Ingegneria del Software T

Kevin Michael Frick

8 giugno 2020

Disclaimer: Questo disclaimer, di solito, lo metto alla fine. Ma questa volta siamo nel 2020 e la COVID-19 è una realtà, così come gli esami online, quindi ho pensato che fosse il caso di assicurarmi che tu lo leggessi (legga? boh, dopo tre anni di ingegneria mi sto scordando l'italiano). Lo so che hai notato che il docente del corso, durante la simulazione, non ha mostrato di preoccuparsi particolarmente di cosa ci fosse nella stanza intorno a te. Quindi, forse, sei tentat@ di stamparti questo file, nasconderlo chissà dove e, come disse un vecchio saggio, *copiare come un animale* durante l'esame. È comodo e facile, no? *E poi, Kevin, non fai questi file proprio perché vuoi che i professori si accorgano che non è il caso di fare sempre le stesse domande?*

Già. Li faccio per questo motivo. Però se tutt@ scrivono esattamente quello che c'è scritto in questo file, il docente lo noterà subito, capirà cosa è successo e potrebbe invalidare la prova. E ovviamente alla prossima prova farà attenzione che nessun@ copi, obbligandoci a mettere cinque webcam in giro per la stanza e una in cucina perché non si sa mai. Se invece userai questo file come aiuto per lo studio, è comunque probabile che andrai bene all'esame (perché allenarsi a rispondere alle domande è un metodo molto efficace per studiare) e questo materiale lo imparerai davvero.

Ed è importante che impari davvero il materiale di questo corso, così come quello di qualunque altro corso universitario.

Questo è un corso del terzo anno. È ragionevole pensare che a breve ti laureerai e andrai a lavorare come ingegnere informatico, e non siamo più nel 1965: i sistemi informatici sono critici quanto ponti e strade. Se i sistemi informatici falliscono, la gente muore. Pensa al caso dei Boeing 737 MAX.

Quindi, per favore, non copiare. Oppure, un giorno, un sistema informatico mal progettato potrebbe uccidere te o persone a te care. E in ogni caso, ricorda:

Questo documento può contenere errori e imprecisioni che potrebbero danneggiare sistemi informatici, terminare relazioni romantiche e rapporti di lavoro, portare i gatti a liberare le proprie vesciche sulla moquette e causare un conflitto termonucleare globale. Procedere con cautela.

1. **Processo di sviluppo del software**

   (a) Revisione black box e white box

   > **R:** Il collaudo delle funzionalità di un sistema, così come quello della sua sicurezza, può avvenire seguendo un approccio a black box e a white box.
   >
   > La revisione black box è mirata a collaudare le proprietà del sistema che sono visibili dall'esterno e che possono essere apprezzare senza conoscere il funzionamento del software: affidabilità, semplicità d'uso, velocità ecc. Nel caso dei test di sicurezza, il collaudatore si mette nei panni dell'attaccante e cerca di penetrare nel sistema sfruttando le conoscenze che si possono avere semplicemente guardando il sistema.
   >
   > La revisione white box invece è mirata ad analizzare la struttura interna del software e valutare caratteristiche come modularità e leggibilità del codice. Queste caratteristiche sono un modo per realizzare le proprietà esterne e ne influenzano la qualità. Nel caso dei

test di sicurezza, il test white box consiste nell'analisi del codice e nella scrittura di test molto specifici che sfruttano le caratteristiche del codice per violare il sistema.

(b) Rational Unified Process

R: Le fasi di processo di sviluppo di un software sono specifica, sviluppo, validazione ed evoluzione. RUP è un modello di processo di sviluppo iterativo unificato, che contiene al suo interno tutti i modelli di processo: qualsiasi modello di processo può essere visto come parte di RUP. È un modello ibrido quindi può comportarsi come qualunque altro modello. Definisce delle direttive che possano essere realizzate e implementate in modo da realizzare diversi processi. Una certa azienda può usare RUP per un processo vicino a quello evolutivo, oppure a componenti ecc. È pensato per progetti di grandi dimensioni perché ha una parte formale abbastanza importante. RUP vede il processo da tre prospettive ortogonali:

