Concurrency in Java

Concurrent Software Systems in Java

- ***Concurrent***: computazione ottenuta componendo computazioni sequenziali che sono eseguite indipendentemente, che possono essere eseguite virtualmente in parallelo
- La computazione di un sistema concorrente è strutturata in termini di un insieme [anche dinamico] di flussi di esecuzione
- I flussi di esecuzione sono composti in parallelo sotto vincoli di sequenzializzazione
    - I Linguaggi che permettono di esprimere almeno alcuni di questi vincoli sono detti linguaggi concorrenti
- I flussi di esecuzione possono anche condividere dati e risorse

Java

- Permette di esprimere esprimere alcuni vincoli base dalla sua creazione
- Dalla versione 5.0 ci sono anche delle librerie ad alto livello per risolvere problemi di concorrenza
- I pattern di sequenzializzazione complessi possono essere espressi combinando l'uso dei costrutti base del linguaggio

Processi Concorrenti

- è un programma in esecuzione
    - Un processo è un entità dinamica che esegue un programma usando un insieme particolare di dati e risorse

    - 2 o più processi possono eseguire lo stesso programma ognuno con i loro dati e risorse
- Componenti di un processo
    1. Il programma da essere eseguito
    2. I dati su cui il programma è eseguito
    3. Le risorse necessitate dal processo a tempo di esecuzione
    4. Lo status del processo in esecuzione
- Un processo è eseguito in un ***ambiente di macchina astratta*** che gestisce la condivisione e isolamento dei dati e risorse di una comunità di processi
- Java Virtual Machine [JVM] — offre l'ambiente descritto sopra

Thread

- Un thread è un Flusso di esecuzione di un processo
- Ogni thread è sempre associato con un singolo processo
- Più thread possono essere associati ad un singolo processo (concorrenza)
    - Un processo sequenziale ha un solo thread
- L'associazione avviene dinamicamente perché i thread vengono inizializzati e terminati dinamicamente
- Un thread alloca un parte di risorse per la sua esecuzione
- Un thread può anche avere dati privati e uno stato

Note

- La memoria condivisa è visibile a tutti i thread di un processo
- I thread sono certe volte chiamati lightweight processes perché l'overhead associato alla loro computazione è ridotto rispetto a quello dei processi

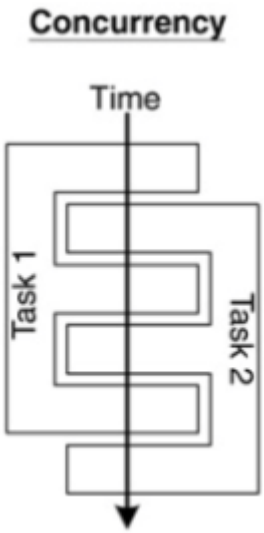
Proprietà dei thread:

- Un thread inizia la sua esecuzione ad un punto preciso nel programma (entry point)

    - il main thread è eseguito all'inizio e parte dall'entry point dell'applicazione [main method]
    - L'entry point è deciso a compile time o a runtime
- Un thread viene eseguito in un ordine predefinito [esegue il suo codice sequenzialmente]
- Un thread viene eseguito indipendentemente dagli altri
- I thread sembra che siano eseguiti in parallelo anche se possono essere interleaved [concurrency]
- Entry point dei thread: `run()`
- Un thread termina quando il metodo `run()` termina.
- Un thread non può essere terminato forzatamente.

Concorrenza e Parallelismo

- Sistemi paralleli: sono allocati su un insieme di CPU che eseguono diversi processi e i loro thread in parallelo
    - Memoria condivisa tra CPU è normalmente presente
    - Se non c'è memoria condivisa e la comunicazione utilizza una rete → Il sistema si chiama distribuito
- Sistemi Concorrenti: possono essere allocati su una singola CPU

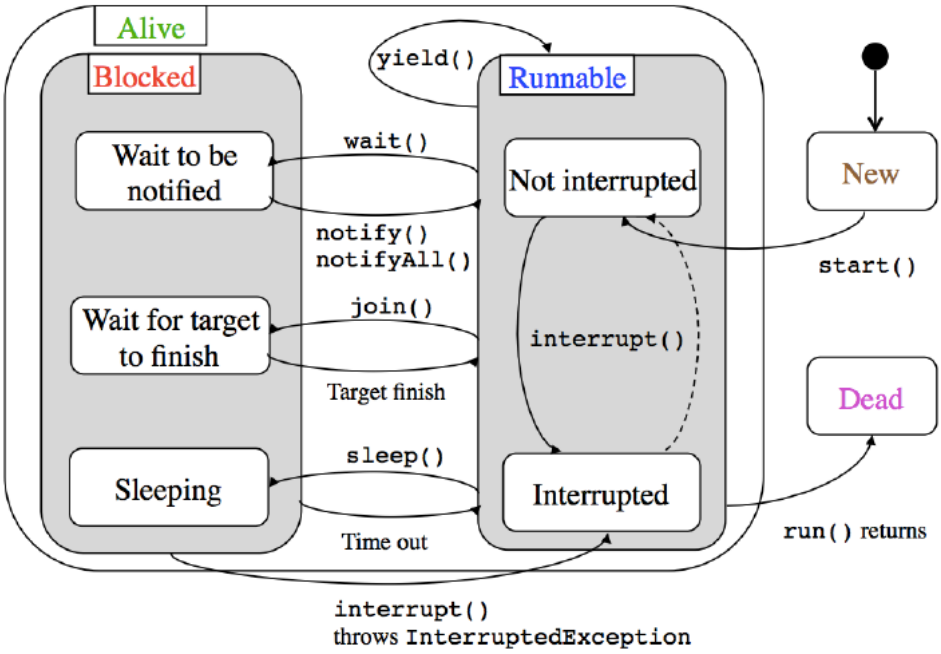


Thread in Java

- 2 Modi di descriverli:
  - Estendendo la classe `java.lang.Thread`
  - Implementando l'interfaccia `java.lang.Runnable`
    - usando: una classe di alto livello, una classe interna anonima, lambda expression, method references (`::`)
- Ogni thread di esecuzione è associato con un oggetto chiamato *thread object* *[istanza di Thread]*

```
public class MyThread extends Thread {
  @Override
  public void run() {
    // il corpo del thread
  }
}
```

- Il thread è rimosso automaticamente quando il flusso di esecuzione finisce
  - *Nella versione 1 di Java si poteva distruggere un thread esternamente*
  - é stato rimosso perché potrebbe causare la non-liberazione delle risorse usate dal thread
- il metodo `start()` è usato per creare e far partire il nuovo thread che esegue `run()`
  - è chiamato sull'oggetto thread
  - Estendento Thread: `new MyThread().start();`
  - Implementando Runnable: `new Thread(new MyRunnable()).start();`
- I thread in Java hanno uno *stato di lifecycle* strutturato su 3 livelli:
  - Alive (che si suddivide in Blocked e Runnable);
  - New;
  - Dead;



- Metodi di Thread:
  - `sleep(millis)`
  - `interrupt()`
  - `interrupted()` → Controlla se il thread è stato precedentemente interrotto
  - `yield()` : Metodo che mantiene il thread nello stato running, ma lasciando il thread scheduler avviare altri thread
  - `join()` : forza il thread corrente nello stato bloccato finché non finisce il thread su cui è chiamato join
- Su una singola CPU, i thread eseguono uno alla volta in modo da dare l'illusione del parallelismo
- La JVM implementa un semplice scheduling algorithm per i thread chiamato *fixed priority scheduling*
  - La JVM esegue il thread con la priorità più alta
  - Se 2 thread hanno la stessa priorità vengono eseguiti in round-robin

Java Memory Model

- I thread di un programma java hanno:

    - Stack privato (che supporta invocazioni di metodi ❓)
    - Storage locale per thread [Mappa chiave valore / poco usata]
    - Shared Object Heap [tutti gli oggetti nell'heap di accesso comune]
- Questo modello descrive come il contenuto delle memorie è storato nella gerarchia di memoria del sistema
- Thread di un programma Java non è necessariamente eseguito sulla stessa CPU perché la JVM normalmente, è distribuita su diverse CPU
- Gestione di cache e registri delegati alla JVM
    - C'è da tenere conto che visto il punto precedente → Potrebbero esserci thread diversi che hanno diversi contenuti nelle cache e quindi che ottengono accesso più veloce, o problemi di coerenza di aggiornamento dei dati, quando un aggiornamento dovrebbe ripercuotersi sull'heap
- esecuzione di thread su multiple CPUs causa rilevanti problemi di *memory coherence*!

