UP (Processo Unificato)

Framework iterativo ed incrementale di sviluppo di processi software. Si divide in:

1. Inception: per descrivere il dominio dell’applicazione ad alto livello
2. Elaboration: si convalida l’architettura
3. Construction: si implementano le features del sistema
4. Transition: training sul prodotto e poi distribuzione

Tra ognuno di questi passaggi ci sono delle “milestone” ovvero dei concetti e criteri da rispettare prima di poter avanzare alla fase successiva

Quattro attributi di un’applicazione poco propensa ad adattarsi al cambiamento:

* Rigidità
* Fragilità
* Immobilità
* Viscosità

Qualità esterne al software(prestazioni e funzionalità del prodotto): correttezza, usabilità, efficienza, integrità

Qualità interne al software(proprietà intrinseche del prodotto): flessibilità, manutenibilità (manutenzione), portabilità(su diversi sistemi), riusabilità, testabilità, comprensibilità

I principi vengono usati per garantire il massimo negli aspetti qualitativi del software:

* Incapsulamento: dettagli nascosti nelle classi ma visibili tramite interfacce
* Ereditarietà: condivisione di stati e comportamenti tra le classi
* Polimorfismo: operare sulle referenze come se fossero una sola

**Principi SOLID** che ci aiutano a progettare in maniera corretta

* **Single Responsability principle** (SRP): ogni classe dovrebbe avere una sola responsabilità, incapsulata al suo interno. Non deve mai esistere più di una ragione per cui una classe debba essere modificata
* Se non rispettata porta ad un problema di fragilità quando ci troviamo di fronte ad un cambiamento di una responsabilità
* **Open closed principle**: le entità della programmazione (classi,moduli,funzioni) dovrebbero essere aperte alle estensioni ma chiuse alle modifiche. Esempio: utilizzo di una sottoclasse per specificare una superclasse, senza andare a modificare la superclasse.
* Se non rispettiamo l’OCP avremo un problema di rigidità poiché un cambiamento comporta cambiamenti a cascata e l’obbiettivo dell’OCP e avvisarci di fare refactor qualora ciò avvenga (ristrutturare un sistema cambiando la struttura interna senza toccare l’aspetto esterno e il comportamento di esso con l’utente che interagisce)
* **Liskov substitution principle**: le sottoclassi devono poter sempre essere utilizzate al posto della loro classe padre
* **Interface segregation principle**: le interfacce specializzate per ogni client sono migliori di una singola grossa interfaccia
* Non rispettarlo può portare ad un aumento della complessità e ad una potenziale violazione dell’LSP
* **Dependency inversion principle**: moduli di alto livello non devono dipendere da moduli di basso livello, entrambi devono dipendere da astrazione (una classe deve dipendere da astrazioni e non da classi concrete)
* Per migliorare la situazione bisogna pagare un prezzo a discapito di una riorganizzazione che ci rende più stabili, le classi di basso livello devono dipendere dalle astrazioni

**Design Patterns**

Soluzioni che funzionano per tutti i problemi conosciuti, sono divisi in ‘cataloghi’, i quali presentano un proprio vocabolario, sintassi e grammatica.

Ci sono 3 tipi di pattern:

* Creazionali: hanno a che fare con il problema di istanziazione di nuovi oggetti (Abstract Factory, Builder, Factory Method, Singleton)
* Strutturali: riguardano come le classi e gli oggetti sono composti o scomposti per formare una struttura coerente per risolvere il problema (Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy)
* Comportamentali: si occupano di algoritmi e di assegnazione di responsabilità tra oggetti (Chain of responsability, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor)

**GRASP** (collezione di pattern)

Metodologia di progettazione che si basa sull’attribuire responsabilità a degli elementi e a garantire che i diversi elementi dialogano tra di loro in base alle responsabilità

**Information expert**

Stabilisce che la responsabilità deve essere assegnata alla classe che ha tutte le informazioni necessarie per arrivare alla soluzione. Può succedere che nessuno abbia tutte le informazioni per completare la soluzione, perché diverse classi potrebbero essersele spartite 🡪 si fa un controllo iniziale

**Creator**

Stabilisce chi si prende la responsabilità nell’istanziare una nuova istanza dell’oggetto. Per fare ciò, pone una serie di domande ai candidati (entità) e quello con più risposte positive verrà scelto.

Regole:

1. B contiene A
2. B memorizza A in una sua variabile d’istanza
3. B utilizza a stretto contatto con A
4. B ha tutte le informazioni che servono per inizializzare A

**Controller**

Stabilisce la classe per la responsabilità di eventi di chiamata a sistema. Bisogna passare dal controllore e non in modo diretto, ci può essere un controllore per ogni classe o uno per il sistema

**Low Coupling** (pattern valutativo)

L’obbiettivo è mantenere basso il numero di dipendenze quindi bisogna valutare qual è l’elemento con meno dipendenze e sceglierlo

**High Cohesion** (pattern valutativo)

Mantiene gli oggetti focalizzati, gestibili e intelligibili in modo appropriato. Le responsabilità devono essere assegnate in modo da mantenere alta coesione tra di loro

**Pure Fabrication**

Quando dobbiamo assegnare una responsabilità e gli elementi analizzati non soddisfano i requisiti, ci fanno comodo delle classi “artificiali”, entità che esistono solo nello spazio della soluzione e che non sono legate al dominio. Si riducono le responsabilità, basso accoppiamento

**Indirection**

Per evitare il basso accoppiamento tra due elementi e aumentare le possibilità di riuso, viene assegnata la responsabilità ad un oggetto intermedio. È un’istanza della Pure Fabrication e ci permette di spostare le dipendenze. Meno responsabilità a classi concrete e più ad astratte

**Polymorphism**

La responsabilità di definizione delle variazioni dei comportamenti viene assegnata ai tipi per i quali avviene tale variazione 🡪 ci permette di attivare comportamenti diversi a seconda di ciò che gli passiamo, senza utilizzare controlli di flusso

Non dobbiamo utilizzare degli if per accedere alle sottoclassi che violerebbero l’OCP

Quando le alternative o i comportamenti sono correlati con la classe, la responsabilità del comportamento viene assegnata alle classi per le quali il comportamento varia, utilizzando operazioni polimorfe

**Protected Variations**

Protegge gli elementi dalle variazioni compiute in altri elementi, mascherandoli con un’interfaccia ed usando il polimorfismo per creare diverse implementazioni di essa

**I 23 pattern fondamentali**

In tutte le soluzioni, viene favorita la composizione a discapito dell’ereditarietà. Le due soluzioni più immediate sono:

1. Creare una superclasse andando a ereditare il metodo alle due sottoclassi, senza definirlo due volte (cosa vietata)
2. Spostare il codice condiviso in una classe nuova che però non è gerarchicamente legata alle due classi

Il problema dell’ereditarietà è che non ha chi usa la delega, anche se dobbiamo inserire le reference nelle due classi, nonostante sia più semplice ed immediato.