1. Prospettiva dinamica: far vedere come il modello si sviluppa nel tempo. Divisa in:
   - Avvio = analisi di fattibilità. Il suo scopo è definire in maniera accurata, non ambigua quale sia l'ambito in cui ci si muove. Esempi: il tipo di mercato, gli attori e i sistemi che interagiranno. Si usano modelli di casi d'uso e una serie di modelli formali.
   - Elaborazione: racconta come il sistema sarà fatto, includendo l'analisi del dominio e buona parte della progettazione, dove non necessariamente si arriva a specificare i componenti ma quantomeno a specificare l'architettura generale del sistema. Modello dei casi d'uso, descrizione architettura, sviluppo di un prodotto eseguibile. Questa fase di elaborazione rivede quanto è previsto nella fase di avvio.
   - Costruzione: parte finale della progettazione, seguita da sviluppo e test.
   - Transizione: deployment, manutenzione, addestramento degli utenti, eventuale retroazione.

   Tutte queste quattro fasi possono avere retroazioni alle fasi precedenti, non necessariamente immediatamente precedenti.
2. Prospettiva statica: quali sono le attività da realizzare all'interno del progetto. Queste attività si chiamano "workflow", da non confondere con la visione dinamica: non è detto che i requisiti, ad esempio, si analizzino nella fase di inizio. Questo è vero solo nel modello a cascata, ma non ad esempio in quello iterativo. Si ha separazione tra tempo e attività svolte. Esisono sei workflow principali e tre di supporto: ci dovrebbero essere nove entità nell'azienda, all'interno delle quali ognuno ha un compito specifico. Ogni workflow ha un modello UML associato. Tutti i workflow possono essere attivi in ogni stadio del processo. I workflow sono;
   (a) Modellazione delle attività aziendali: i processi che sono all'interno dell'azienda vengono modellati come UML use case;
   (b) Requisiti: identificano quali siano gli attori che partecipano al sistema e racconta come sia fatta l'interazione col sistema, servendosi degli UML use case;
   (c) Analisi e progetto: si crea il modello di progetto. UML, per questa fase, definisce vari modelli: architetturali, componenti, a oggetti, sequenziali.
   (d) Implementazione: saranno interessati gli implementatori, e se l'analisi del progetto è stata fatta usando i modelli UML è possibile generare automaticamente

codice oppure in ogni caso UML rende facile il trasferimento di specifiche dalla fase di progetto all'implementazione.

(e) Test: dei sottosistemi e del sistema finale;

(f) Rilascio.

I tre workflow di supporto, che sono ortogonali alla fase di sviluppo e ai sei principali e adottabile da ciascuno di essi, sono:

(a) Gestione delle modifiche: gestisce il cambiamento (es. diverse versioni del software), non solo del software ma anche dei modelli;

(b) Gestione del progetto: personale, costi, ecc. Supporto in toto allo sviluppo;

(c) Ambiente: si occupa degli strumenti utilizzati per progettare e implementare (disegni UML, IDE, ecc).

3. Prospettiva pratica: definisce le "buone prassi" da seguire durante il processo. Sono idee che sono già note all'interno dell'azienda. Divisa in sei pratiche fondamentali, sei regole che dovrebbero sempre guidare il processo di sviluppo. I design pattern / principle derivano da queste pratiche.

(a) Sviluppo ciclico: il modello a cascata va bene, ma deve esserci retroazione. Fin da subito vanno mostrate al cliente le parti del software che soddisfano i requisiti più importanti.

(b) Gestione dei requisiti = documentazione: usare modelli per la specifica dei requisiti, condividerli con i clienti e farli accettare da loro.

(c) Usare architetture basate su componenti: riutilizzare parti di software già scritti rende l'architettura del sistema più modulare ùindi più facilmente modificabile emanutenibili.

(d) Creare modelli visivi del software: scrivere il codice non basta e ci vuole una visualizzazione di tutte le fasi di produzione del software. Non è un caso che UML sia stato sviluppato insieme a RUP, perché deriva proprio da questa prospettiva pratica. I modelli UML permettono di capire velocemente che cosa farà il software. Questi modelli devono mostrare sia il comportamento statico che quello dinamico del software.

(e) Verificare la qualità: ci devono essere delle misure che dicano se il software soddisfa o meno degli standard di qualità, che devono essere definiti all'interno dell'azienda.

(f) Controllare le modifiche: utilizzare degli strumenti e delle pratiche che permettano di gestire modifiche, configurazioni ecc.