Mutua Esclusione in Java

- Uno dei problemi più semplici della programmazione concorrente
- Programma A: Problema di mutua esclusione se, dato un insieme M di sezioni mutualmente esclusive
    - Solo un thread alla volta può accedere ed eseguire una delle sezioni del programma in M
    - I thread che non possono eseguire le sezioni in M sono bloccati e riesumati appena possibile
- Una sezione critica di un programma è associata con un MUTual EXclusion device (mutex) per controllare l'accesso ad una sezione critica
    - Mutex: può essere ottenuta e rilasciata dai thread ed è di proprietà del thread dopo che è stata ottenuta e prima del suo rilascio
    - I thread sono forzati in uno stato bloccato se non riescono ad ottenere la mutex, e sono forzati ad eseguire dopo che la ottengono
    - un thread può eseguire la sezione critica solo quando ha la mutex relativa alla sezione critica che vuole eseguire

- Nota: le acquisizioni e rilasciamenti sono:
  - **Rientranti** perché la mutex può essere ottenuta e rilasciata in loops innestati diverse volte senza bloccare il thread che ha la mutex
  - **Operazioni di sincronizzazione** all'interno del processo perché i loro effetti sono condivisi tra tutti i thread del processo [e possibilmente più CPU]
- Una sezione critica è identificata da:
  - Il modificatore `synchronized` dei metodi per dire che il corpo è una sezione critica e che il riferimento `this` è usato come mutex
  - Il blocco `synchronized`, per dire che il corpo del blocco è la sezione critica, e che l'oggetto referenziato nella testa del blocco è usato come mutex
- Le sezioni critiche assicurano una soluzione per:
  - Thread indifference: problema che occorre quando due operazioni che girano su diversi thread agiscono sugli stessi dati e sono in parallelo. Questo causa una race condition.
  - Coerenza di memoria: problema che occorre quando diversi thread hanno visioni diverse e inconsistenti dello stesso dato tra più CPU a causa di cache non allineate

Happens-Before

Per sfruttare bene i registri a nostra disposizione il compilatore può decidere di modificare la sequenza di esecuzione degli statement del codice in modo da sfruttarli meglio e quindi cambiare l'ordine di esecuzione [mantenendo la semantica]

Java garantisce che se c'è una relazione di tipo **happens-before** nel sistema tutta la catena rispetterà questa relazione

Esempio: Se l'operazione `write` *Happens-Before* l'operazione `read`, allora il valore sarà sicuramente quello rettificato da write e quindi aggiornato e visibile a tutti i thread

Ogni azione in un thread *happens-before* di ogni azione che in quello stesso thread avviene dopo nel programma

Un rilascio di un mutex *happens-before* qualsiasi lock dello stesso mutex

Una scrittura ad un campo `volatile` *happens-before* ogni lettura successiva di quello stesso campo ⇒ Risolve problemi di incoerenza di memoria SOLO quando non ci

sono problemi di interferenza di altri thread, quindi garantisce che l'accesso in lettura avvenga solo dopo che le cache siano rinfrescate sia del campo che del contesto di esecuzione
(*Nota*: il modificatore `volatile` non garantisce mutua esclusione in accesso ad un campo)

Una chiamata a `start()` ad un thread *happens-before* qualsiasi altra azione nel thread eseguito

Tutte le azioni di un thread *happen-before* qualsiasi altra `return` delle thread o chiamata a `join`

La keyword `synchronized` può essere usata per risolvere problemi di coerenza

L'uso di `synchronized` su un oggetto *obj* assicura che tutti i cambiamenti a *obj* siano propagati a tutti i thread interessati prima di qualsiasi accesso sincronizzato a obj

Nota: I campi `final` non hanno problemi di sincronizzazione in quanto sono inizializzati a compile-time e significa che si possono leggere sempre.

Operazioni Atomiche:

- In java, le operazioni read e write non possono essere interrotte per sospendere un thread e avviarne un altro
- In dettaglio: read e write sono atomiche per i riferimenti e per la maggior parte delle primitive tranne `long` e `double`, read e write sono atomiche per tutte le variabili e i campi `volatile`
- Nota: le operazioni atomiche non soffrono di problemi di interferenza di thread ma possono comunque essere afflitti da problemi di consistenza di memoria se il modificatore `volatile` non è usato

Waiting e Notifying Events in Java

- Il problema di Attesa (`wait`) e notifica (`notify`, `notifyAll`) di eventi è un altro dei problemi più semplici di concorrenza.
- Problema degli eventi: thread che aspettano che altri thread completino un evento e quest'ultimi che notifichino quando l'evento è avvenuto
- Busy waiting: tecnica di polling che richiede sempre se è passato un evento che permette al thread di partire [da non usare perché inefficiente e non fa uso di

strumenti forniti dalla JVM]

- Metodi:
  - `notify()` e `notifyAll()`
    - Utilizzabile solo se è chiamato sull'oggetto guardia della sezione critica mentre si è in quella stessa sezione critica
    - Negli altri casi viene lanciata un eccezione
    - Quando viene chiamato, i thread in wait si sbloccano, non appena la sezione critica finisce dopo aver finito di eseguirla
    - `notifyAll`: sveglia tutti i thread in wait, anche se solo uno può eventualmente procedere all'interno della sezione critica
    - `notify`: sveglia solo uno dei thread che sono in wait per la sezione critica — è più efficiente ma è anche più error-prone (nel caso uno dei thread nella catena si dimentichi o non riesca ad eseguire `notify` alcuni thread rimangono bloccati)
  - `wait()`
    - Può essere chiamato solo sull'oggetto guardia della sezione critica mentre si è in quella stessa sezione critica
    - Esce dalla sezione critica e si mette in attesa di una notify
    - `wait`: lancia InterruptedException se il thread in wait viene interrotto dall'esterno [`interrupt`]
    - `wait(millis)`: metodo per aspettare non più di una certa deadline [best effor deadline]
  - `Thread#interrupt()`
    - Quando si chiama interrupt non si esce dalla sezione critica istantaneamente
    - Quando la sezione critica viene sbloccata in modo anomalo in questo modo (senza una `notify`)
    - viene lanciata un eccezione `InterruptedException`

High-Level Abstractions for Concurrency

Separarsi dalla macchina per avere astrazioni in modo che si evitino i problemi di notify e wait vicino alla macchina poiché diventa molto complicato da fare management di questi problemi quando il sistema è grande (far si che l'utente non si debba preoccupare di gestire synchronized, lock, ecc.. perchè tutto gentito nelle classi utilizzate).

- Java da questi meccanismi nel package: `java.util.concurrent`
- Bisogna prima aver capito in profondo il concetto prima di usare le classi di libreria

Vantaggi:

- Maggiore manutenibilità dei sistemi concorrenti
- Migliorare la riutilizzabilità di soluzioni a problemi di concorrenza
- Migliorare le caratteristiche delle soluzioni ai problemi di concorrenza [risoluzione più semplice]

Blocking Queues

- Sequenza di elementi che cambia dinamicamente according alla policy FIFO
- Basic operations:
  - *Creazione*: creare una coda vuota
  - *Distruzione*: per distruggere una coda
  - *Is Empty Test*
  - *Is Full Test*
  - *Enqueue*: aggiunge un oggetto alla coda
  - *Dequeue*: rimuove un oggetto dalla coda

- Una coda bloccante è intesa per uso concorrente.
- Le operazioni sono bloccanti se non possono essere eseguite immediatamente:
  - Enqueue quando la coda è piena
  - Dequeue quando la coda è vuota

Locks and conditions

- Il lock (esplicito) è utilizzato come astrazione per fare mutua esclusione in modo più libero, in modo che non debba per forza essere una sezione di codice contigua [`synchronized`]
- Sono simili alle mutex dei pthread → Vengono bloccati (acquired) e sbloccati (released) in modo manuale
- Usare i lock espliciti va fatto con cautela perché introduce molti problemi in più rispetto alle sezioni critiche normali (`synchronized`)

Conditions

- Quando andiamo ad aggiungere le condition allora è più sensato usare il lock
- Per ogni lock si crea almeno una condition:
  - Una condizione è un astrazione usata per aspettare eventi interessanti e segnalare questi eventi, un thread può:
    - segnalare che la condizione è diventata vera
    - aspettare bloccando l'esecuzione aspettando che la condizione venga segnalata

Atomic References

Strumento che permette di fare letture sull'atomic reference e scritture che verranno rese atomiche rispetto a altre letture e scritture.

