Sviluppo Software

Daniel Biasiotto

[2022-03-03 Thu 00:08]

Contents

- Info Corso
  - Matteo Baldoni
  - Sviluppo Agile
- PDF Version

1 Software

Include:

- tutta la documentazione elettronica che serve agli utenti dei sistemi, agli sviluppatore e i responsabili della qualitá

É caratterizzato da:

- mantenibilitá
- fidatezza
- efficienza
- accettabilitá

In generale un processo descrive

- chi
- fa cosa
- come
- quando

Per raggiungere un obiettivo

Le 4 *attivitá fondamentali* comuni a tutti i processi software:

1. specifiche
2. sviluppo
3. convalida
4. evoluzione

1.1 Modello a cascata

Nel modello a cascata queste sono distinte e separate

- requisiti in dettaglio
  - non c'e' feedback, molto lavoro speculativo
- piano temporale delle attivitá da svolgere
- modellazione
- progetto software
- programmazione software
- verifica e rilascio

Parte dal presupposto che le specifiche sono prevedibili e stabili e possono essere definite correttamente sin dall'inizio, a fronte di un basso tasso di cambiamenti

- nella realtá questo non avviene quasi mai, questo modello é ottimo in caso di sistemi critici

1.2 Modello di Sviluppo Incrementale

Nel modello di sviluppo incrementale queste sono intrecciate, aggiunte di funzionalitá alla versione precedente (versioning)

- utilizzato in caso di requisiti che cambiano durante lo sviluppo
  - in molti casi se si procede progettando tutto fin dall'inizio si rischia di buttare molto del lavoro in seguito
- si implementano immediatamente le funzionalita' piu' critiche
  - per rilasciare il prima possibile: il *feedback* e' l'aspetto piu' critico
  - si procede per incrementi, *patch*
    - il codice si degrada progressivamente
    - per arginare la degradazione e' necessario un continuo *refactoring* del codice
- per il management e' piu' complesso gestire le tempistiche
  - almeno in parte puo' essere essenziale pianificare le iterazioni
- fin dall'inizio si procede con progettazione e testing del sistema

L'ambiente odierno richiede cambiamenti rapidi:

- la rapiditá delle consegne é quindi un requisito critico
- i requisiti reali diventano chiari solo dopo il feedback degli utenti

per ció questo metodo di sviluppo ha preso piede
Lo sviluppo e' organizzato in sotto-progetti

- progettazione
- iterazione
- test

Il progetto si adatta iterazione dopo iterazione al feedback, é *evolutivo*

- ogni iterazione é una scelta di un sottoinsieme dei requisiti
  - produce un sistema eseguibile e subito testabile

NB L'output di una iterazione *non* é un esperimento o un prototipo. É una sottoinsieme a livello di produzione del sistema finale.

1.2.1 Esempi

- Unified Progress
- Extreme Programming
- Scrum

1.2.2 Vantaggi

- riduzione rischi
- progresso subito visibile
- feedback immediato
- gestione della complessita', evita la *paralisi da analisi*

1.2.3 Test Driven Development

`TDD` Diversi tipi di test:

- *unitari*
  - verificano il funzionamento di singole unitá
  - struttura in 4 parti
    1. preparazione, instanziazione degli oggetti di testing e il contesto
    2. esecuzione
    3. verifica, spesso *assert*
    4. rilascio, *garbage collection*
- di *integrazione*
  - verificano la comunicazione tra parti
- *end-to-end*
  - verificano il collegamento complessivo tra gli elementi del sistema
- di *accettazione*
  - verificano il funzionamento complessivo del sistema

1.2.4 Refactoring

Strettamento legato al *testing* in un ciclo di sviluppo incrementale. A seguito di un *refactoring* vengono rieseguiti tutti i test per assicurarsi di non aver provocato una *regressione*.