Le varie fasi procedono per iterazioni: le iterazioni iniziali della fase di avvio ad esempio usano molto il workflow dei requisiti, così come quella di elaborazione, mentre è meno usato dalla fase di costruzione.

2. **Analisi e progettazione orientata agli oggetti: linguaggi di modellazione, UML.**

(a) Polimorfismo secondo Cardelli-Wegner

R: Il polimorfismo è la capacità di uno stesso elemento di assumere forme diverse in contesti diversi, o di elementi diversi di assumere la stessa forma in un determinato contesto. La classificazione Cardelli-Wegner dei polimorfismi li divide in due macro-

categorie, universali e ad-hoc.

Gli elementi universalmente polimorfici possono assumere un numero infinito di forme. Questo si osserva, nella programmazione orientata agli oggetti, nel *polimorfismo per inclusione*, rappresentato dall'overriding dei metodi e dal binding dinamico delle funzioni. Il polimorfismo universale si osserva inoltre nella programmazione generica rispetto ai tipi, che esprime il *polimorfismo parametrico*: si definisce una classe in cui il tipo di una o più variabili è un parametro della classe stessa. Da ogni classe generica vengono generate classi indipendenti, che non hanno un rapporto di ereditarietà.

Un elemento polimorfico ad-hoc, invece, può assumere un numero finito di forme. Ciò si osserva nell'*overloading* di funzioni e operatori, che richiede una ridefinizone per ogni insieme di argomenti accettati. Il polimorfismo ad-hoc si osserva inoltre nella *coercion*, quando una variabile di un certo tipo viene convertita, esplicitamente o implicitamente, a un tipo diverso. Anche in questo caso, le conversioni possibili devono essere definite una per una in fase di programmazione.

3. **Analisi dei requisiti: raccolta dei requisiti e loro validazione, analisi del dominio, analisi dei requisiti, casi d'uso e scenari.**

   (a) Tipologie di requisiti

R: Vi sono tre tipologie di requisiti:

1. Requisiti funzionali: descrivono cosa fa il sistema, ma non come lo fa. Sono un elenco di servizi che il sistema deve fornire. In particolare, per ciascuno di questi servizi, bisogna indicare come gli input fanno reagire il sistema e come si comporta in alcune situazioni. Descrivono anche cosa *non* deve fare il sistema in precise situazioni. Le specifiche devono essere complete, ovvero definire tutti i servizi forniti, e coerenti, cioè non essere in contraddizione tra loro.
2. Requisiti non funzionali: descrivono altre caratteristiche del sistema, ad esempio quali siano le proprietà del sistema (requisiti del prodotto), vincoli di sviluppo (requisiti organizzativi) o vincoli derivanti da altri sistemi esterni o contesti legislativi/etici (requisiti esterni). I requisiti non funzionali tendenzialmente sono espressi in modo vago. Inoltre, vengono spesso mescolati con i requisiti non funzionali o si contraddicono tra di loro.
3. Requisiti di dominio: derivano dal dominio di applicazione del sistema e indicano come il sistema debba funzionare all'interno di un certo dominio, e raccontano certi requisiti che sono specifici del dominio. Dovrebbero raccontare a progettisti quale sia il dominio di applicazione del sistema, che spesso non è chiaro al progettista.

4. **Progettazione: progettazione dell'architettura del sistema.**

   (a) Principi per l'architettura dei package.

R: Vi sono tre principi che guidano la progettazione dell'architettura dei package:

- Il principio di equivalenza riuso/rilascio (reuse/release equivalence principle) afferma che la granularità del riuso e del rilascio debbano coincidere. Se si ha un elemento che deve essere riutilizzato, esso deve essere periodicamente rilasciato/fornito: chi realizza il software non è chi lo usa. Se non si è in grado di garantire che il software venga mantenuto, nessuno lo utilizzerà. Il motivo per il quale le classi si raggrup-

pano in package è il riutilizzo: se si vogliono riutilizzare certi elementi, essi vanno raggruppati in package.