Struttura della classe: uni/AtomicReference.java at master · Davoleo/uni (github.com)

Pool of Resources

- I problemi di concorrenza sono causati spesso da risorse condivise ⇒ Importante la corretta gestione delle risorse condivise
- Un gruppo di risorse identiche si chiama pool di risorse
  - Risorse acquisite e rilasciate nel momento in cui viene creata la piscina
  - Risorse assegnate su richiesta
- le pool di risorse sono utilizzate per controllare la quantità di risorse necessitate da un sistema concorrente in modo da assicurare che ci siano una quantità sufficiente di risorse disponibili e che siano utilizzate efficientemente

Thread Pool

- I thread sono virtualmente le risorse più importanti di un sistema concorrente
- La costruzione, distruzione, accesso ai thread è controllato da thread pools semplici
- Nel caso della `SimpleFixedThreadPool`: Quando la pool è creata, *essa crea e avvia tutti i thread*, per assicurare che siano tutti subito disponibili e che il livello di concorrenza sia controllato
- Esistono anche thread pool complesse [classe `Executors`]

Executors

Astrazione (interfaccia) per eseguire task concorrenti:

- E' associato con una thread pool per eseguire le task [implementation]
- Accoda le task che non possono essere eseguite subito
- Offre un modo per ritornare il valore risultato delle task e anche le eventuali eccezioni [`Callback` & `Future`]
- Permette interruzione graceful delle task di tutti i thread

- (Certe Volte) offre policy di scheduling
- `ExecutorService`: estende `Executor` aggiungendo qualche metodo importante
- Classe `Executors`
  - offre in disponibilità statica diversi servizi relativi alla concorrenza
  - `Facade`: Classe che si presenta come punto di accesso principale per un sottosistema [Structural pattern: Funziona come control panel per un sottosistema]
  - In questo caso permette di astrarre la creazione degli oggetti e quindi è una classe chiamata `Abstract Factory`
  - `newFixedThreadPool(int nThreads)`: blocco di thread preallocati di numero fisso nThread
- Metodi importanti:
  - `execute(Runnable command)`: Esecuzione di un comando in un qualche momento nel futuro
  - `shutdown()`: chiude il servizio → non aspetta che terminino i thread [chiamata non bloccante], ma blocca l'aggiunta di nuove task lanciando l'eccezione `RejectedExecutionException`

3 Modi per eseguire task

1. *One Way Execution:* Executor non offre un modo per sapere se esiste tantomeno accedere al valore risultato della task eseguita
2. *Execution with Callback*: Per cui l'esecutore permette ad una callback task di utilizzare il risultato della task [di solito nel thread che ha eseguito la task richiesta]
3. *Execution with Future*: l'esecutore offre un futuro [una promessa] che gestisce il risultato della task [se esiste] non appena diventa disponibile

Futures

- Implementati come oggetti
- Gestiscono il risultato di una task asincrona
- Gestiscono Le eccezioni che hanno causato la terminazione della task asincrona

- Un Future blocca il thread che richiede di accedere al suo valore incorporato se esso non è ancora disponibile
  - Il thread è riesumato non appena il risultato è disponibile
  - Il thread è riesumato anche non appena un eccezione è disponibile e lanciata
- Nota: I future non sono bloccanti se il valore è già disponibile quando richiesto

Future Pools: Per evitare di bloccarsi ad ogni richiesta del valore quando si hanno multipli Future da gestire, permette di aspettare tutti i future e leggere quanti ne pare aspettando in modo concorrente e non sequenziale. (Promise.all)

Java Reflection

- la JVM da modi di postporre certe decisioni a runtime tramite il package `java.lang.reflect`
- Java, pur rimanendo tipizzato staticamente offre metodi object-oriented per fare:
  - Dynamic linking di classi
  - Introspezione [di oggetti]
  - Creazione dinamica di oggetti
  - accesso dinamico ai campi
  - Invocazione dinamica dei metodi
- Factory class*: classe che è associata ad ogni oggetto che crea, in particolare, Il factory design model dice di definire un'interfaccia (Un'interfaccia java o una classe astratta) per la creazione di oggetti e lasciare che le sottoclassi decidano quale classe istanziare.
- Ogni classe viene rappresentata a runtime da un oggetto chiamato class object [o descriptor]
  - la JVM offre un oggetto di classe `java.lang.reflect.Class<C>`
  - Data una classe o interfaccia C l'oggetto che rappresenta la classe o interfaccia è riferito con C.class
- Ogni oggetto classe è associato con un class loader, l'oggetto usato per caricare il bytecode della classe
  - Il class loader è molto importante perché è ciò che può essere modificato per cambiare ciò che viene caricato e come viene caricato
- Class objects sono il mattone base della Java reflection

*Esempio Di factory class o factory method (design pattern):

```
public interface Notification {
    void notifyUser();
}

public class SMSNotification implements Notification {
    @Override
    public void notifyUser(){
        System.out.println("Sending an SMS notification");
    }
}

public class EmailNotification implements Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending an e-mail notification");
    }
}

public class PushNotification implements Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending a push notification");
    }
}

public class NotificationFactory {
    public Notification createNotification(String channel) {
        if (channel == null || channel.isEmpty())
            return null;

        switch (channel) {
        case "SMS":
            return new SMSNotification();
        case "EMAIL":
            return new EmailNotification();
        case "PUSH":
            return new PushNotification();
        default:
            throw new IllegalArgumentException("Unknown channel "+channel);
        }
    }
}
```

Class Objects

- Dato un oggetto `o`, `o.getClass()` ritorna la classe oggetto associata con `o`
- Data una stringa `n` contenente il fully qualified name di una classe: `Class.forName(n)` ritorna l'oggetto classe

    - Il classloader utilizzato è quello usato per caricare la classe del metodo che sta eseguendo il forName
- Dato un classloader `l` e una stringa `n` contenente il fully qualified name di una classe, `l.loadClass(n)` ritorna l'oggetto classe
    - [se la classe è già caricata la ritorna subito, altrimenti la carica in memoria e ritorna]
- Se il FQName non è trovato è lanciata una `ClassNotFoundException`
- Tramite il descriptor si possono fare a runtime certe azioni che normalmente si fanno nel codice sorgente:
    - `c.cast(o)` equivalente a `(C)o`
    - `c.isInstance(o)` equivalente a `c instanceof C`
    - `c.isAssignableFrom(k)`: controlla se la classe riferita con `c` è la stessa o una superclasse della classe rappresentata da `k`
    - more methods…
-

Introspection:

Gli oggetti descrittori di classi rendono possibile l'introspezione

- è possibile listare i campi visibili di C
- è possibile elencarei descrittori dei costruttori visibili in C
- è possibile elencare i descrittori dei metodi visibili in C
- è possibile ottenere un riferimento al class object delle superclassi o delle interfacce implementate

Dynamic Object Creation

- Data una classe `Class<C>` è possibile creare oggetti dinamicamente tramite:
    - `c.newInstance()`
    - usare c per accedere ad uno dei descrittori dei costruttori di c e invocandolo

- Se C **non** è un tipo generico o ? allora non è necessario fare un downcast se a sinistra si sta assegnando ad una variabile for example

Dynamic access to fields

- Accesso ai descrittori dei campi visibili di c → e si possono anche prendere i valori tramite
  - `field.get(o)` | dove o è l'oggetto di cui ottenere il valore del campo
  - `field.set(o, v)`
  - Nota: `Field` non è generico

Dynamic Method Invocations

- Data una classe c è possibile accedere ai descrittori di metodi
- Si possono anche invocare dinamicamente tramite `method.invoke(o, a...)` | dove `o` è l'oggetto su cui è invocato il metodo e `a` sono gli argomenti passati alla chiamata
- Nota: `Method` non è generico

Aspect Oriented Programming

AOP (Aspect-Oriented Programming): advocated come lo step successivo alla OOP (Object-Oriented Program) dagli anni 2000 [per promuovere il riuso].