Esempi di refactoring:

- *Rename*
- *Extract Method*
- *Extract Class*
- *Extract Constant*
- *Move Method*
- *Introduce Explaining Variable*
- *Replace Constructor Call with Factory Method*

1.3 Modello di Integrazione e Configurazione

Nel modello dell'integrazione e configurazione si basa su un gran numero di componenti o sistemi riutilizzabili, piccoli sistemi che vengono configurati in nuove funzionalitá

Il processo appropriato dipende dai requisiti e le politiche normative, dall'ambiente in cui il software sará utilizzato

2 Object Oriented Analysis/Design

`OOA/D`

Ai concetti vengono attribuite le *responsabilitá*, a partire da queste si passa alla progettazione e poi al software `OOD` é fortemente correlata all'*analisi dei requisiti*:

- casi d'uso
- storie utente

L'analisi si concentra sull'identificazione e la descrizione degli oggetti:

- *concetti nel dominio del problema*

Queste analisi dei requisiti sono svolte nel contesto di processi di sviluppo:

- Processo di sviluppo iterativo
- Sviluppo Agile
- Unified Process - `UP`

2.1 Unified Process

`UP`

- cerca di bilanciarsi tra estrema agilita' e pianificazione
- la versione commerciale si chiama `RUP`, di `Rational`
- iterazioni corte e timeboxed
- raffinamento graduale
- gruppi di lavoro auto-organizzati

Orizzontalmente:

- **ideazione**
  - approssimazione
  - portata
  - studio della fattibilita'
- **elaborazione**
  - visione raffinata
  - implementazione iterativo del nucreo
  - risoluzione rischi maggiori, parte piu' critica
  - implementata l'architettura del sistema, mitigazione rischi
- **costruzione**
- **transizione**

Tutte queste fasi includono analisi, progettazione e programmazione
Verticalmente si procede con:

- discipline
  - modellazione del business

  - requisiti
  - progettazione
  - implementazione
  - test
  - rilascio
- artefatti
  - qualsiasi prodotto di lavoro

In questo processo é utilizzato solo `UML`

- utilizzato solo se necessario, se viene tralasciato va indicato il motivo
- i diagrammi seguono le iterazioni e gli incrementi

Quasi tutto in `UP` e' opzionale, deciso dal project leader

2.1.1 Requisiti

Capacita' o condizioni a cui il sistema e il progetto devono essere conformi

- e' l'utente che li stabilisce, non il progettista

Possono essere

- *funzionali*
  - requisiti comportamentali
  - comportamenti del sistema
- *non funzionali*
  - scalabilita'
  - sicurezza
  - tempi di risposta
  - fattori umani
  - usabilita'

Nei processi a cascata sono molti i requisiti non utilizzati nei casi d'uso

- spreco di tempo, denaro, rischi in piu'

Per evitare questo `UP` spinge al feedback
Modello requisiti `FURPS+`

- modello dei casi d'uso
- specifiche supplementali
- glossario
- visione
- regole di business

La disciplina dei requisiti é il processo per scoprire cosa deve essere costruito e orientare la sviluppo verso il sistema corretto Si incrementalmente una lista dei requisiti: *feature list*

- breave descrizione
- stato
- costi stimati di implementazione
- prioritá
- rischio stimato per l'implementazione

Casi d'uso Catturano (in `UP` e `Agile`) i requisiti funzionali Sono descrizioni testuali che indicano l'uso che l'utente fara' del sistema

- attori; qualcuno o qualcoso dotato di comportamento
- scenario (istanza di caso d'uso); sequenza specifica di azioni e interazioni tra sistema e attori
- caso d'uso; collezione di scenari correlati (di successo/fallimento) che descrivono un attore che usa il sistema per raggiungere un obiettivo specifico

`UP` e' *use-case driven*, questi sono il modo in cui si definiscono i requisiti di sistema

- i casi d'uso definiscono analisi e progettazione
- i casi sono utilizzati per pianificare le iterazioni

- i casi definiscono i test

Il **modello dei casi d'uso** include un grafico `UML`

- e' un modello delle funzionalita' del sistema

I casi d'uso non sono orientati agli oggetti, ma sono utili a rappresentare i requisiti come input all' `OOA/D`

- l'enfasi e' sull'utente, sono il principale metodo di inclusione dell'attore nel processo di sviluppo
- questi non sono algoritmi, sono semplici descrizioni dell'interazione, non la specifica di implementazione
  - il *come* e' obiettivo della progettazione `OOD`
  - i casi descrivono gli eventi o le interazioni tra attori e sistema, si tratta il *cosa* e nulla riguardo al *come*

I casi devono essere *guidelines*, espremerle in uno **stile essenziale**. A livello delle intenzioni e delle responsabilitá, non delle azioni concrete.

