commit con più genitori è un **commit di merge**.

1.4.4 Branch (Ramo)

Un **Branch** è una sequenza nominata di commit. I branch permettono agli sviluppatori di lavorare su nuove funzionalità o bugfix in isolamento dal codebase principale. Quando si crea un branch, esso punta allo stesso commit a cui fa riferimento il branch corrente. I nuovi commit fatti su quel branch lo fanno avanzare. Se nessun branch è stato creato al primo commit, viene utilizzato un nome predefinito (es. `master` o `main`).

1.4.5 Riferimenti ai Commit (Tree-ish)

Per navigare nella cronologia o cambiare branch, è necessario fare riferimento ai commit. In Git, un riferimento a un commit è chiamato `<tree-ish>`. I riferimenti validi includono:

- **Hash di commit**: Identificatori univoci (completi o abbreviati) di un commit (es. `b82f7567961ba13b1794566dde97dda1e501cf88` o `b82f7567`)
- **Nomi di branch**: Fanno riferimento all'ultimo commit di quel branch
- **HEAD**: Un nome speciale che si riferisce al commit corrente, ovvero il "punto" della cronologia su cui si sta lavorando. Quando si esegue un commit, `HEAD` si sposta in avanti verso il nuovo commit.
- **Nomi di tag**: Etichette permanenti per punti specifici nella cronologia (non discussi in dettaglio in questo PDF, ma menzionati come riferimenti validi)

È possibile anche creare **riferimenti relativi** utilizzando la tilde (`~`) seguita da un numero per indicare i genitori di un commit (es. `HEAD~1` per il genitore diretto, `HEAD~3` per il trisavolo) (da `06-git_slides.pdf`, Pag. 27, 58).

1.4.6 Checkout

Il **Checkout** è l'operazione di spostamento verso un altro commit o un altro branch. Questo aggiorna tutti i file tracciati nel working tree alla loro versione nel commit o branch specificato. Sposta `HEAD` verso il `<tree-ish>` di destinazione.

Un concetto importante è lo stato di **"detached HEAD"** (HEAD staccato). Questo si verifica quando si effettua il checkout di un commit vecchio che non è la "fine" di un branch. In questo stato, Git consente di effettuare commit, ma questi non sono associati a nessun branch esistente e rischiano di essere "persi" se non si crea un nuovo branch da quel punto (da `06-git_slides.pdf`, Pag. 62).

1.5 Introduzione a Java

Java è un linguaggio di programmazione fondamentale nel contesto dell'ingegneria del software moderna, in particolare per la sua aderenza al paradigma orientato agli oggetti.

1.5.1 Storia e Filosofia di Java

- **Origini**: Inventato da James Gosling alla Sun Microsystems nel 1995, Java è nato come una semplificazione di C++ pensata per sistemi embedded.
- **Filosofia "Write Once, Run Everywhere"**: Il principio cardine di Java è la sua **portabilità**. Il software scritto in Java può funzionare su diverse piattaforme (Windows, macOS, Linux, Solaris, architetture ARM, x86, PowerPC, amd64) senza bisogno di ricompilare il codice sorgente.
- **Diffusione Industriale**: Java è estremamente diffuso nell'industria del software, rendendolo un riferimento chiave per la programmazione e la progettazione. Le linee guida progettuali apprese con Java sono spesso applicabili anche ad altri linguaggi come C#, C++, Ruby, Python, Kotlin, Scala.

1.5.2 Strumenti Java (JDK, JVM)

L'ecosistema Java si basa su due componenti principali:

- **Java Development Kit (JDK)**: È un insieme di strumenti essenziali per gli sviluppatori Java. Include:
  - Il **compilatore** Java (`javac`)
  - La **Java Virtual Machine** (JVM)
  - Le **librerie** standard (Java Class Library - JCL)
- **Java Virtual Machine (JVM)**: Un programma che **carica il bytecode** delle classi Java e lo **esegue**. La JVM è fondamentale per l'**indipendenza dall'hardware** e fornisce servizi aggiuntivi come la gestione della memoria (Garbage Collector)

Il processo di sviluppo tipico in Java prevede:

1. **Compilazione**: Il codice sorgente Java (`.java`) viene **compilato** dal `javac` in **bytecode** (`.class`).
2. **Esecuzione**: Il bytecode viene **interpretato** ed **eseguito** dalla JVM (da `01-oo-abstraction_slides.pdf`, Pag. 31-32).

(da `01-oo-abstraction_slides.pdf`, Pag. 32)

Java è un linguaggio ibrido (tra linguaggi compilati e interpretati)

Java è un ottimo esempio di un approccio ibrido che cerca di combinare i vantaggi di entrambi:

1. Il codice sorgente Java (`.java`) viene **compilato** dal compilatore Java in un formato intermedio chiamato **bytecode** (`.class`). Questo bytecode non è codice macchina nativo, ma un codice ottimizzato per essere eseguito su una "macchina virtuale" (la JVM - Java Virtual Machine).
2. Il **bytecode** viene poi **interpretato** dalla JVM sulla macchina di destinazione. La JVM, a sua volta, può utilizzare un compilatore Just-In-Time (JIT) che compila "al volo" le parti più utilizzate del bytecode in codice macchina nativo per migliorare le prestazioni.

1.5.3 Sviluppo Storico di Java

Java ha avuto un'evoluzione continua, con rilasci regolari che hanno introdotto nuove funzionalità e miglioramenti:

- **Versioni Iniziali (Java 1.0 - 1.4)**: Le prime versioni hanno stabilito le basi del linguaggio, con l'aggiunta di framework grafici come Swing.
- **Java 5 (1.5) (2004)**: Introduzione di generici, inner class e annotazioni. Da questa versione, si è deciso di non usare più "2.x" nel nome, passando direttamente a "5"
- **Java 8 (2014)**: Aggiunta delle espressioni lambda e degli stream, che hanno introdotto elementi di programmazione funzionale.

- **Java 9 (2017)**: Introduzione del Java Module System (Jigsaw), che ha migliorato la modularità e la gestione delle dipendenze. Da questa versione, si è passati a un ciclo di rilascio di 6 mesi
- **Versioni LTS (Long Term Support)**: Ogni due anni circa viene rilasciata una versione LTS, che riceve supporto a lungo termine ed è consigliata per la produzione (es. Java 11, Java 17, Java 21)

1.5.4 Modello di Sviluppo Attuale

Attualmente, OpenJDK produce l'implementazione di riferimento del JDK, ma esistono anche altre implementazioni (es. Eclipse OpenJ9, Amazon Corretto, GraalVM). Il ciclo di rilascio di 6 mesi garantisce un'evoluzione rapida del linguaggio e delle sue funzionalità.