In un immagine semplice, l'AOP riguarda aggiungere *aspetti* [aggettivi] alla OOP e aggiunte agli oggetti per aggiungere funzionalità senza cambiare la classe.

Un *aspect provider* è un oggetto che può essere attaccato staccato da un altro oggetto e che fornisce certe feature.

In questa visione particolare, l'interesse della AOP è dare implementazioni di aspetti che:

- Sono (Mostly) indipendenti dalle caratteristiche dell'oggetto a cui sono attaccate;
- Sono Componibili in modo libero in modo che un oggetto possa essere attaccato a diversi aspetti e quindi ottenendo diverse features;
- Sono ortogonali, cosicché la composizione di aspetti da una somma delle feature indipendenti (esempio possiamo sommare la SharedAspect con la LoggingAspect);

Questa specifica incarnazione di AOP può essere ottenuta in java tramite i *dynamic proxies.*

- Dato un oggetto 'o', un aspect provider attacca l'aspetto ad 'o' tramite un proxy dinamico che intercetta tutte le invocazioni e metodi pubblici di 'o'

Aspetti general-purpose considerati normalmente:

- **Shared Object**: Assicura la mutua esclusione per l'esecuzione dei suoi metodi;
- **Logging Object**: Oggetto che traccia le invocazioni dei metodi in un message log;
- **Persistent Object**: Oggetto che sopravvive allo shutdown del sistema nel quale è stato creato/modificato;

- **Active Object:** Un oggetto che esegue i suoi metodi in una thread pool dedicata;
- **Remote Object**: Accetta invocazioni di metodi da client remoti;
- **Reloadable Aspect:** permette di ricaricare il codice di una classe modificata a runtime. [sfruttando il classloader]

Shared Aspect

- Oggetto condiviso: Mutua esclusione per l'esecuzione dei suoi metodi (solo i metodi delle interfacce implementate).
- Tramite un proxy dinamico che intercetta le chiamate e le "wrappa" in una sezione critica.

Logging Aspect

- Oggetto che traccia le invocazioni a suoi metodi in un message log.
- Solo per metodi implementati sotto interfacce.
- Un proxy dinamico è sufficiente per intercettare tutte le invocazioni dei metodi.
  - l'invocationHandler fa il logging prima e dopo l'invocazione del target method.
  - Invocation Handler logga anche in caso di eccezioni.

Persistent Aspect

- Oggetto Persistente: Sopravvive allo shutdown del sistema.
- I proxy dinamici non sono necessari perché l'utente richiede esplicitamente.
  - il commit dei cambiamenti;
  - Rollback dei cambiamenti;
- Persistenza semplice può essere ottenuta caricando/salvando oggetti serializzabili su file.
  - Oggetti che implementano `java.io.Serializable` possono essere facilmente serializzati con `java.io.ObjectOutputStream` e deserializzati con `java.io.ObjectInputStream`.

- Gli oggetti serializzabili offrono un campo privato `serialVersionUID` per disambiguare versioni diverse della classe.

Dynamic Proxies

- Dato un array di oggetti di una classe associati con interfacce un proxy dinamico è un oggetto che implementa le interfacce dell'array e invoca user code.

Active Aspect

- Un oggetto attivo è un oggetto che esegue i suoi metodi in una thread pool dedicata [non necessariamente il chiamante].
- Solo i metodi dalle interfacce implementate sono interessanti perché sono i metodi esportati.
- Un proxy dinamico è sufficiente per intercettare tutte le invocazione di metodi interessanti.
- Bisogna creare un interfaccia attiva con signature simili ma includendo `Future<T>` e `Callbacks<T>`.
- Normalmente l'interfaccia attiva A dell'interfaccia T è richiesta di estendere: `Active<T>`.

Remote Aspect

- Simile abbastanza alle Remote Procedure Calls [sono stateless di solito] in questo caso degli oggetti remoti potrebbe anche esserci dello stato salvato in remoto.
- Il canale di comunicazione più semplice per gli oggetti remoti sono socket [TCP/IP] associati a interazioni singole.
- Se la classe remota è di una versione diversa allora viene mandata anche quella tramite la socket.
- Un oggetto server crea una socket per accettare richieste di connessione utilizzando `java.net.ServerSocket`.
  - TCP port passata al costruttore;

    - il metodo `accept()` è utilizzato per aspettare e ottenere un oggetto di classe `java.net.Socket` per leggere e scrivere dal peer connesso;
- Normalmente un server organizza le proprie connessioni in thread diversi [di una pool].
- Un client crea una socket per chiedere una connessione al server usando `java.net.Socket`.
    - Hostname e port nel costruttore.
- Remote Aspect provider:
    - Registra un oggetto come un server su una porta specifica;
    - Offre un proxy per mandare richieste al server e ricevere risposte corrispondenti;
- sia l'interfaccia sul server sia il dynamic proxy sul client implementano T.
- Nota che il proxy offre chiamate sincrone ai metodi del server [il client aspetta finché la risposta di una richiesta è disponibile].
- Le async method calls possono essere ottenute facilmente attaccando l'aspetto attivo al proxy.

Reloadable (Class) Aspect

Le istanze di una classe reloadable sono oggetti reloadable anche se gli oggetti singolo non sono ricaricati.

- i class loader sono usati per:
    - Caricare il bytecode delle classi nella memoria;
    - Creare descrittori di classi e metterle disponibili alla JVM;
    - **Non possono "unloaddare" le classi;**
    - Le classi caricate da diversi class loader sono tra di loro diverse, anche se i fully qualified name sono uguali;
- é possibile fare Hot Swap di classi.
    - Una nuova classe è necessaria per ogni swap;
    - Le dipendenze della classe devono anch'esse essere swappate;

ClassLoader

I `ClassLoader` sono abbastanza vecchi [una delle prime cose introdotte in Java], poiché caricare e scaricare classi dinamicamente era uno dei requisiti base del linguaggio.

Bytecode viene caricato → Descrittore di classe in Memoria [classe non ancora inizializzabile] → Class Resolution [rende la classe istanziabile e fa calcolo di frequenza di utilizzo metodi per compilazione in codice nativo].

BootstrapClassLoader: Primo classloader, unico che non ha il padre.

SystemClassLoader: ClassLoader che carica le nostre classi, discendente del `BootstrapClassloader`.

Per le nostre classi: ogni classe è associato ad un descrittore di classe e ogni descrittore è associato ad un class loader che l'ha caricato, e quindi nel caso non si abbia un ClassLoader personalizzato da utilizzare si può usare quello che ha caricato la classe stessa come default.

Procedura di caricamento dinamico: Realizzare un classloader dedicato al caricamento delle classi che viene utilizzato ogni volta che ci sono dei cambiamenti, e quando i vecchi oggetti diventano inaccessibili, i quali puntano al class descriptor che diventa inaccessibile e quindi viene eliminato e a sua volta punta al vecchio ClassLoader che se non è accessibile da nessun descrittore viene eliminato.

codebase notes:

- è `abstract` quindi non la puoi usare per costruire oggetti, senza fare sottoclassi.
- Endpoints
  - `public Class<?> loadClass(String name) throws ClassNotFoundException`;
  - `public Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException`;
    1. Verifica se la classe è già stata caricata [anche nel classloader parent], nel caso viene ritornato direttamente il descrittore delle classe già caricata;
    2. Invoca la loadclass sul parent, se il parent è null viene chiamata sul classloader built-in della JVM nel caso;
    3. Invoca `findClass(String)` per trovare la classe;

Test-Driven Development

Costo relativo al software di solito è la parte predominante del costo del sistema. Tuttavia la parte più significativa del costo del software è associata alla manutenzione più che allo sviluppo.

Uno degli obbiettivi dell'ingegneria del software è di ridurre i costi di manutenzione tramite riuso strutturato di software framework e libraries.

Costo dei sistemi software

2 Categorie di costo:

1. Costi Diretti: Imputabili direttamente al prodotto che si utilizza (Costo degli sviluppatori e degli strumenti di sviluppo + framework, Infrastrutture in cloud, …)
2. Costi indiretti: Associate con le attività di supporto per le attività del sistema (costo di consulti amministrativi e legali, …)
    a. Normalmente tra il 50% per le piccole medie imprese e il 100% per le grandi imprese rispetto ai costi diretti

Evoluzione e Manutenzione

I sistemi software devono evolversi perché:

- I requirement iniziali non sono stati catturati correttamente;
- I requirement sono cambiati durante il ciclo di vita del sistema;

Evoluzione di un sistema software è inevitabile anche se i requisiti iniziali sono catturati correttamente e lo sviluppo iniziale ha portato ad un buon prodotto.