Il vantaggio dell’ereditarietà è che creando una sottoclasse eredito i metodi della superclasse (anche il corpo dei metodi, nonostante non ce ne sia necessità).

Ereditarietà si porta quindi dietro sostituibilità e condivisione del codice, unendo due in uno, cosa che è superflua

**GoF Template Method**

Definiamo lo scheletro di un algoritmo all’interno di una certa classe, delegando alcuni suoi passi ad una sottoclasse. In questo modo posso gestire la struttura di un algoritmo che può essere comune a più classi e che presenta piccole variazioni interne a seconda dell’oggetto che viene dato in input. L’idea è spostare il comportamento comune nella classe principale, andiamo quindi a creare un algoritmo astratto che dovrà poi essere concretizzato all’interno delle sottoclassi

* Vantaggio dal punto di vista qualificativo

**GoF Strategy**

Definiamo una famiglia di algoritmi che possono essere incapsulati e interscambiati

* Utile quando bisogna modificare dinamicamente gli algoritmi utilizzati da un’applicazione

**GoF State**

Struttura come il Strategy ma con differenze nella determinazione della classe concreta

* Permette ad un oggetto di cambiare il suo comportamento quando cambia un suo stato interno. Il vantaggio è che rinchiudiamo l’oggetto in classi esterne ed è la soluzione immediata per non scrivere comportamenti variabili all’interno della soluzione senza usare else if

**Factory**

Per creare degli oggetti senza conoscerne i dettagli ma delegando un creator.

Nella struttura viene proposto un’ulteriore elemento a cui noi deleghiamo l’istanziazione dell’oggetto. Il client usa un’interfaccia per evitare un collegamento diretto con la classe e quindi evitare un cambiamento a cascata. A noi serve un oggetto di tipo interfaccia senza dipendenza diretta.

* Si guadagna qualità ed efficienza a discapito della complessità.

**Factory Method**

* Definisce un’interfaccia per la creazione di un oggetto ma ciò permette alla sottoclasse di decidere quale classe istanziare

**GoF Observer** (per rispondere al notification problem)

Fare in modo che ogni osservatore implementi una stessa interfaccia

* Definisce una dipendenza uno-a-molti fra oggetti diversi, così che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti ad esso vengono notificati del cambiamento e possono aggiornarsi

**GoF Facade**

Viene utilizzato per nascondere la complessità del sistema e ridurre la comunicazione e la dipendenza dal Client

* Permette, attraverso un’interfaccia semplificata, l’accesso a sottoinsiemi che espongono interfacce complesse e diverse tra loro

**GoF Singleton**

* Garantisce che esista un’unica istanza di una classe e che esista un meccanismo globale in grado di accedere a tale istanza

**GoF Proxy**

Fornisce una rappresentazione di un oggetto di accesso difficile o che richiede un tempo importante per l’accesso o la creazione.

* Consente di posticipare l’accesso o la creazione, al momento in cui sia davvero richiesto

**GoF Decorator**

Risponde al problema ‘Too many dimensions’, ovvero quando si vogliono aggiungere dinamicamente funzioni o opzioni ad un oggetto

Oggetto intermediario che scherma l’oggetto reale, ha la stessa interfaccia dell’oggetto concreto.

* Consente di aggiungere metodi a classi esistenti durante il run-time (svolgimento del programma), permettendo una maggior flessibilità nell’aggiungere funzionalità a oggetti

Rispetta l’open-closed perché rende la classe aperta alle estensioni ma chiusa alla modifiche

**GoF Adapter**

Non andiamo ad utilizzare la stessa interfaccia della classe stessa perché, in caso di cambiamento, dovremmo modificare quella che è l’interfaccia, violando il DIP

* Converte l’interfaccia di una classe in un’interfaccia diversa

**GoF Bridge**

* Permette di separare l’astrazione di una classe dalla sua implementazione, così che possano variare indipendentemente. In questo modo è possibile effettuare uno switch a run-time e estendere la specializzazione delle classi. DIP e OCP

**GoF Mediator**

Se all’interno della soluzione ci sono oggetti che interagiscono tra di loro, il numero è rilevante e c’è una fitta rete di dipendenze -> modifiche a cascata.

Fa in modo che il dialogo tra gli oggetti non avviene in maniera diretta ma tramite un mediatore

**GoF Composite**

Utilizzato per realizzare una gerarchia di oggetti in cui l’oggetto contenitore può detenere oggetti elementari e/o oggetti contenitori 🡪 permette di trattare un gruppo di oggetti come se fossero l’istanza di un solo oggetto, organizza gli oggetti in una struttura ad albero nella quale i nodi sono delle composite (ruolo intermediario) e le foglie sono oggetti semplici (ruolo terminale)

**GoF Memento**

Ci sono casi in cui serve la ripristinazione dello stato di un oggetto (es. funzionalità di salvataggio o undo) ma senza violare l’incapsulamento 🡪 versione serializzata del proprio stato sotto forma di insieme opaco(non leggibile) di dati e lo memorizza

**GoF Iterator**

Fornisce un meccanismo di accesso ad una serie di oggetti senza che si espongano i dettagli su come la collezione è organizzata internamente. Iterator ha i metodi per cambiare elementi, sapere se ha finito, ecc.. (pochi metodi per scandire la collezione)

**GoF Visitor**

Separa un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa

Invece che chiedere un iteratore alla collezione, fa il contrario, applica su tutti gli oggetti il metodo passato

**GoF Builder**

Separa la costruzione di un oggetto complesso dalla sua rappresentazione, cosicché il processo di costruzione possa creare diverse rappresentazioni

Utilizzati se le istanziazioni degli oggetti sono complesse. I builder costruiscono gli oggetti in maniera distinta ma coordinata (da director). Identifica i concrete builder come implementazione di una stessa interfaccia o classe base astratta (sono tutti di tipo builder).