- Il principio di chiusura comune (common closure principle) afferma che le classi che vengono modificate assieme debbano essere incluse nello stesso package. Siccome, all'interno di un progetto SW, ogni volta che viene modificato un package l'intero package va ridistribuito, e tutti gli utilizzatori devono aggiornarlo, è importante minimizzare il numero di package che cambiano quando si cambia una classe.
- Il principio di riutilizzo comune (common reuse principle) afferma che le classi che non vengono riutilizzate insieme non vadano incluse nello stesso package. Ciò complementa il CCP, ma non ne è implicato: deriva invece dal fatto che una dipendenza da una classe in un package porta con se una dipendenza dall'intero package.

È difficile soddisfare tutti e tre i principi contemporaneamente: il primo e il terzo tendono a facilitare l'utilizzo dei package da parte dei clienti mentre il secondo tende a semplificare l'utilizzo dei package da parte dello sviluppatore. Il CCP tende a suggerire di creare package molto grandi, mentre il RREP e il CRP tendono a suggerire package molto piccoli. Per questo motivo, normalmente si cerca di soddisfare il CCP, e una volta che l'architettura dei package si stabilizza si esegue un refactor per rimpicciolire i package e soddisfare gli altri due principi. In questo modo, all'inizio si facilita il lavoro agli sviluppatori, e in seguito per aumentare la vita utile del SW si applicano gli altri due principi.

Vi sono inoltre tre principi che descrivono le relazioni tra i package:

- Il principio delle dipendenze acicliche afferma che vadano evitati cicli di dipendenze tra package: dato che modificare una classe richiede di ricompilare il package a cui appartiene quella classe e tutti i package che dipendono da esso, una singola dipendenza ciclica può portare alla necessità di ricompilare l'intero progetto se si cambia una sola classe. È possibile rompere un ciclo spostando la dipendenza da una classe esterna al package a una interfaccia definita all'interno del package e implementata dalla classe, ma non è sempre questa la soluzione;
- Il principio delle dipendenze stabili afferma che un package dovrebbe dipendere solo da package più stabili di esso. Un package dal quale dipendono molti altri package non può essere volatile, e al contrario i package che cambiano spesso non possono avere molte dipendenze, per evitare che diventi necessario ricompilare molto spesso tutte queste dipendenze. Per ottenere dipendenze stabili ci si conforma al CCP: se molte classi dipendono da una classe, questa e le sue dipendenze vanno tutte inserite nello stesso package e ricompilate assieme, evitando in questo modo effetti a cascata;
- Il principio delle astrazioni stabili afferma che i package stabili dovrebbero essere package astratti. Ciò significa che un modo per ottenere stabilità è rendere astratti gli elementi che si vuole siano stabili: un package astratto non ha codice al suo interno che possa cambiare.

Il principio di stratificazione unisce questi ultimi tre principi: un sistema di package a livelli non ha cicli e permette di posizionare i package più instabili in alto, in modo che abbiano poche dipendenze entranti, e i package più stabili - tendenzialmente astratti - in basso, in modo che possano avere molte dipendenze entranti.

5. **Principi di progettazione**

(a) Rigidità, fragilità, viscosità e immobilità del software.

R: La qualità di un dato progetto potrebbe dipendere da vari fattori, principalmente dalle priorità organizzative. Utilizzare principi che rendono il codice manutenibile, a valle spesso si migliora anche affidabilità, efficienza e sicurezza. Definire cosa rende un software mantenibile, però, è molto più complesso che definire cosa rende un software difficile da mantenere.

- Un software è rigido quando un singolo cambiamento apportato a una singola parte del progetto ha effetto su tante altre parti del sistema, e modificare un componente porta a dover modificare molti altri componenti. L'effetto della rigidità si vede quando i dirigenti, se sanno che il SW è rigido, tenderanno di impedire agli sviluppatori di effettuare modifiche, anche piccole, perché non sanno quando gli sviluppatori finiranno effettivamente di modificare il SW.
- Un software è fragile quando una modifica di una parte del progetto ha effetto su parti del sistema che non era previsto interessasse e che sembrano non essere correlate. L'effetto della fragilità si vede quando i dirigenti non permettono agli sviluppatori di modificare il SW perché temono che tali modifiche inficino il funzionamento di tutto il sistema.
- Un software è immobile quando il codice non è utilizzabile in altre applicazioni, ad esempio perché porta con sé troppe dipendenze oppure perché richiede troppo tempo per essere adattato al nuovo, specifico caso d'uso. Come effetto di software immobile, gli sviluppatori perdono molto tempo a riscrivere, inutilmente, pezzi di SW che sono già stati scritti.
- Un software è viscoso quando fare modifiche al software, anche piccole, porta a fare qualcosa di "sbagliato". In un software viscoso è facile fare la cosa sbagliata e difficile fare la cosa giusta. Uno sviluppatore, normalmente, effettua modifiche a un software nel modo meno costoso: è possibile che il modo in cui è scritto il SW assicuri che il metodo meno costoso sia anche quello che meglio preserva il progetto iniziale, oppure no. Se il progetto è viscoso in partenza, allora è più complicato preservarlo che mettere una "toppa". Ciò può capitare anche se tutto l'ambiente di sviluppo è macchinoso, lento e inefficiente. L'effetto della viscosità è un visibile degrado della manutenibilità, che porta con il tempo a software rigido, fragile e immobile.