Attività di manutenzione sono in vista di evolvere il sistema software concorrentemente al suo utilizzo per:

- Rimuovere anomalie [**manutenzione correttiva**] {20% dei costi};
- Migliorare le qualità del sistema [**manutenzione perfettiva**] {60% dei costi};

- Adattarsi ai cambiamenti dell'ambiente [**manutenzione adattiva**] {20% dei costi};

Costi di manutenzione spesso più del 50% dei costi relativi al sistema software in tutto il suo ciclo di vita. Ora spesso si aggirano intorno al 75%, sono rilevanti per il lungo periodo di tempo in cui il sistema è operativo

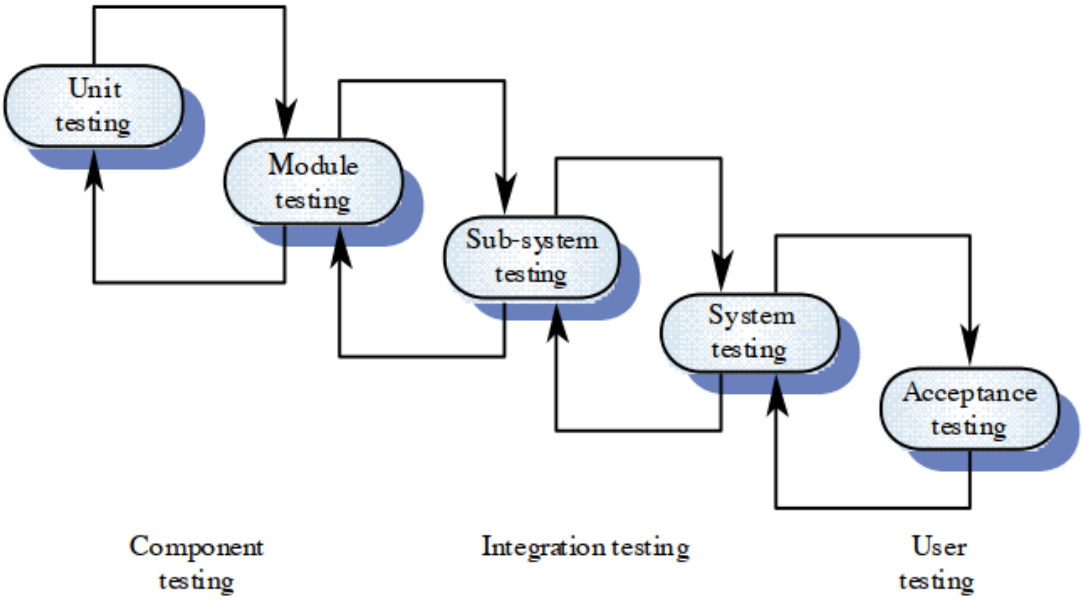
Studi dei costi di manutenzione in sistemi deployed danno evidenza empirica che la maggior parte delle anomalie sono trovate in revisioni sistematiche degli artefatti di progetto, testing strutturato e accurato è un buon modo per assicurarsi che le anomalie non si propaghino a sistemi deployed

Testing

Permette di trovare anomalie del comportamento di un sistema, non può provare che un sistema sia corretto, se il comportamento di un sistema è testato in un numero sufficientemente grande di casi, allora il comportamento è considerato accettabile, anche nei casi rimanenti.

Testing diviso tra:

- *Testing in the small*: testare le singole parti, i singoli dettagli;
    - Il testing in the small tratta il sistema come una white box e ne esamina parti sufficientemente piccole ispezionando il codice consegnato.
- *Testing in the large*: Intero sistema e le sue integrazioni
    - Il testing in the large tratta il sistema come un black box ed il testing serve a controllare che il comportamento del sistema sia come ci si aspetta.
    - `Unit testing ⟷ Module testing ⟷ Sub-system testing ⟷ System Testing ⟷ Acceptance testing`;
    - Unit testing: programmatore;
    - Module testing: capoprogetto organizzatore del modulo [o package];
    - Sub-system e System testing: Persone dedicate che fanno testing e anche delle interfacce;
    - Acceptance testing: fatto dal committente per accettare o no il prodotto;



3 Tecniche di testing in the small:

Statement Testing

anche chiamato coverage testing perché è basato sul fatto che nessuna parte del codice può essere considerata testata se non è stata eseguita → Esecuzione di ogni singola linea dell'unità per avere coverage.

Branch Testing

Chiamato anche path coverage testing: parte del codice testata se siamo entrati almeno una volta per ogni branch del codice.

Branch & Condition Testing

anche chiamato condition coverage testing perché viene targhettato la analisi delle cause che generano percorsi di esecuzione diversi.

JUnit

- Strumento per supportare testing accurato e strutturato per codice Java.
- JUnit lavora su:

    - Test (cases);
    - Test suites;
- Eclipse ha supporto diretto per JUnit per promuovere il test-driven development.

Annotations

Annotazioni

Le **annotazioni** possono essere introdotte nel codice sorgente usando la “@” (per esempio `@Override`).
Possono anche avere argomenti (esempio: `@SuppressWarning(”uncheked”)`).

Sono:

- Usate in un sorgente per suggerire al compilatore delle proprietà relative ad alcune parti del codice.
- **Metadati sintattici.**
- Spesso usate per controllare warning, errori e il comportamento del compilatore.

Le annotazioni possono essere attaccate a definizioni di:

- classi, interfacce, enums.
- costruttori, metodi, parametri.
- campi e var locali.
- packages.
- annotazioni etc.

La definizione di un’annotazione può usare le seguenti annotations per descrivere alcune proprietà delle annotazioni definite dall’utente.

- `@Target (t)`: si usa per enumerare parti del sorgente che possono essere annotate con user defined annotations.
- `@Inherited`: può essere usato come segnale che una Java annotation usata in una classe deve essere ereditata dalla sua sottoclasse.
- `@Retention (p)`: può essere usato per dichiarare la retention policy della annotazione, che è la politica che il

compilatore dovrebbe usare per decidere quando smettere di propagare le annotazioni dal compile al run time.

Introduction to Software Configuration Management Part I: The Team

Lars Bendix
bendix@sneSCM.org

Scandinavian Network of Excellence in Software Configuration Management (sneSCM.org)

Department of Computer Science Lund University Sweden

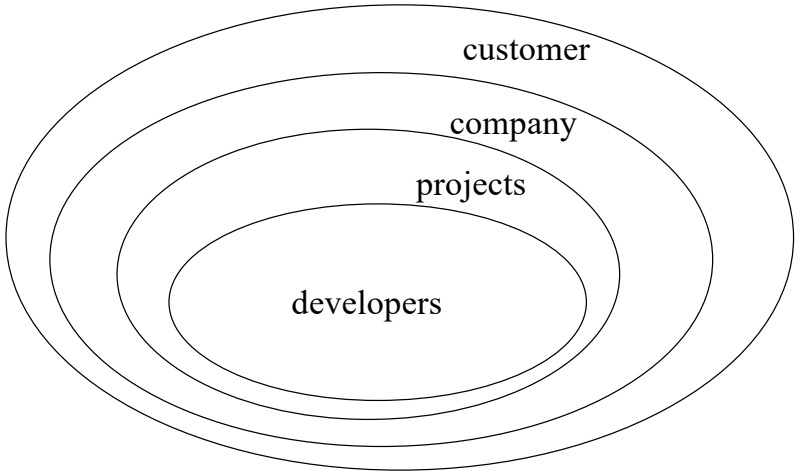
https://cs.lth.se/~bendix/Teaching/1-ECTS-SCM/

Learning goals

After this first part the student will:

- understand coordination problems
- understand some important versioning concepts
- understand basic co-ordination strategies
- know some basic CM concepts
- be able to use a version control tool

History of SCM



What is SCM?

Software Configuration Management:
is the discipline of organising, controlling and managing the development and evolution of software systems. (IEEE, ISO,...)

The goal is to maximize [programmer] productivity by minimizing [co-ordination] mistakes. (Babich)

Problems

Identification:

You should be able to identify the single components and configurations.