Se un costruttore ha molti parametri con magari valori null 🡪 si usa builder che mette a disposizione dei setter per identificare i diversi parametri

**GoF Command**

Separa l’invocazione di un comando dai suoi dettagli implementativi, così da separare colui che invoca il comando da colui che esegue l’operazione. Permette di attivare l’oggetto quando richiamato

**GoF Abstract** **Factory**

Usato per creare delle famiglie di oggetti interconnessi, senza avere la necessità di conoscere dettagli implementativi. Vengono create interfacce che consentano di lavorare con una varietà di elementi che presentano le stesse funzionalità ma con implementazioni diverse

**GoF Prototype**

Fare in modo che le nuove istanze siano copie di un prodotto e inizializzate nello stesso modo (creare un clone), non si viola l’incapsulamento perché l’oggetto stesso crea la sua copia senza esporre le informazioni

**GoF Flyweight**

Problema: elevata quantità di istanze di oggetti che condividono determinati aspetti, cercando di ottimizzare l’allocazione di memoria (è più raro adesso trovarsi di fronte a questo problema)

🡪 si creano stati fuori che vengono condivisi da tutti gli oggetti che fanno riferimento ad esso

**GoF Chain of Responsability**

Definisce il comportamento in relazione all’esecuzione di un’operazione quando essa può venir realizzata da diversi elementi 🡪 prevede una catena di oggetti che possono operare sull’operazione, i vari oggetti vengono interrogati in ordine per trovare il primo disponibile ad eseguire l’operazione (scarica barile)

**GoF Interpreter**

Crea un interprete per un linguaggio

MODERN PATTERNS

**Null Object**

Problema: reference non associate a istanze, bisogna testare che la reference non sia null

🡪 usare un ‘do nothing’ al posto di null, nessun check per null

**Hollywood Principle**

(nome da come si concludono I provini ad Hollywood: “non richiamarci ti richiamiamo noi”)

IoC = Inversion of Control

Il framework esegue gli oggetti quando è il momento opportuno (arriva la richiesta)

**Dependency Injection**

Prevede che quando viene mandato in esecuzione il codice, vengono istanziati diversi metodi che a catena hanno bisogno di altre funzionalità creando così la una rete didipendenze allocate in memoria.

Per semplificare lo sviluppo e migliorare la testabilità di software di grandi dimensioni (modularità)

**\***Togliere l’istanziazione degli oggetti dalla logica e spingerla verso l’esterno (fino al composition root)

Si ha un componente esterno che si occupa della creazione degli oggetti, delle loro relative dipendenze e di assemblarle mediante l’utilizzo dell’injection

Tipi di injection:

* Constructor: la dipendenza viene iniettata tramite l’algoritmo del costruttore
* Setter: la dipendenza viene iniettata attraverso un metodo “set”
* Interface: si basa sul mapping tra interfaccia e relativa implementanzione
* Method: istanziazione all’interno di un metodo comune

**Component based architectures**: le componenti creano le interfacce e vengono specificati il funzionamento e le interfacce necessarie. Per ogni interfaccia richiesta deve esserci un componente che la offre.

**AGILE**

Insieme di metodi di sviluppo software con dei principi comuni derivati dal “Manifesto per lo sviluppo agile del software”

Nella realizzazione del codice potrebbe nascere un problema di overhead, soprattutto se creiamo molti artefatti intermedi inutili per la realizzazione del codice

* Bisogna trovare un punto di equilibrio tra la realizzazione e l’effettiva implementazione del codice

Principi di agile

* Soddisfare il cliente attraverso contigue consegne, anche anticipato. Cliente incluso nello sviluppo, gli viene mostrato come procede il software e delle demo funzionali
* Sfruttano il cambiamento per avere vantaggio competitivo verso il cliente
* Uomini d’affari e sviluppatori lavorano insieme
* Persone del team motivate
* Dialogo faccia a faccia del team
* Costante ricerca all’attenzione, all’eccellenza tecnica e ad un buon design

PRINCIPALI PRATICHE:

Refactoring

Ristrutturare un sistema cambiando la struttura interna senza toccare l’aspetto esterno e il comportamento di esso con l’utente che interagisce

Code review

Quando il codice viene scritto, non può essere direttamente aggiunto al codebase ma deve essere revisionato da un altro membro del team 🡪 si evitano errori o sviste e si amplia la conoscenza dei membri del team (di chi fa la revisione)

Pair programming

Forma estrema di code review che ci dice che quando scriviamo codice, si giocano due ruoli interscambiabili: driver e navigator.

Non c’è tempo morto tra la conclusione del codice e la revisione di esso

Test Driven Design (TTD)

Le attività vengono invertite: si parte dal test e poi si scrive il codice stesso. Scritto il test, si compone abbastanza codice da poter passare il test(il minimo possibile) e poi con un refactor si va ad aggiungere modifiche al codice

**METODI AGILE**

XP Extreme Programming

Si basa su 4 attività: coding, testing, listening e designing

5 valori: comunicazione, semplicità, feedback, coraggio e rispetto

SCRUM

Framework in cui le persone possono affrontare i problemi complessi in maniera adattiva, ottenendo prodotti con il più alto livello possibile, attraverso produttività e creatività

Ruoli:

* Core (committed) sono le persone a tempo pieno sul processo: product owner, scrum master, development team
* Additional (involved): customer, executive manager

Artefatti:

* Product backlog: lista ordinata, a carico del produttore, contenente tutto ciò che deve essere fatto (requisiti funzionali, sistemazione di bug, apparecchi particolari per il client, chores, ecc..)
* Sprint backlog: tutto ciò che deve essere progettato dello sprint. Contiene dei task che si ottengono dal raffinamento degli elementi con priorità più alta all’interno del product backlog. Task che sono determinate dai membri del team e che devono essere completate nello sprint corrente. Si scompongono i task per fare in modo che abbiano sempre una durata inferiore alla giornata
* Burn down chart: diagramma che mostra come variano il numero di task ed il monte ore disponibile

Sprint planning è il meccanismo per definire come riempire lo sprint backlog. Solitamente avviene una runione:

1. Il product owner definisce gli obiettivi e presenta gli elementi essenziali da inserire nel prodotto
2. Il team seleziona gli elementi (task) e popola lo sprint backlog

Scrum estimation è quando andiamo a tirar fuori gli elementi con alta priorità (task) e capiamo quanti prendere

Planning poker per raggiungere un’idea comune sullo sviluppo

Velocity driven planning: story point in base alla velocità del team

Capacity driven planning: story point in base all’effort

Daily SCRUM: riunione per vedere come sta procedendo

Sprint review: riunione tra team e stakeholder per mostrare l’evoluzione del progetto