Il software può essere mal progettato per le ridotte capacità dei progettisti, per l'obsolescenza delle tecnologie utilizzate, per limiti imposti che riguardano il tempo da spendere o le risorse da utilizzare, o a causa di un dominio troppo complesso.

Inoltre, il SW può diventare mal progettato per motivi meno ovvi: la "marcescenza" del software è un processo lento. I requisiti possono cambiare in maniera inaspettata, e se i cambiamenti sono molto forti il software può smettere di essere valido. Un esempio di una situazione simile è la necessità di mantenere la retrocompatibilità tra versioni successive di un SW. Infine, talvolta gli implementatori potrebbero aggiungere dipendenze non pianificate, che vanno gestite accuratamente.

(b) Principio di sostituibilità di Liskov.

R: Il principio di sostituibilità di Liskov è un principio di programmazione orientata agli oggetti che definisce il rapporto tra una classe e le sue sottoclassi. È possibile enunciare una "versione debole" del principio: a ogni variabile che fa riferimento alla classe base deve poter essere assegnato un riferimento di una sottoclasse. Questa versione, però, non coincide con l'effettiva formulazione del principio, che è molto più forte. Il principio

afferma che ogni programma che utilizzi istanze della classe base deve poter utilizzare istanze delle sottoclassi senza che il comportamento logico del programma cambi. Ciò significa che, per ogni metodo re-implementato dalla sottoclasse:

- le precondizioni devono essere uguali o meno stringenti;
- le postcondizioni devono essere uguali o più stringenti;
- la semantica della classe base deve essere mantenuta;
- non è possibile aggiungere vincoli comportamentali alla classe base.

Ad esempio, un Quadrato è un Rettangolo ma aggiunge un vincolo: quando viene modificata l'altezza viene modificata anche la larghezza. Ciò significa che una modifica a una dimensione comprende un cambiamento nello stato dell'oggetto che non avviene invece nella superclasse, quindi un Quadrato non può essere sostituibile secondo Liskov a un Rettangolo.

Un altro modo per esprimere il principio è affermare che ogni sottoclasse onori lo stesso "contratto" della classe base, con riguardo alle postcondizioni e alle precondizioni.

(c) Open/Close Principle

R: L'Open/Close Principle è un principio di programmazione che agevola la scrittura di software riutilizzabile. L'OCP afferma che le entità software (classi, package ecc) dovrebbero essere aperte alle estensioni ma chiuse alle modifiche. Ciò significa che le entità possono essere estese aggiungendo nuovo stato o nuovo comportamento, ma l'interfaccia non deve essere modificabile: è ben definita, pubblica e statica. Una volta chiusa e definita l'interfaccia, l'interno dell'entità può essere modificata a piacimento. Ciò permette di modificare/estendere l'oggetto senza che il funzionamento sia affetto dalle modifiche. Per realizzare l'OCP si ricorre all'incapsulamento e all'astrazione: si definisce per prima cosa un'interfaccia stabile, che può poi essere estesa definendo, ad esempio, una classe concreta che implementa quell'interfaccia.

(d) Dependency inversion principle

R: Il principio di inversione delle dipendenze afferma che le dipendenze debbano essere relative ad astrazioni e non a implementazioni concrete. La direzione della specializzazione, quindi, dovrebbe essere verso una classe astratta o un'interfaccia e non verso una classe concreta. I moduli di alto livello non dovrebbero dipendere da quelli di basso livello, bensì entrambi dovrebbero dipendere da astrazioni: il cliente deve interrogare le astrazioni e il fornitore di servizio implementarle.