- This worked yesterday, what has happened?
- Do we have the latest version?
- I have already fixed this problem. Why is it still there?

Change tracking:

Helps in tracking which changes have been made to which modules and by whom, when and why.

- Has this problem been fixed?
- Who is responsible for this change?
- This change looks obvious - has it been tried before?

Problems

Software production:

Construction of a program involves pre-processing, compilation, linking, etc.

- I just corrected this, has something not been compiled?
- How was this binary produced?
- Did we do all the necessary steps in the right order?

Concurrent updating:

The system should offer possibilities for concurrent changes to components.

- Why did my changes disappear?
- How do I get these changes into my version?
- Are our changes in conflict with each other?

Excuses for not using CM?

- CM only applies to source code
- CM is not appropriate during development because we use rapid prototyping
- It's not that big a project
- You can't stop people from making a quick patch
- We lower our cost by using only minimum-wage persons on our CM staff because CM does not require much skill ;-)

How does a programmer spend his time?

- 50% interacting with other team members
- 30% working alone
- 20% non-productive activities

(Software) development

- re-use things
- sharing things
- memory/history

- collaboration
- co-ordination
- communication

SCM Hall of Fame

Wayne Babich, 1986:
Software Configuration Management – Coordination for Team Productivity

Team co-ordination problems:

- shared data
- double maintenance
- simultaneous update

"An ounce of [SBoM] is worth a pound of analysis"

Wayne Babich I

Sometimes it is embarrassing to be a computer programmer. What other profession has such a remarkable rate of schedule and cost overrun and outright failure? [...]

Our failures are not of the individual contributors; most of us design, code and debug adequately or even well. Rather, the failure is one of coordination. Somehow we lack the ability to take 20 or 30 good programmers and meld them into a consistently productive team.

Wayne A. Babich, 1986

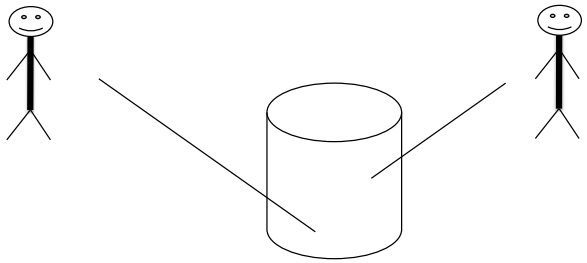
Wayne Babich II

The term configuration management derives from the hard engineering disciplines [...], which use change control techniques to manage blueprints and other design documents. The term software configuration management has traditionally been applied to the process of describing and tracking releases of software as the product leaves the development group for the outside world.

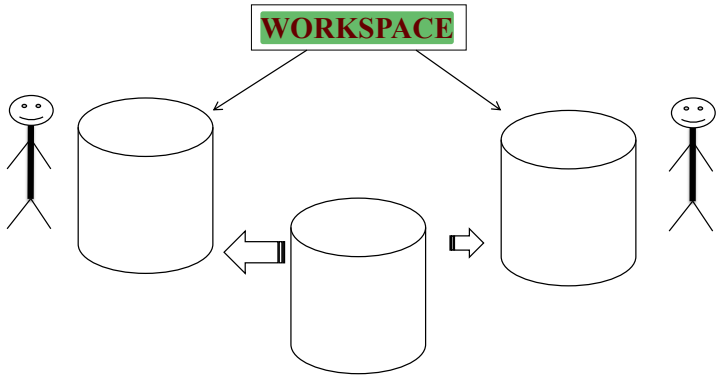
I use the term in a more expansive sense. I include not just the formal release of software to the customer, but the day-to-day and minute-by-minute evolution of the software inside the development team. Controlled evolution means that you not only understand *what* you have *when* you are **delivering it**, but you also understand *what* you have *while* you are **developing it**.

Wayne A. Babich, 1986

Shared data



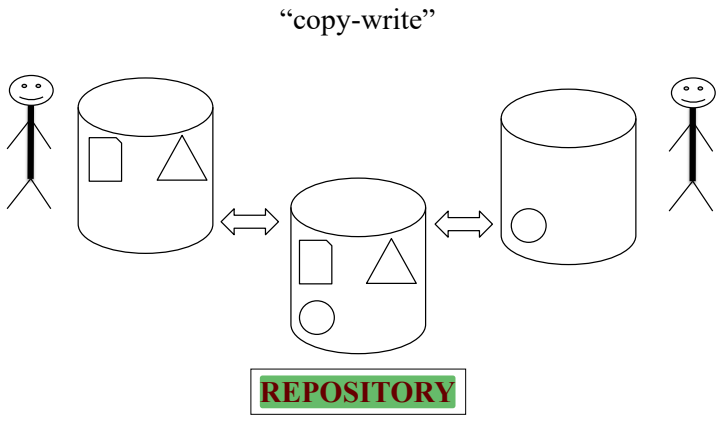
Solution 1



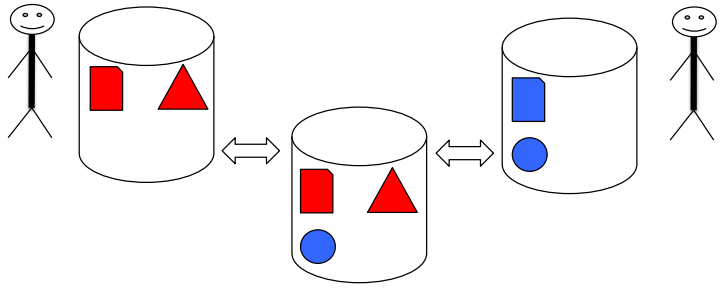
doppia manutenzione

Double maintenance

Solution 2



Simultaneous update



REPOSITORY

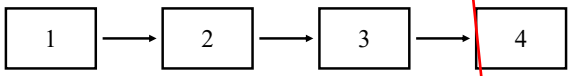
Co-ordination

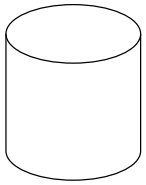
Working in isolation:
- local dynamicity
- global stability
- problems:
  - double maintenance

Working in group:
- global dynamicity
- problems:
  - shared data
  - simultaneous update

Solution 3

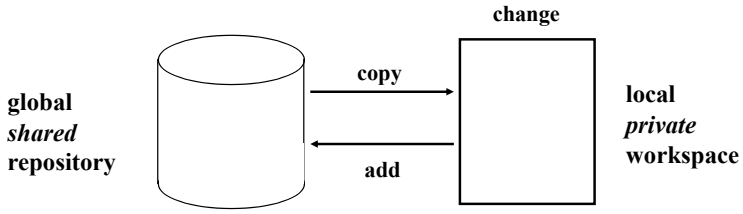
"copy-add"





VERSIONING

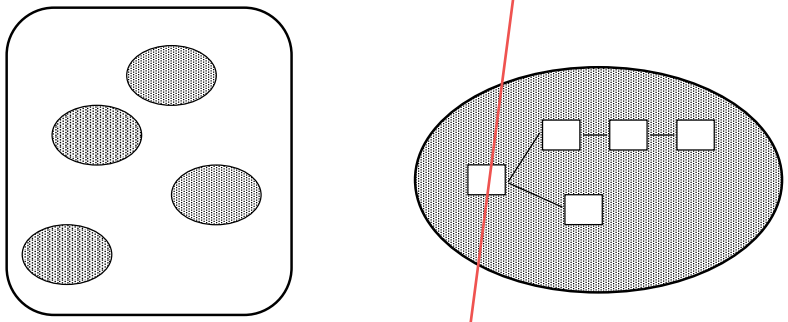
Model



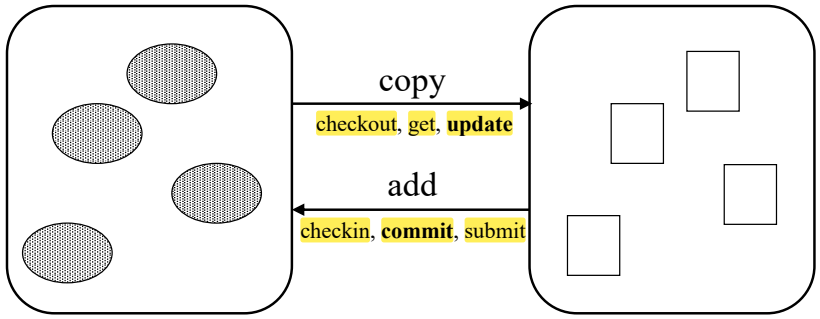
Principle: components are **immutable**

Babich:
- shared data
- double maintenance
- simultaneous update

Concepts

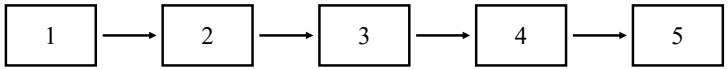


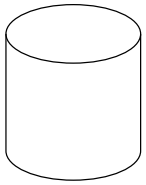
Workspace/repository



Solution 3

"copy-add"