Test statici:

Non ci dicono se ci sono errori o no nel codice ma ci possono dare garanzie formali in determinate situazioni. Vengono usati per applicazioni che nascono critiche e dove non bisogna eseguire algoritmi complicati ma che comunque diano garanzie funzionanti

Test dinamici:

BLACK BOX

Quando non abbiamo conoscenza del codice e lo eseguiamo per capire quali problemi ci sono e come funziona

* I tester non sono i programmatori del codice (evita conflitto di interesse)
* Si avvicina notevolmente ai requisiti richiesti
* MA La conoscenza del codice è sconosciuta

WHITE BOX

Conosciamo la struttura interna del codice

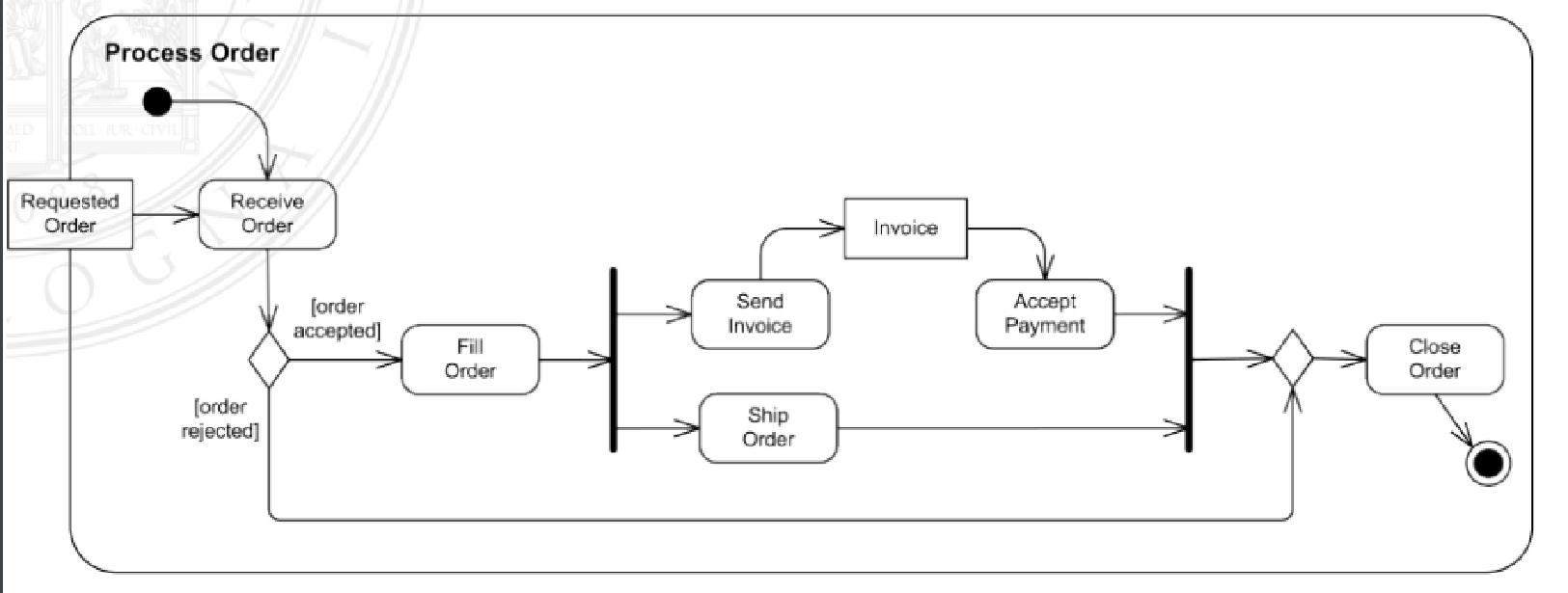
* Conoscenza del codice acquisita quando si crea il test
* Alta copertura (cercare di massimizzare il test per più righe di codice possibili)
* MA complessità (nella creazione del test)