Attori Sono ruoli svolti da persone, organizzazioni, sotware, macchine

- primario
- di supporto
  - offre un servizio al sistema
  - chiarisce interfacce esterne e protocolli
- fuori scena
  - ha interesse nel comportamento del caso d'uso

Formati

- breve
  - un solo paragrafo informale che descrive solitamente lo scenario principale
- informale
  - piu' paragrafi in modo informale che descrivono vari scenari
- dettagliato
  - include precondizioni e garanzie di successo

Requisiti non funzionali Possono essere inclusi nei casi d'uso se relazionati con il requisito funzinale descritto dal caso Altrimenti vengono descritti nelle specifiche supplementari

Contratti

2.1.2 Modello di Dominio

Casi d'uso e specifiche supplementari sono input che vanno a definire il modello di dominio

DEFINITION Nel `UP` il *Modello di Dominio* é una rappresentazione delle classi concettuali della situazione reale. Queste *non sono* oggetti software.

- si puó pensare come un dizionario visivo, mostra le astrazioni e le loro relazioni in maniera immediata
- non tratta le responsabilitá/metodi degli oggetti, questi sono prettamente software
- possibile distinguere:
  - **simboli**
  - **intenzioni**
    - proprietá intrinseche, definizione
  - **estensioni**
    - esempi e casi in cui la classe concettuale si applica

2.1.3 Modello di Progetto

Architettura Logica e Layer Si tratta di un modello indipendente dalla piattaforme che definisce i `layer`:

- gruppi di classi software, `packages`, sottoinsiemi con responsabilitá condivisa
  - `User Interface`
  - `Application Logic`
  - `Domain Objects`
  - `Technical Services`

I modelli per gli oggetti possono essere

- statici, definiscono (*diagrammi delle classi*)
  - package
  - nomi delle classi
  - attributi
  - firme delle operazioni
- dinamici, rappresentano il comportamento del sistema (*diagrammi di sequenza*)
  - collaborazione tra oggetti per realizzare una caso d'uso
  - i metodo delle classi software

Diagrammi dei Package Vista *statica*

Diagrammi di Interazione Vista *dinamica*

> Un interazione é una specifica di come alcuni oggetti si scambiano messaggi nel tempo per eseguire un compito nell'ambito di un certo contesto.
>
> Un compito é rappresentato da un messaggio che dá inizio all'interazione

- questo messaggio é detto *messaggio trovato*

Per questo scopo vengono usati i *diagrammi di sequenza* o i *diagrammi di comunicazione* In particolare questi sono chiamati `Design Sequence Diagram - DSD`.

Diagrammi delle Classi `Design Class Diagram - DCD` Vista *statica*

> Il diagramma delle classi di progetto é un diagramma delle classi utilizzato da un punto di vista software o di progetto.

A differenza del `Modello di Dominio` in questo contesto la visibilitá ha un significato:

- le associazioni qui hanno un verso

Progettazione a oggetti

- *Quali sono le responsabilitá dell'oggetto?*
- *Con chi collabora l'oggetto?*
- *Quali design pattern devono essere applicati?*

Si parte dal `Modello di Dominio`, ma l'implementazione impone dei vincoli ulteriori dovuti al `Object Oriented`

- vengono letti e implementati i contratti, con le loro pre e post-condizioni
- non si creano nuove associazioni nel `Modello di Dominio`: siamo a livello del codice e si fanno scelte progettuali di *visibilitá*

2.1.4 Ideazione

Si tratta dello studio di fattibilitá

- si decide se il caso merita un'analisi piú completa

La documentazione possibile é tanta ma tutto é opzionale

- va documentato solo ció che aggiunge valore al progetto

2.1.5 Elaborazione

Alla fine di questa fase si ha un'idea chiara del progetto

- vengono stipulati contratti e obiettivi chiari, temporali e sui requisiti

2.1.6 Costruzione

Durante questa fase i requisiti principali dovrebbero essere stabili

2.1.7 Transizione

3 Unified Modeling Language

`UML`

Strumento per pensare e comunicare

- utilizzato per rappresentare il modello di dominio/concettuale
- permette un passaggio piú veloce da modello a design/progettazione

  - il gap rappresentativo sará piú semplice

É un linguaggio visuale per la specifica, la costruzione e la documentazione degli elaborati di un sistema software

- de facto standard un particolare per software OO
- puó essere utilizzato come abbozzo, progetto o linguaggio di programmazione
- la modellazione agile enfatizza l'uso di `UML` come abbozzo

4 Pattern

Riassunto di esperienze precedenti, permettono di individuare le pratiche ottime nello sviluppo di progetti complessi. Un *Pattern* é una coppia *problema-soluzione* ben conosciuta e con un nome associato.