Tendenzialmente, i moduli di basso livello sono i più soggetti ai cambiamenti, in quanto contengono codice più complesso. Se i requisiti cambiano o è necessario correggere un errore e si modificano i moduli di basso livello, tutti i moduli che ne dipendono devono essere modificati, causando rigidità, dato che bisogna intervenire su un alto numero di moduli, fragilità, perché si ha la potenzialità di introdurre errori in più parti del sistema, e immobilità, perché non si può riutilizzare il codice perché porta con sé troppe dipendenze.

Se i moduli di basso livello realizzano le interfacce e quelli di alto livello dipendono dalle interfacce, che sono stabili, è possibile modificare i moduli di basso livello senza modificare nulla nel codice dei moduli di alto livello. Le interfacce dei componenti vanno definite in anticipo, in analisi, in modo da garantire stabilità delle interfacce.

Il principio funziona perché le astrazioni contengono poco codice e quindi sono poco soggette a cambiamenti. I cambiamenti non si propagano oltre il "muro" di astrazioni (design for change) e i moduli, disaccoppiati, facilitano il riutilizzo.

(e) Single responsibility principle

R: Il principio di singola responsabilità afferma che una classe debba avere una sola responsabilità e che ci debba sempre essere al massimo un solo motivo per modificare una classe. Non appena una classe ha più di una responsabilità, esse diventano accoppiate e una modifica ad una richiede di modificare l'altra, portando a progetti fragili e poco riutilizzabili. Nel caso una classe si trovi ad avere più responsabilità, va effettuato un refactoring per separare la classe in due, assegnando a ciascuna una responsabilità.

(f) Interface segregation principle

R: Il principio di segregazione delle interfacce "estende" il principio di singola responsabilità affermando che i clienti non debbano dipendere da interfacce che non usano. Ciò significa che, al momento della scrittura di un'interfaccia, è necessario minimizzarne i contenuti: avere tante interfacce specifiche è meglio che averne una sola (*fat interface*) che fa molte cose.

6. **Design pattern.**

(a) Pattern singleton

R:

Problema: Va garantita l'esistenza di al più una istanza di una classe, alla quale deve essere possibile accedere. Se esiste già una istanza di quella classe, allora è necessario intercettare tutte le richieste di creazione di una nuova istanza.

Soluzione: Si rende privato o protected il costruttore della classe. All'interno della classe si inserisce un membro statico con lo stesso tipo della classe. Si definisce un metodo statico `GetInstance()` che crea una istanza se non è già stata creata e la restituisce.

Conseguenze: non si può chiamare `new Singleton()` e si accede all'istanza solo tramite `GetInstance()`.

Una classe statica con solo membri statici non è una buona alternativa al pattern singleton, perché quest'ultimo permette di specializzare il metodo `GetInstance()` e restituire istanze specializzate, a seconda del contesto. Inoltre, una classe singleton non statica può implementare un numero arbitrario di interfacce.

(b) Pattern observer

R:

Problema: Si ha un `Subject` le modifiche al quale devono ripercuotersi su uno o più `Observer`. Non si vuole scrivere nel codice del Subject l'aggiornamento degli Observer perché ciò richiederebbe la conoscenza reciproca, portando a un accoppiamento forte tra le parti e a codice difficile da riutilizzare.

Soluzione: Si imposta una relazione uno-a-molti tra `Subject` e `Observer`. Si definisce una clsse astratta `Subject` che dichiara i metodi `Attach()`, `Detach()` e `Notify()`. Si definisce inoltre una classe astratta `Observer` che dichiara il metodo `Update()`. Un `Observer` si registra presso il `Subject` chiamando `Attach()`. `Notify()` chiama `Update()` di tutti gli `Observer` che si sono registrati.

Conseguenze: Obbliga a implementare in tutti gli `Observer` metodi per gestitre ogni tipo di modifica, anche se a un dato `Observer` ne interessano solo alcune.