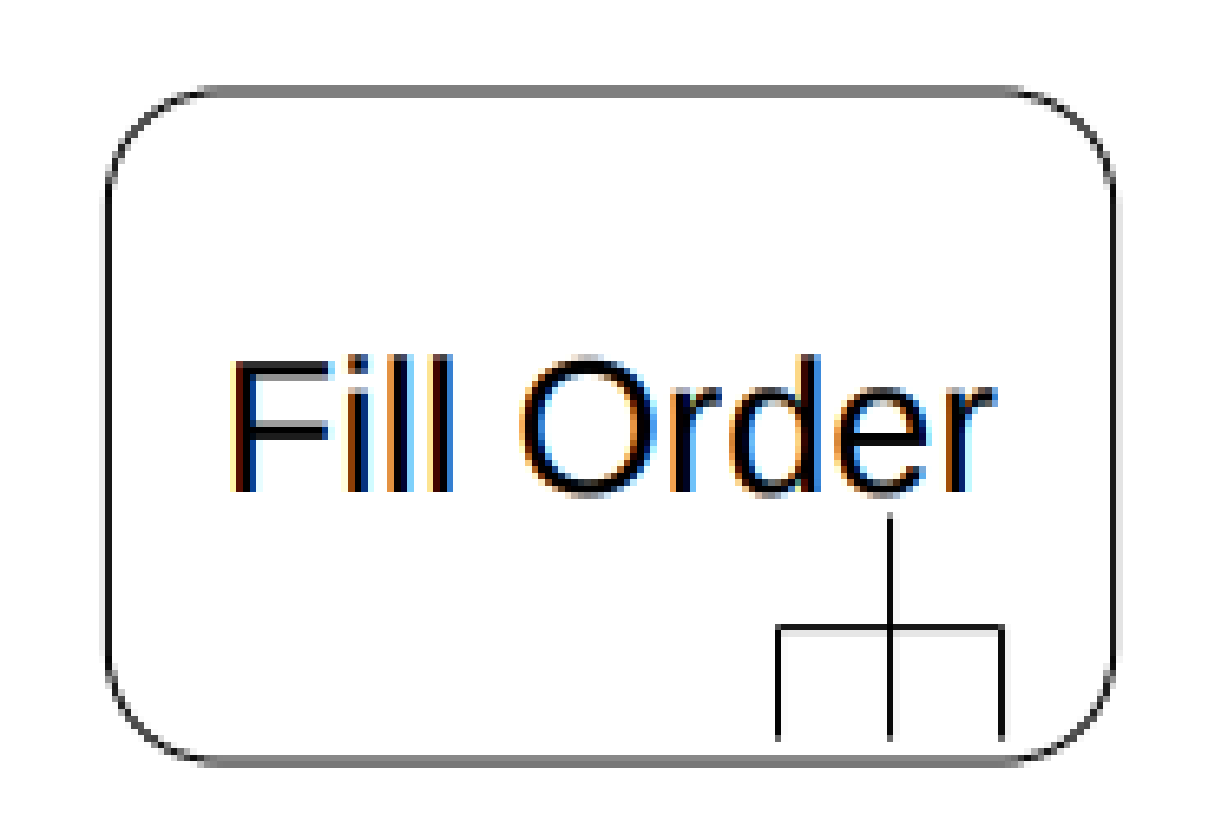
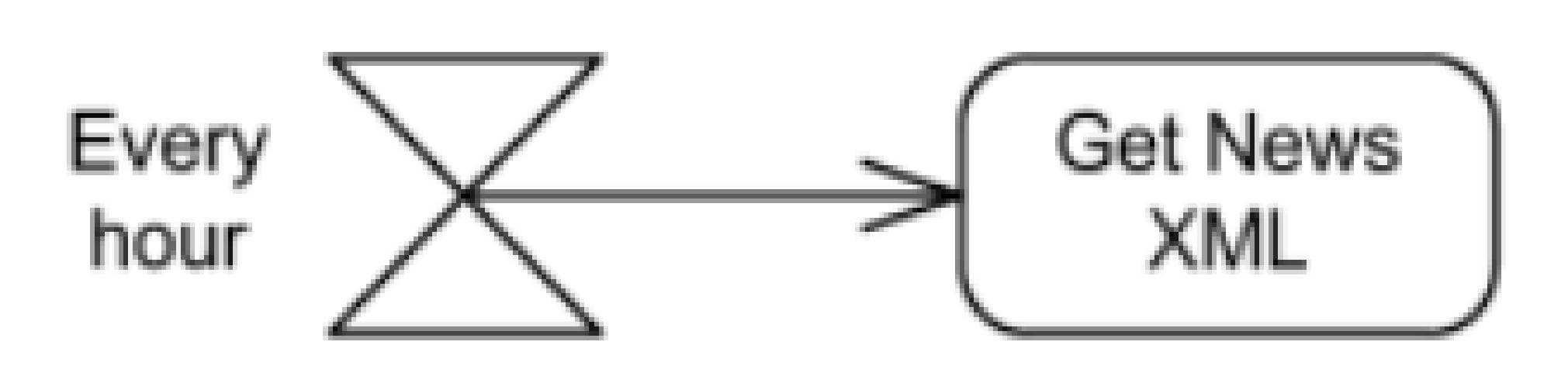
USER STORIES

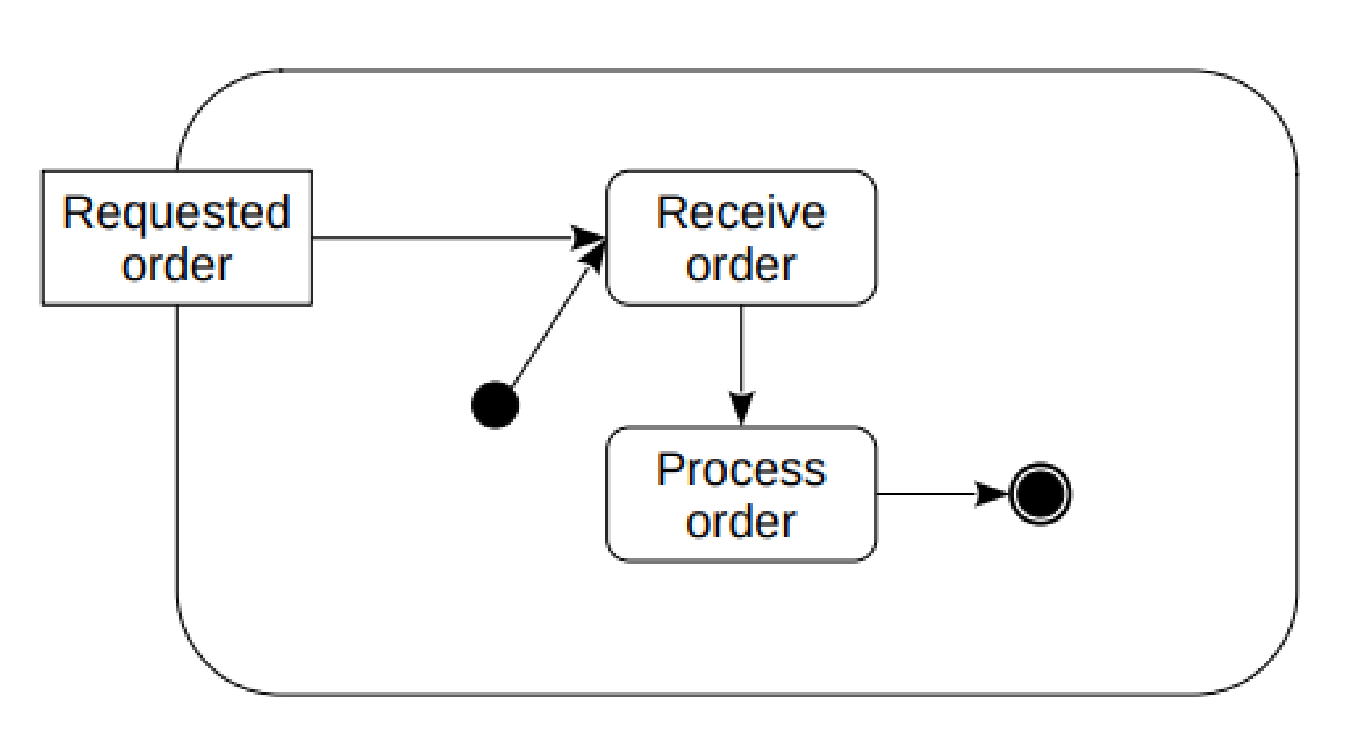
Specifiche da condividere con il team di sviluppo per stabilire cosa deve essere sviluppato e descrivono l’utente a cui ci si rivolge, il bisogno da soddisfare e il motivo per cui va soddisfatto.