L'approccio complessivo é guidato dalla **responsabilitá**:

- `RDD` - Responsibility-Driven Development
  - **NB** quella della responsabilitá é una metafora per semplificare il ragionamento

In `UML` la responsabilitá é un *contratto* o un *obbligo* di un classificatore. Sono correlate agli obblighi o al comportamento di un oggetto, sono di due tipi:

1. di fare
   - fare qualcosa esso stesso
   - chiedere ad altri di aseguire azioni
   - controllare e controllare attivitá di altri
2. di conoscere
   - i propri dati
   - gli oggetti correlati
   - cose che puó derivare o calcolare

4.1 GRASP

`General Responsibility Assignment Software Patterns`

> Capire le responsabilitá é fondamentale per una buona programmazione a oggetti. - Martin Fowler

GRASP tratta i pattern di base per l'assegnazione di responsabilitá.

- buon blog post a riguardo

Disegnare i diagrammi di interazione é occasione di considerare le responsabilitá (metodi) e assegnarle.

La progettazione modulare é uno dei principi (`High Cohesion - Low Coupling`)

- questi sono pattern *valutativi*, non ci danno la soluzione direttamente

4.1.1 Creator

- *Chi crea un oggetto* `A`?
  - *Chi deve essere responsabile della creazione di una nuova istanza di una classe?*

Assegna alla classe `B` la responsabilitá vale una delle seguenti condizioni:

- `B` contiene o aggrega con una composizione oggetti di tipo `A`
- `B` registra `A`
  - ovvero ne salva una `reference` in un campo
- `B` utilizza strettamente `A`
- `B` possiede i dati per l'inizializzazione di `A`
  - quindi `B` é un `Expert` rispetto ad `A`

4.1.2 Information Expert

- *Chi ha una particolare responsabilitá?*

Assegna la responsabilitá alla classe che contiene le informazioni necessarie per soddisfarla.

- `Expert`

4.1.3 Low Coupling

- *Come ridurre l'impatto dei cambiamenti?*
- *Come sostenere una dipendenza bassa?*

Assegna le responsabilitá in modo tale che l'accoppiamento (non necessario) rimanga basso. Questo é un principio da utilizzare per valutare le scelte possibili e gli altri pattern.

- classi per natura **generiche** e che verranno riutilizzate devono avere un accoppiamento particolamente basso.
- il rapporto tra classi-sottoclassi é un **accoppiamento forte**
- accoppiamento alto con elementi *stabili* o *pervasivi* causano raramente problemi
  - il problema sorge con *accoppiamento alto con elementi per certi aspetti instabili*

4.1.4 High Cohesion

- *Come mantenere gli oggetti focalizzati, comprensibili e gestibili?*
  - effetto collaterale, sostenere `Low Coupling`

Assegna le responsabilitá in modo tale che la coesione rimanga alta. Questo é un principio da utilizzare per valutare le scelte possibili e gli altri pattern alternativi.

Una classe con una bassa coesione fa molte cose non correlate tra loro o svolge troppo lavoro. La coesione puó essere misurata in termini di:

- coesione di dati
- coesione funzionale
  - questa corrisponde al principio di `High Cohesion`
  - Grady Booch: c'é una coesione funzionale alta quando gli elementi di un componente *lavorano tutti insieme per fornire un comportamente ben circoscritto*
- coesione temporale
- coesione per pura coincidenza

4.1.5 Controller

- *Qual é il primo oggetto oltre lo strato* `UI` *che riceve e coordina ("controlla") un'operazione di sistema?*

Assegna la responsabilitá a un oggetto che rappresenta uno di questi:

- il sistema complessivo, un oggetto radice o entry point del software, un sottosistema principale
  - *controller facade*
- uno scenario di un caso d'uso all'interno del quale si verifica l'operazione di sistema
  - *controller di sessione* o *controller di caso d'uso*

Il `Controller` é un pattern di delega:

- oggetti dello strato `UI` catturano gli eventi di sistema generati dagli attori
- oggetti dello strato `UI` devono delegare le richieste di lavoro a oggetti di un altro strato
- il `Controller` é una sorta di *facciata*
  - controlla e coordina ma non esegui lui stesso le operazioni, secondo la `High Cohesion`

  Il controller `MVC` é distinto e solitamente dipende strettamente dalla tecnologia utilizzata per la `UI` e fa parte di questo strato, a sua volta delegerá al `Controller` dello strato di Dominio.