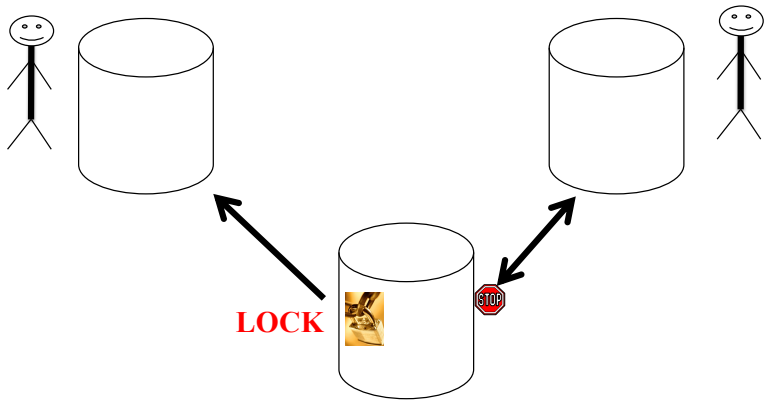
Concurrency check

Parallel work

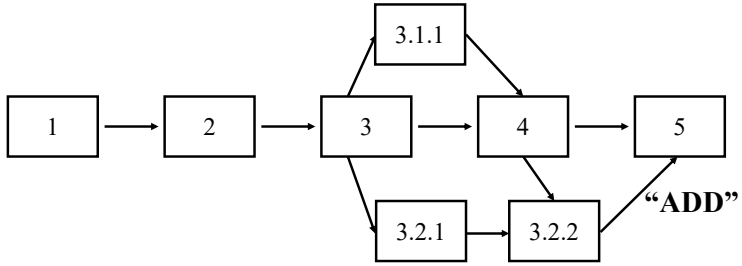
Concurrency strategies:

- pessimistic (locking)
- optimistic (copy-merge)

Locking



Branching



Copy/merge work model

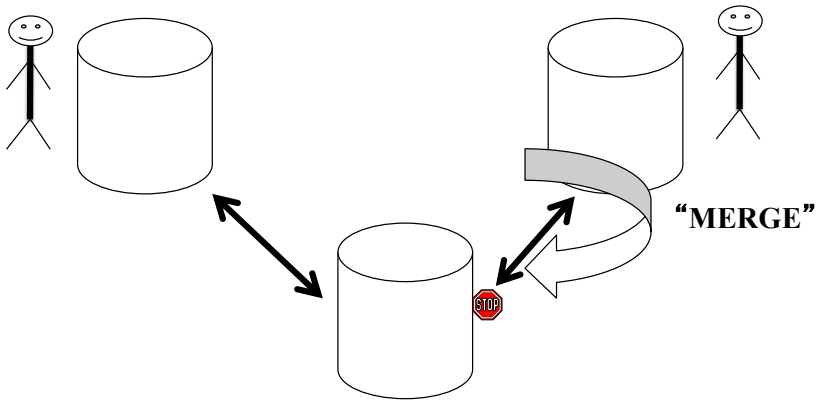
Can we *lock* the things we want to work on? NO!

So we **copy** everything to our workspace ...
... and everyone else copy to their workspaces ...
⇒ double maintenance !!

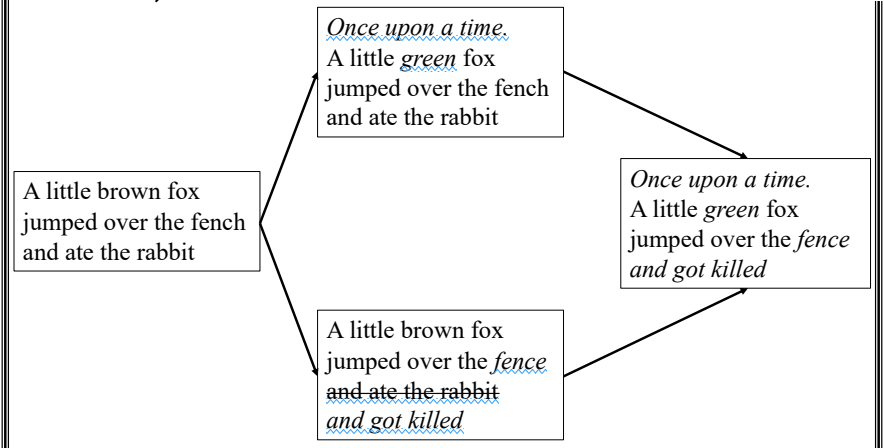
Fortunately ”update” has a built-in **merge** facility:

- We first merge *from* the repository *into* the workspace
- Then we check and fix problems
- Finally we commit (**add safely**) to the repository

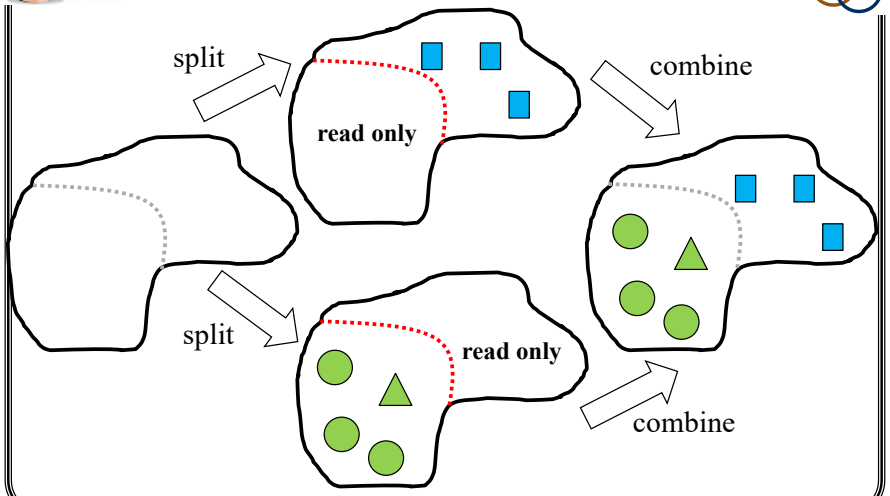
Copy-merge



3-way merging



Split-combine



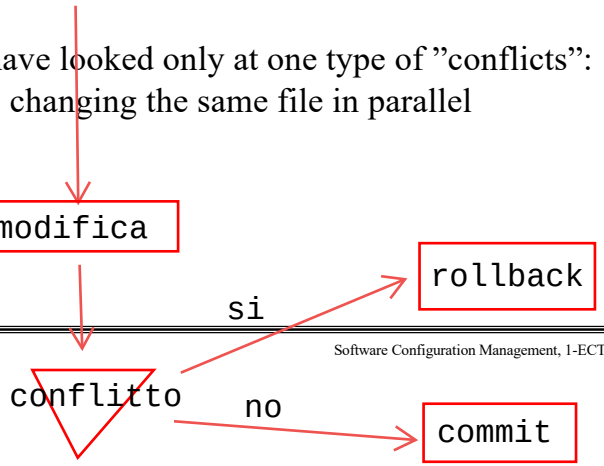
Long transactions

We do our change as a **logical unit**, we want this logical unit to be kept/recorded:

⇒ if there is anything that ”conflicts”, we must abort the **whole** commit (long transaction/atomic commit)

So far we have looked only at one type of ”conflicts”:

- people changing the same file in parallel



Strict long transactions

But we are interested in whole **systems/configurations**:

- should we merge systems/configurations?
- yes, but **never** into the repository!

From physical conflicts to logical conflicts:

No new configurations should be created in the repo
New configurations should be created in the workspace
New configurations should be tested in the workspace
Then it should be “copied/added” to the repository

What can we do if the tool does not have SLT?