**Diagramma di attività**

* I rettangoli arrotondati rappresentano delle attività
* Rombi come nodi di controllo
* Nodo nero di inizio processo
* Nodo nero con cerchio di fine processo
* Predicati tra parentesi quadre nelle frecce
* Linea nera verticale per indicare che entrambi i segnali vengono eseguiti
* Clessidra per indicare un tempo (si scrive di fianco ogni quanto)
* Call behavior action (simbolo forchetta dentro l’azione) rappresenta la chiamata di un’attività

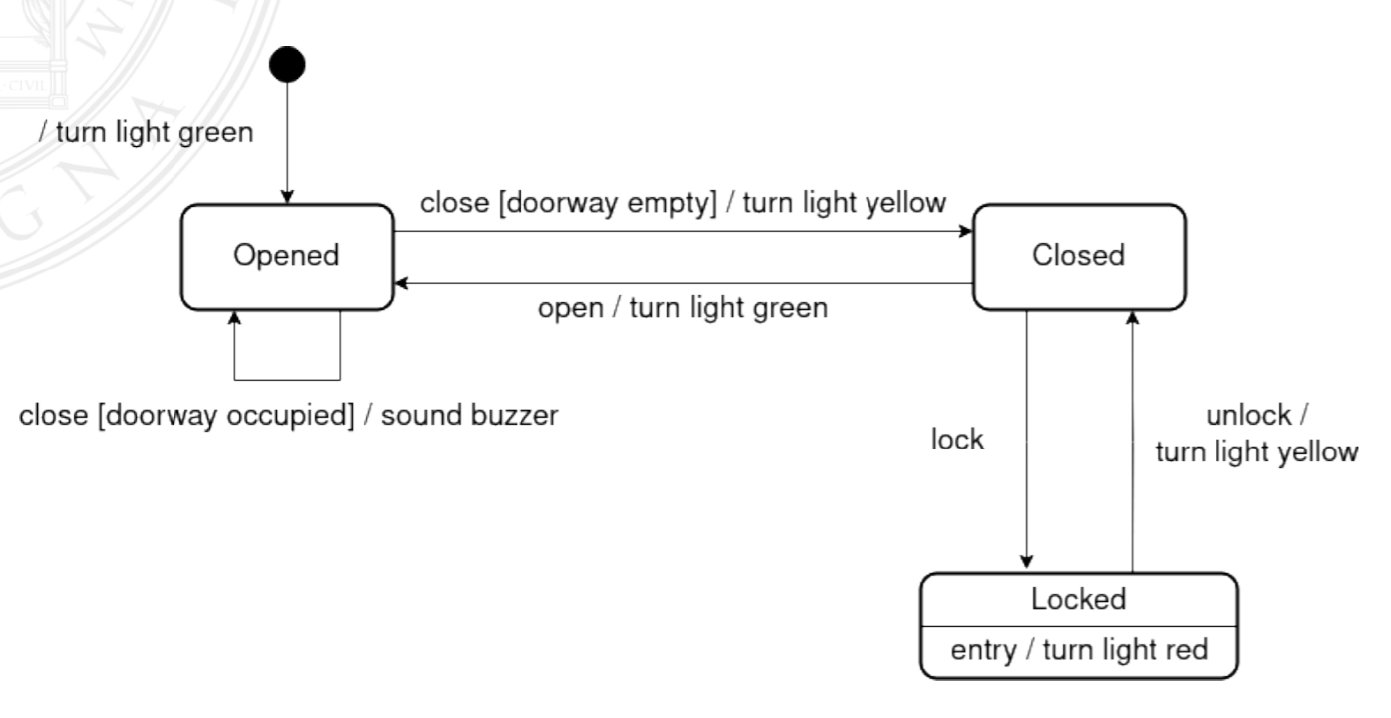






**Diagramma degli stati (Transizione degli stati di un elemento)**

* I rettangoli arrotondati rappresentano degli stati
* Le frecce definiscono le transazioni e viene scritto sopra l’evento che le provoca (trigger)
* Nodo nero di inizio processo
* Nodo nero con cerchio di fine processo



**Descrizione testuale casi d’uso:**

* Id (caso d’uso che si sta analizzando)
* Attori
* Pre condizione
* Sequenza principale
* Sequenze alternative
* Post condizione

DOMANDE INGEGNERIA

1. Design pattern GOF “observer” e legami con i principi della progettazione orientata agli oggetti

* Definisce una dipendenza uno-a-molti fra oggetti diversi, così che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti ad esso vengono notificati del cambiamento e possono aggiornarsi. Grazie ad esso, possiamo eliminare il DIP

1. GRASP pure fabrication e GOF facade e il legame concettuale che li accomuna

* Pure fabrication consente, quando gli elementi analizzati a cui assegnare la responsabilità non soddisfano i requisiti, di creare classi “artificiali” che esistono solo nello spazio della soluzione e non sono legate al dominio. Facade permette, attraverso un’interfaccia semplificata, l’accesso a sottoinsiemi che espongono interfacce complesse e diverse tra loro. Il legame che li accomuna è che l’interfaccia generata dal Facade è una facciata Pure Fabrication.

1. GOF decorator e le relazioni con i principi della progettazione orientata agli oggetti

* Risponde al problema ‘too many notifications’(quando si vogliono aggiungere dinamicamente funzioni o opzioni ad un oggetto) e consente di aggiungere metodi a classi esistenti durante il run-time. Rispetta l’open-closed principle in quanto rende la classe aperta alle estensioni ma chiusa alle modifiche.

1. GRASP creator e se ne giustifichino le regole

* Stabilisce chi prende la responsabilità di istanziare una nuova istanza dell’oggetto, ponendo varie domande ai candidati e scegliendo l’entità con più risposte positive. Le domande o regole sono:

B contiene A, B memorizza A, B utilizza A, B inizializza A

1. Factory method e il suo impatto sui principi della progettazione orientata agli oggetti

* Definisce un’interfaccia per la creazione di un oggetto senza conoscerne i dettagli ma delegando un Creator che, in base alle informazioni ricevute, saprà quale oggetto restituire. Permette di separare il Client dal Framework, così da poter modificare i dettagli senza modificare il client.