Un'alternativa al pattern observer, che però presenta alcune problematiche, è la *callback relationship*. La *class-based callback relationship* inserisce un riferimento al `Subject` in ogni `Observer` e viceversa. Questo, però, porta a una dipendenza reciproca tra i `Subject` e tutti gli `Observer`. La *interface-based callback relationship* inserisce in ogni `Observer` un riferimento a una interfaccia `IObserverEvents`, che viene implementata dal `Subject`. In questo modo `Subject` dipende da `Observer` ma non viceversa, perché `IObserverEvents` sta all'interno di `Observer`. Tuttavia, questa relazione non permette a più `Observer` di osservare uno stesso `Subject`.

(c) Pattern MVC

R: Il pattern MVC divide un'applicazione in tre componenti: modello, view e controller.

Modello: Gestisce i dati, risponde a interrogazioni e modifiche di stato, deve essere osservabile

View: Si aggiorna per visualizzare i cambiamenti nello stato del model, fornisce una vista (parziale) dei dati del Modello, deve osservare

Controller: gestisce gli input utente

(d) Pattern MVP

R: Il pattern MVP è una versione più "incapsulata" del pattern MVC, che favorisce il riuso delle view al costo di maggiore complessità.

Modello: Gestisce i dati, risponde a interrogazioni e modifiche di stato, deve essere osservabile

View: è passiva, riceve i comandi dall'input e li passa al presenter.

Presenter: gestisce gli input utente, passa al model i comandi della view e aggiorna la view.

(e) Pattern flyweight

R:

Problema: Si ha una serie di oggetti molto leggeri, ma nonostante la loro leggerezza non se ne vogliono istanziare tanti.

Soluzione: Si condividono gli oggetti, nascondendo la condivisione a tutti gli utilizzatori. Si usa un metodo factory che conserva un dizionario di Flyweight e invece di usare `new()` si chiama la factory, che cerca nel dizionario il flyweight corrispondente alla chiave e lo crea se e solo se non esiste.

Conseguenze:

- l'oggetto flyweight non deve essere distinguibile da un oggetto non condiviso (quindi non può fare ipotesi sul contesto di utilizzazione).
- i clienti non devono istanziare i propri flyweight, ma essi vanno creati tramite una factory
- va esplicitata la distinzione tra stato intrinseco (che non dipende dal contesto e può essere condiviso) ed estrinseco (deve essere memorizzato dai clienti, che lo passano al flyweight)

(f) Pattern strategy

R:

Problema: Si hanno diverse possibilità per implementare un certo comportamento, e l'utilizzo di una di queste è indifferente.

Soluzione: Si dichiara una interfaccia che dichiara i metodi che svolgono la funzionalità richiesta, e si fa dipendere il client dall'interfaccia.

(g) Pattern adapter/wrapper

R:

Problema: Si vuole convertire l'interfaccia di una classe (*adaptee*) in un'altra interfaccia che il cliente si aspetta.

Soluzione: Si scrive una classe che implementa l'interfaccia richiesta dal client e realizza le operazioni delegandone l'effettiva esecuzione a una istanza dell'*adaptee*.

(h) Pattern decorator

R:

Problema: Si hanno molti comportamenti possibili e non si vuole creare un numero di sottoclassi che cresce esponenzialmente.

Soluzione: Si scrive una interfaccia`IComponent`, implementata da`ConcreteComponent` e `Decorator`.

`Decorator` è a sua volta astratta ed è specializzata dai vari `ConcreteDecorator` e ha un'associazione molti-a-uno con `IComponent` in modo da poter avere riferimenti innestati fino ad arrivare all'operazione base.

(i) Pattern state

R:

Problema: Si vuole che un oggetto cambi comportamento a seconda del suo stato, come se cambiasse classe durante l'esecuzione, cosa non possibile nella maggior parte dei linguaggi a oggetti.

Soluzione: Si associa il contesto a una classe astratta `AbstractState`, implementata dai vari `ConcreteState`, che hanno un metodo `HandleStateChange()`.

(j) Pattern composite

R:

Problema: Si vuole rappresentare una gerarchia di contenitori e contenuti. È necessario poter navigare dal contenitore al contenuto e viceversa.

Soluzione: Si scrive una classe astratta `Component`, estesa da `Leaf` (che non può avere figli) e `Composite`, che può avere figli. Composite è quindi una composizione di `Component`. Ogni figlio ha un riferimento al genitore.