Working together

I might still not want things to change:

- baseline (freezing a configuration)

I might want to know who is affected by my changes:

- traceability

I might want to know exactly how a system was built:

- software bill of material (SBoM)

Introduction to Software Configuration Management Part II: The Company

Lars Bendix
bendix@sneSCM.org

Scandinavian Network of Excellence in Software Configuration Management (sneSCM.org)

Department of Computer Science Lund University Sweden

Learning goals

After this lecture the student will:

- know traditional SCM activities
- know central SCM concepts and their use

Different CM roles

Operational SCM (users & operators):

- developers (ordinary/wizards)
- project managers
- Quality Assurance
- company managers
- customers

Strategic SCM (designers):

- SCM processes
- SCM tools
- SCM improvement

Building on rock?

SCM is a CMM level 2 key process area

Req.	Design	Testing	Coding	QA

Software Configuration Management

Definition of CM

Configuration management is a systems engineering process for establishing and maintaining *consistency* of a product's performance, functional, and physical attributes with its requirements, design, and operational information throughout its life

CM, when applied over the life cycle of a system, provides *visibility and control* of its performance, functional, and physical attributes. CM *verifies* that a system performs as intended, and is identified and documented in sufficient detail to support its projected life cycle. The CM process *facilitates orderly management* of system information and system changes

Traditional configuration management

Identification: The selection and handling of which artefacts that are important for creating the product.

Change Control: The controlling of changes to a configuration of a product and its artefacts.

Status accounting: The recording and reporting of the implementation of changes to a product and its artefacts.

Audit: The validation of configuration of a product for compliance with its definition.

Configuration Identification – I

Definition: the selection, designation and description of **configuration items**.

Purpose/goal: to capture, preserve and make available all the important things in a project – and to make sure that we have unique identification for these.

Configuration Identification – II

Terminology:

- artefacts
- configuration items (CIs)

Types of CIs:

- atomic CIs
- configurations
- (built) products

Configuration Identification – III

What to do with CIs:

- **kept safely**
- shared with others
- versioned

How to handle CIs:

- **unique naming**
- meta data
- structured storage

Configuration Identification – IV

Traceability:

- horisontal (versions)
- vertical (dependencies/relations)

Software Bill of Materials (SBoM):

- *what* went into the product
- *how* it was built

CMDB:

- items + information + traceability
- entities + attributes + relations

Identification

We should be able to identify any version of a component, also when it is outside of the version control tool:

- as a text file
- as object code
- as a part of a configuration
- on paper

Configuration Status Accounting – I

Definition: the **recording and reporting of information** needed to manage and work on a project.

Purpose/goal: to allow people to easily get all the information that they need to carry out their work in an informed way.

Configuration Status Accounting – II

Information reporting:

- Status Accounting answers questions
- to answer questions **we need data**
- different people have different questions in different situations

Queries on the CMDB:

- standardized
- ad hoc

Sometimes we (still) make a report

Configuration Status Accounting – III

Recording of information:

- Status Accounting may require "updates" to the CMDB schema
- make sure that data is captured by the right person at the right time
- incorrect data is worse than no data
- data can/should be captured automatically by tools

The information should be available - and useful - to everyone, to keep them informed of the status on a day-to-day basis.

Answers to Questions

Questions:

- Do we have the latest version?
- I have already fixed this problem. Why is it still there?
- I just corrected this, has something not been compiled?
- How was this binary produced?
- Has this problem been fixed?
- Who is responsible for this change?
- This change looks obvious - has it been tried before?

More questions:

- who made this change?
- when was it made?
- what changed?
- why did it change?

Configuration Change Control – I

Definition: the proposal, evaluation, coordination, approval or disapproval, and implementation of approved **changes to baselined CIs**.

Purpose/goal: to ensure that proposed changes are classified and evaluated and that approved changes are implemented, documented and verified.

Configuration Change Control – II

Change request (CR):

- problem report
- waiver
- requirement

- testers
- customers
- (sub-) contractors

Must contain **all** needed information about the change

Configuration Change Control – III

Change process:

CR => filtering process => Impact Analysis => CCB

Impact Analysis:

- time and resources
- costs and consequences

CCB => implementation => test => QA => closed

Project manager's "tool"

Configuration Change Control – IV

Change Control Board (CCB):

- meet on a regular basis
- CM prepares (CRs) before the meeting
- experts can be invited
- it is headed by a chairman
- the chairman has dictatorial power
- ad hoc meetings if needed

Outcomes:

- approved
- rejected
- deferred
- escalated

Configuration Audit – I

Definition: an independent evaluation of a (changed) CI to **ascertain compliance** to specifications, standards, contractual agreements and other criteria.

Purpose/goal:

- to verify that the CI matches the description in the specification and documentation.
- to ensure that work has been performed in the correct way, that is, in conformance with the development standards and guidelines.

Configuration Audit – II

It is a verification of:

- the product (CI) – does it conform?
- the process – did we follow the "rules"?

Before the product (CI) is accepted into the baseline

A Configuration Audit acts as a "**quality gate**"

CAs can be done at:

- various times
- various formality

Configuration Audit – III

A **baseline** is:

- something that we want to remain fixed – for some time
- something that is named so we can return to it
- something that has a specified (high?) quality
- when it is changed, it is done with much care

What can be baselined:

- requirements
- tests
- code
- …

Release management

Re-creation (and identification) is important:

- identification
- options
- tools

- tracing all relevant pieces and information
- baselining of the project/product
- software bill-of-material for a product
- version control for tools too

Everything that has been in contact with a release must potentially be preserved.

Business value

How does SCM translate to business value?

- faster development
- better quality
- greater reliability = `affidabilità`

Seven critical requirements:

- safety
- stability
- control
- auditability
- reproducibility
- traceability
- scalability

ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/pmpp/sevenkeys.pdf

Safety

SCM keeps project assets safe from:

- corruption
- unintentional damage
- unauthorised access
- other disasters
- even safe from changes!!!

SCM provides:

- secure access
- reliable recovery
- an easy way to rematerialize *past* configurations

It is the protection of key business assets.

Stability

True stability has some essential elements:

- guaranteed stable work areas
- individual control over introduction of new code
- the option to gradually update the work area

Introducing instability can cause a downward spiral.

a situation in which something continuously decreases or gets worse.

```
Audit: verifica della correttezza dei dati di bilancio e delle procedure
```

Control

Find the delicate balance between:

- enforcing appropriate controls
- imposing bothersome constraints

Integrating contributions requires control over:

- who works on what
- how changes flow from developer to integration
- who can work where
- consider a “food chain” approach to development

Plan how you want to work - and enforce those rules.

Auditability

Auditability refers to the ability to answer:

- who, what, when and where
- about components or configurations
- tracks and records meta-data about changes

Common questions:

- did foo.java get changed?
- who made a change to this line in this version?
- were all bugs fixed for this patch?
- who did the fixes?

It allows you to query the data base to find answers easily.

Reproducibility

You must be able to quickly reproduce previous configurations:

- reproduce the exact configuration of the code
- and also corresponding versions of test cases
- including documentation and more
- and the right version of all tools used
- so you can roll back everything to any point in time

The tool must be able to:

- reproduce all the files in a build
- take into consideration the directory structure
- provide "name space" versioning

Can ensure that what is being built has also been tested.

Traceability

Comprehensive traceability involves the ability to:

- identify the version of a released product
- identify files and versions changed by a change request
- identify the change request that creates a certain version
- identify the configurations that contain a specific file

This allows us to:

- implement the "time machine"
- answer questions about the past
- modify the past?

Traceability tells you how you arrived at where you are.

Scalability

Scalability of SCM systems includes:

- configurable and functional when little is needed
- versatility to manage growth of the project
- ability to support (geographically) distributed teams
- handling of outsourced production
- reliability in the light of scaling

The tool should:

- scale to support large teams
- not impose burdens on small teams

Changing SCM tool is very painful for everyone involved.

Summarising

Effective SCM will:

- create a secure and predictable environment for working
- automate everyday build and versioning tasks
- provide quick access to file and versioning information
- be agile and robust to adapt to changing conditions
- provide managers with key information and data
- support end-to-end tracking of changes
- make it easy to do the right things
- make it hard to do the wrong things

It looks like it is worth the money.

Benefits of CM

- Insurance against accidents
- Help in debugging
- Support for co-ordination
- Company assets
- Intellectual Capital Report

Configuration management promotes consistency, productivity and quality