Problema del ‘new’ che può creare un collegamento diretto tra classi di alto e basso livello

1. Prinicipi SOLID e discussione di uno a scelta

* SRP (Single Responsability Principle) = ogni classe deve avere una sola responsabilità

OCP (Open Closed Principle) = le classi devono essere chiuse alle modifiche ma aperte alle estensioni (es. creare una sottoclasse per aggiungere funzioni ad una classe)

LSP (Liskov Substitution Principle) = le sottoclassi devono sempre poter essere utilizzate al posto delle classi padre

ISP (Interface Segregation Principle) = le interfacce specializzate per ogni classe sono migliori di una singola grossa interfaccia

DIP (Dependency Inversion Principle) = moduli di alto livello non devono dipendere da moduli di basso livello, ma entrambi devono dipendere da astrazioni

1. GOF state e il suo impatto sui principi della progettazione orientata agli oggetti
2. Si descrivano e discutano alcuni dei principi che ispirano lo sviluppo software agile
3. Proxy e il suo impatto sui principi della progettazione orientata agli oggetti
4. Differenze tra XP e SCRUM

* Scrum non consente cambiamenti negli sprint fino alla fine di essi, mentre XP è più flessibile. XP lavorano con una più stretta priorità, il cliente specifica su quali features lavorare per prime. I team Scrum invece determinano a proprio piacimento la sequenza degli oggetti backlog. Gli sprint Scrum durano da 2 a 4 settimane mentre le iterazioni xp da 1 a 2 settimane, sono quindi più brevi.

1. GOF strategy e il suo legame con i principi della progettazione orientata agli oggettti
2. GRASP polymorphism
3. Bridge e il suo impatto sui principi della progettazione orientata agli oggetti
4. Principi della programmazione orientata agli oggetti open-closed e dependancy inversion
5. Cosa si intende per dependency injection e che vantaggi offre nella realizzazione dei sistemi software \*\*
6. GRASP creator con un esempio significativo
7. GRASP creator e controller con esempi \*\*
8. Observer e il suo impatto sui principi della progettazione orientata agli oggetti
9. GRASP pure fabrication e GOF facade e il legame concettuale che li accomuna

Entrambi fanno uso di oggetti astratti, non legati al dominio. Entrambi riducono le responsabilità

Concetti di dipendenza accoppiamento e coesione nella progettazione orientata agli oggetti

3 qualità interne del software

* Incapsulamento: dettagli nascosti nelle classi ma visibili tramite interfacce
* Ereditarietà: condivisione di stati e comportamenti tra le classi
* Polimorfismo: operare sulle referenze come se fossero una sola

Si descrivano le principali pratiche utilizzate nelle metodologie agili per definire le attività da affrontare in un ciclo di sviluppo

* Code review:

quando il codice viene scritto, non può essere direttamente aggiunto al codebase ma deve essere revisionato da un altro membro del team 🡪 si evitano errori o sviste e si amplia la conoscenza dei membri del team (di chi fa la revisione)

* Pair programming:

forma estrema di code review che ci dice che quando scriviamo codice, si giocano due ruoli interscambiabili: driver e navigator. Non c’è tempo morto tra la conclusione del codice e la revisione di esso

* Refactoring:

ristrutturare un sistema cambiando la struttura interna senza toccare l’aspetto esterno e il comportamento di esso con l’utente che interagisce

RIASSUNTO 2.0

**Le qualità esterne** al software sono rappresentate dalle prestazioni e dalla funzionalità del prodotto che poi andranno a riflettersi sull’utente. Abbiamo visto la correttezza, l’usabilità, l’efficienza, l’affidabilità, l’integrità ovvero che i dati non vengano alterati, l’adattabilità del software su diversi sistemi operativi, l’accuratezza (più il calcolo è preciso in termini matematici, meglio è) e la robustezza ovvero ottenere risultati anche da fallimenti così da rendere il software più “robusto” una volta eliminato l’errore.

**Le qualità interne** al software sono rappresentate dalle proprietà intrinseche del prodotto, ovvero da come è organizzato il software al suo interno. Abbiamo visto la flessibilità rispetto all’aggiunta di nuove caratteristiche, la portabilità su diversi sistemi operativi, la testabilità del codice, la manutenibilità ovvero quando posso farci manutenzione sopra e la comprensibilità

**Dipendenza,Accoppiamento e Coesione**

Sono delle caratteristiche fondamentali che vengono gestite da alcuni pattern, ad esempio

High Coesion e Low Coupling sono due pattern qualitativi del GRASP che ci permettono, rispettivamente, di mantenere una forte coesione tra gli elementi che dialogano, in modo da ridurre i problemi di rigidità, e avere un numero basso di accoppiamenti, per ridurre i problemi di fragilità. Invece le dipendenze devono mantenersi minime così da evitare ad esempio cambiamenti a cascata

Le metodologie **Agile** sono dei metodi di sviluppo software con principi e pratiche comuni, alcune di quelle che abbiamo visto sono:

* Il refactoring, ovvero ristrutturare un sistema al suo interno senza modificare l’aspetto esterno e senza cambiare il comportamento di interazione con l’utente
* Code review, ovvero dopo che il codice è stato scritto, deve passare per una revisione così da correggere eventuali errori ma anche far conoscere il codice ad un altro membro del team
* Pair Programming, ovvero quando la scrittura del codice e la revisione avvengono nello stesso momento, così da evitare il tempo morto tra le due attività. È una forma di code review avanzata e i due ruoli vengono chiamati ‘driver’ e ‘navigator’
* TTD(Test-Driven-Design), ovvero quando viene fatto l’opposto di ciò che avviene normalmente, quindi viene prima scritto il test e poi il codice minimo che possa superare quel test. Dopodichè, tramite refactoring si andrà a modificare il codice aggiungendo ciò che serve.

Pattern Creazionali

**Abstract Factory** fornisce un’interfaccia per creare famiglie di oggetti interconnessi senza conoscere i dettagli di implementazione. Consente di isolare e incapsulare le classi concrete che vengono visualizzate solamente attraverso l’interfaccia creata da abstract factory

**Factory Method** permette di creare oggetti senza conoscerne i dettagli ma delegando un Creator che, tramite le informazioni ricevute, saprà quale oggetto restituire.