Conseguenze: Il contenitore dei figli deve essere un attributo di `Composite` (array, lista, hashtable ecc). Tutti gli elementi che hanno come genitore un componente devono essere figli, quindi diventa necessario aggiornare il padre quando si aggiornano i figli o viceversa (non entrambi per evitare un aggiornamento ciclico).

(k) Pattern visitor

R:

Problema: Si vogliono effettuare varie operazioni sugli elementi di una struttura e queste operazioni non sono necessariamente note quando definisco la struttura stessa.

Soluzione: Si definisce una classe astratta `Visitor`. Si creano nuove classi per ogni operazione, che estendono `Visitor`. Si dichiara nella struttura un'operazione per accettare un generico `Visitor`.

Conseguenze:

- Si deve prestare attenzione all'incapsulamento, perché i `Visitor` devono poter accedere allo stato della struttura.
- Per ogni elemento concreto della struttura diventa necessario dichiarare un metodo `Visit()` in tutti i visitatori concreti, quindi la modifica della struttura ha un grosso impatto sui `Visitor`. La struttura deve quindi essere stabile.
- L'operazione effettuata è di tipo *double dispatch*, perché dipende dal tipo di visitatore e visitato.

7. **Sistemi di controllo delle versioni.**

(a) Vantaggi e svantaggi del modello Lock-Modify-Unlock

R:

Con l'uso di sistemi di controllo delle versioni, si può verificare la seguente situazione:

- Utente A e B eseguono il check-out e lavorano sulle proprie working copy;
- Utente A esegue il check-in;
- Utente B esegue il check-in dopo di A, sovrascrivendo le modifiche di A.

Un modo per impedire il verificarsi di questa situazione, o almeno notificare quando avviene, è il modello Lock-Modify-Unlock (LMU), nel quale il sistema tiene traccia di chi stai modificando i vari file: ogni volta che qualcuno esegue il check-out e vuole modificare un file dichiara l'impostazione un lock su quel file, che non può essere modificato finché il lock non viene rilasciato. Quando l'utente termina di modificare il file ed esegue il check-in, il lock viene rilasciato (unlock). Questo presenta alcuni problemi:

- Notevoli ritardi se ci si scorda di eseguire l'unlock di un file;
- Serializzazione superflua: se due utenti vogliono modificare parti diverse di un file, anche se le modifiche non si sovrappongono non è permesso lavorare su parti diverse di un file;
- Falso sentimento di sicurezza: il fatto che due utenti modifichino file diversi non vuol dire che le oro modifiche siano necessariamente indipendenti. Se i due file che stanno venendo modificati dipendono l'uno dall'altro, potrebbero non funzionare come dovrebbero se le modifiche sono incompatibili;
- Richiede agli sviluppatori di essere costantemente online.

Questi problemi significano che l'unico caso in cui il modello LMU è preferibile al Copy-Modify-Merge è nel caso di file che non possono essere facilmente uniti, come ad esempio i file immagine.

(b) Vantaggi e svantaggi del modello Copy-Modify-Merge

R: Nel modello Copy-Modify-Merge (CMM) non ci sono lock, chiunque può modificare qualsiasi cosa, e al momento del check-in le modifiche effettuate dai vari utenti sono unite, non sovrascritte, tra di loro. Non è importante sapere quali sono gli altri utenti, ma solo quali modifiche hanno effettuato.

Prima di poter eseguire il check-in, il sistema si accorge che le modifiche sono state effettuate partendo da una versione precedente del file e pretende che l'utente aggiorni il proprio file con tutte le modifiche effettuate dagli altri utenti nel frattempo. Questa operazione, chiamata *merge*, può andare a buon fine o presentare conflitti: si hanno conflitti se l'applicazione delle modifiche non è simmetrica, cioè non è indifferente applicare prima o dopo le modifiche di uno qualunque degli utenti. I conflitti vanno risolti a mano. Solitamente, in ogni caso, il tempo necessario per risolvere i conflitti è sensibilmente minore del tempo che si sarebbe perso adottando un modello a lock.

Nel caso in cui due utenti effettuino modifiche che sono sintatticamente compatibili ma rompono l'interazione semantica tra più moduli, il VCS non rileva alcun conflitto. Il modello CMM non è quindi in grado di individuare i conflitti logici/concettuali/semantici.