4.1.6 Polymorphism

4.1.7 Pure Fabrication

4.1.8 Indirection

4.1.9 Protected Variations

4.2 GoF

`Gang of Four` GoF sono idee di progettazione piú avanzate rispetto a GRASP.

- non sono proprio principi

- articoli di journaldev a riguardo

Soluzioni progettuali comuni, emengono dal codice di progetti di successo. Un fattore emerso é la superioritá della *composizione* rispetto all'*ereditarietá*:

- **Ereditarietá**
  - la sottoclasse puó accedere ai dettagli della superclasse
  - **whitebox**, a scatola aperta
  - é definita *staticamente*, non é modificabile a tempo di esecuzione
  - una modifica alla superclasse potrebbe avere ripercussioni indesiderate sulla classe che la estende
    - non rispetta l'incapsulamento
- **Composizione**
  - le funzionalitá sono ottenute tramite composizione/assemblamento di oggetti
  - riuso **blackbox**, i dettagli interni sono nascosti
  - una classe che utilizza un'altra classe puó referenziarla attraverso una *interfaccia*, questo meccanismo é dinamico
    - questa composizione tramite interfaccia rispetta l'incapsulamento, solo una modifica all'interfaccia comporterebbe ripercussioni

Questo aiuta a mantenere le classi *incapsulate* e *coese*. L'ereditarietá puó essere realizzato in due modi:

1. Polimorfismo
   - le sottoclassi possono essere scambiate l'una con l'altra
   - si utilizza una superclasse comune
   - si sfrutta *l'upcasting*
2. Specializzazione
   - le sottoclassi guadagnano elementi e proprietá rispetto alla classe base

I pattern mostrano che il **polimorfismo** e il *binding dinamico* é molto sfruttato, mentre la **specializzazione** non é comunemente utilizzata.

4.2.1 Creazionali

Riguardanti l'instanziazione delle classi

1. Abstract Factory
   - *interfaccia* factory
   - classe factory concreta per ciascuna famiglia di elementi da creare
   - opzionalmente definire una classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle factory concrete che la estendono
   - il cliente che la utilizza non ha conoscenza delle classi concrete
     - la factory si occupa di creare oggetti correlati tra loro
   - una variante crea la factory come Singleton
   - utilizzata in libreria Java per le `GUI`
2. Builder
3. Factory Method
4. Lazy Initialization
5. Prototype Pattern
6. Singleton
   - é consentita/richiesta una sola istanza di una classe
   - gli altri oggetti hanno bisogno di un punto di accesso globale e singolo al *singleton*
   - si definisce un **metodo statico** della classe che restituisce l'oggetto *singleton*
     - questo in Java
     - restituisce un puntatore all'oggetto se giá esiste, se non esiste ancora prima lo crea
       * Lazy Initialization
     - questa implementazione é preferibile
       * la classe puó essere raffinata in sottoclassi
       * la maggior parte dei meccanismi di comunicazione remota object oriented supporta l'accesso remoto solo a metodi d'istanza

        * una classe non é sempre *singleton* in tutti i contesti applicativi, dipende dalla `virtual machine`
    - il *singleton* puó essere anche implementato come **classe statica**
        - non un vero e proprio *singleton*, si lavora con la classe statica non l'oggetto
        - la classe statica ha metodi statici che offrono ció che é richiesto
    - in `UML` é indicato con un 1 nella sezione del nome, in alto a destra
    - puó esserci concorrenza in *multithreading*

7. Double-check Locking

4.2.2 Strutturali

Riguardanti la struttura delle classi/oggeti

1. Adapter
    - gestire interfacce incompatibili
    - fornire interfaccia stabile a comportamenti simili ma interfacce diverse
    - converti l'interfaccia originale in un'altra interfaccia, attraverso un *adapter* intermedio
    - da preferire l'utilizzo di un riferimento `adaptee` da parte del `Adapter`, per incapsulamento
        - questo piuttosto che *estendere* direttamente l'`Adaptee`
2. Bridge
3. Composite
    - trattare un gruppo o una struttura composta nello stesso modo di un oggetto non composto
    - si definiscono classi per gli oggetti composti e atomici in modo che implementino la stessa *interfaccia*
    - rappresenta gerarchie *tutto-parte*
    - permette di ignorare le differenze tra oggetti semplici e composti
        - saranno le differenze interne a definire le operazioni, il `client` non vede questo