Siccome si basa su costruttori privati, la classe non può essere estesa quindi va in conflitto con l’OCP, infatti ogni sottoclasse dovrebbe invocare il costruttore ereditato ma non può farlo perché è privato

**Singleton** permette di avere un singolo punto di accesso all’istanza di una classe

Può essere usato per gestire una licenza software, come nel caso in cui si vogliano limitare il numero di connessioni, tramite il singleton si riesce a disporre di una sola connessione

Problemi: viola il Liskov principle perché impedisce la sostituibilità di oggetti legati da un vincolo di parentela in quanto non consente relazioni di ereditarietà

//Esempio (candidati del singleton): factories, loggers, configuration classes, remote classes, classi con attributi statici

**Builder** separa la costruzione di un oggetto complesse dalla sua rappresentazione, ciò rende più semplice la classe, permettendo ad un builder la costruzione di un’istanza e lasciando che la classe originale si concentri sul funzionamento degli oggetti

Pattern Strutturali

**Adapter** viene usato quando bisogna utilizzare sistemi la cui interfaccia non è compatibile con le applicazioni esistenti. Converte l’interfaccia di una classe in una compatibile con l’applicazione che si voleva utilizzare. È poi possibile trasformare dei dati, per cui l’adapter si occupa anche di interfacciarsi con il nuovo sistema e fornire regole per il mapping dei dati

**Bridge** permette di separare l’astrazione di una classe dalla sua implementazione effettiva, così da renderle indipendenti. Grazie ad esso possiamo effettuare uno switch durante il run-time, garantire il disaccoppiamento, nascondere l’implementazione ed estendere la specializzazione delle classi. Rispetta l’OCP in quanto va ad estendere la classe e non modificarla direttamente

**Composite** viene utilizzato per creare una gerarchia di oggetti in cui un oggetto contenitore può comprendere oggetti elementi o altri contenitori. Viene creata una struttura ad albero con delle composite ai nodi, per gestire gli intermediari, e gli oggetti semplici come foglie, in quanto hanno ruolo terminale

**Decorator** si occupa di rispondere al problema ‘Too many dimension’ ovvero quando vogliamo aggiungere funzioni o opzioni ad un oggetto in modo dinamico. Utilizza un oggetto intermedio che scherma l’oggetto reale e ha la sua stessa interfaccia, in questo modo possiamo aggiungere metodi a classi esistenti durante il run-time, così da permettere una maggior flessibilità nell’aggiungere funzionalità ad oggetti.

Rispetta l’OCP perché rende la classa aperta alle estensioni ma chiusa alle modifiche

**Facade** (fa sad) permette di nascondere la complessità e di creare un’interfaccia semplificata da cui accedere alle parti più complesse. In questo modo riduce il numero di associazioni poiché disaccoppia il client dal sistema. Rispetta il DIP perché le classi sono collegate tramite astrazione

Rispetta l’ISP, perché va a creare un’interfaccia specifica per il client

**Flyweight** ottimizza l’utilizzo delle risorse ed evita la presenza di oggetti duplicati, spesso migliora le prestazioni di un’applicazione. Si creando quindi stati esterni condivisi da tutti gli oggetti a cui si riferiscono

**Prototype** fa in modo che le nuove istanze siano copie di un prodotto e inizializzate nello stesso modo (creare un clone), non si viola l’incapsulamento perché l’oggetto stesso crea la sua copia senza esporre le informazioni

**Proxy** consente di accedere ad un oggetto complesso tramite uno semplice. Di solito viene create un’istanza di un oggetto complesso e molteplici proxy, ognuno con riferimento ad un singolo oggetto complesso, ogni operazione svolta sui proxy viene poi trasmessa all’oggetto originale.

Rispetta l’ISP perché vengono utilizzate interfacce specializzati invece che una singola e grossa interfaccia

Pattern Comportamentali

**Chain of responsability** definisce il comportamento quando un’operazione può essere eseguita da più elementi. Va a cercare il primo elemento disponibile ad effettuare l’operazione, interrogandoli in ordine, un po’ come uno scarica barile

**Command** separa l’invocazione di un comando dai suoi dettagli implementativi, così da separare colui che invoca il comando da colui che esegue l’operazione. Permette di attivare l’oggetto quando richiamato

**Interpreter** crea un interprete per un linguaggio

**Iterator** utilizzato quando si vuole accedere ad elementi di un oggetto senza modificare la sua struttura. Il pattern disaccoppia l’utilizzatore e l’implementatore dell’aggregazione di dati, tramite un oggetto intermedio, così da poter accedere agli elementi senza problemi. La conseguenza è però che l’accesso ai dati avviene tramite un’interfaccia dell’iterator

**Mediator** fa in modo che gli attori possano dialogare tramite un intermediario, in questo modo si riducono le dipendenze tra gli attori e si evitano eventuali modifiche a cascata. Si può pensare al mediator come ad un centralino che collega tutti gli attori di un sistema, così da facilitare la comunicazione e ridurre le connessioni da many to many a one to many. Il problema è che se smette di funzionare il mediator, l’intero sistema cade e gli attori non possono più comunicare

**Memento** serve a salvare lo stato di un oggetto, così da avere una versione di ripristino in caso di necessità. Crea una versione serializzata dello stato dell’oggetto sotto forma di insieme opaco di dati, ovvero non leggibile, e lo memorizza

**Observer** definisce una dipendenza uno a molti così che quando un oggetto cambia il suo stato, gli altri oggetti dipendenti da esso ricevono una ‘notifica’ del cambiamento. Risponde quindi al notification problem: domanda su come invio la notifica

Esempio: newletters, gli utenti si iscrivono inserendo il loro indirizzo email e di fronte a novità, ricevono una email di notifica

**State** permette ad un oggetto di cambiare il suo comportamento quando cambia un suo stato interno. Il vantaggio sta nel fatto che rinchiudiamo l’oggetto in classi esterne così da non scrivere comportamenti variabili all’interno della soluzione e senza usare else if. Rispetta l’OCP perché vengono definiti i comportamenti tramite un estensione della classe e non andando a modificare la classe direttamente.

**Strategy** usato per definire una famiglia di algoritmi, incapsularli (ovvero nascondere i dettagli alle classi ma renderli visibili all’interfaccia) e renderli intercambiabili.

Può capitare che vengano proliferati gli oggetti dell’applicazione e quindi si potrebbe utilizzare il pattern

**Template method** va a definire lo scheletro di un algoritmo all’interno di una certa classe, delegando alcuni passi di dettaglio alle sue sottoclassi. In questo modo posso specificare l’ordine in cui eseguire le operazioni ma lasciare che siano le sottoclassi ad implementarle. In questo modo nella superclasse viene implementato il metodo che definisce l’algoritmo mentre i vari comportamenti vengono implementati nelle sottoclassi

**Visitor** consente di definire le operazioni di un elemento senza doverlo modificare. Separa un algoritmo dalla struttura di oggetti composti a cui è applicato, così da poter aggiungere operazioni o comportamenti senza modificare la struttura stessa. Invece che chiedere un iteratore alla collezione, fa il contrario e applica su tutti gli oggetti il metodo passato in input

IoC = non è più il programmatore ad istanziare gli oggetti ma egli passa dei parametri in input ad un framework che darà poi l’output