- costruisce strutture ricorsive dove il cliente gestisce un'unica entitá

4. Decorator o *Wrapper*
    - permettere di assegnare responsabilitá addizionali a un oggetto dinamicamente
    - inglobare l'oggetto all'interno di un altro che aggiunge le nuove funzionalitá
        - piú flessibile dell'estensione della classe, completamente dinamico
        - evitano l'esplosione delle sotto classi
        - simile al Composite ma aggiunge funzionalitá
5. Facade
6. Flyweight
7. Proxy

4.2.3 Comportamentali

Riguardanti l'interazione tra classi

1. Chain of Responsibility
    - utilizzato nella gestione delle *eccezioni*, delega a ritroso
2. Command
3. Event Listener
4. Hirarchical Visitor
5. Interpreter
6. Iterator
7. Mediator
8. Memento
9. Observer

- oggetti *subscriber* interessati ai cambiamenti o agli eventi di un oggetto *publisher*
  - spesso associato al pattern architetturale `MVC`
- Il *publisher* vuole un basso accoppiamento con i *subscriber*
- `interface` *subscriber* o *listener*, gli oggetti subscriber implementano questa interfaccia
  - il *publisher* notifica i cambiamenti
- dipendenza **uno-a-molti**

10. State
    - il comportamento di un oggetto dipende dal suo stato
      - i metodi contengono logica condizionale per casi
    - classi *stato* per ciascun stato implementanti una `interface` comune
      - delega le operazioni che dipendono dallo stato all'oggetto stato corrente corrispondente
      - assicura che l'oggetto contesto referenzi sempre un oggetto stato che riflette il suo stato corrente
11. Strategy
    - algoritmi diversi che hanno obiettivi in comune
    - stategie come oggetti distinti che implementano una `interface` comune
12. Template method
13. Visitor
    - separare l'operazione applicata su un contenitore complesso dalla struttura dati cui é applicata
    - oggetto `ConcreteVisitor` in grado di percorrere la collezione
      - applica un metodo proprio su ogni oggetto `Element` visitato (parametro)
    - gli oggetti della collezione implementano una `interface Visitable` che consente al visitatore di essere accettato e invocare l'operazione relativa all'elemento

5 Laboratorio

Progetto `Cat & Ring`

5.1 Fase Preliminare dell'ideazione

5.1.1 Glossario

5.2 UC Dettagliati

5.2.1 Chef

- Chef Claudio, ansioso
    1. foglio riepilogativo ricette e preparazioni di tutti i servizi (automatico)
        - *opzionalmente* puó decidere di aggiungere cose al foglio (non al menú)
    2. ordina l'elenco per importanza/difficoltá (il metodo é soggettivo)
        - questo puó essere fatto anche in un momento successivo o puó essere modificato
    3. tabellone dei turni: assegna a ogni elemento dell'elenco il *turno* e un cuoco (disponibile per quel turno)
        - stima del tempo necessario a ogni cuoco
        - quantitá e porzioni
    4. revisione degli assegnamenti e dell'ordine di questi
    5. parallelamente sono creati i fogli riepilogativi dei *servizi*
- Chef Tony, rilassato
    1. fogli riepilogativi ricette e preparazioni di tutti i servizi (automatico)
    2. ordina l'elenco per giorno del servizio
    3. fogli riepilogativi dei *servizi*: assegna turno e cuoco (disponibile in quel turno)
        - segna se ci sono preparati giá pronti/avanzati da servizi precedenti
    4. tabellone dei turni: per preparazioni critiche nelle tempistiche le assegna a turni successivi

  - anche senza scegliere subito il cuoco

NB emergono due nuovi concetti:

- il **foglio riepilogativo**
  - è associato ad un servizio all'interno di un evento, e riassume le ricette/preparazioni da preparare per quel servizio, riportando per ciascuna: se è stata assegnata, a chi e quando; se non è stata assegnata perché non serve prepararla; se il compito assegnato è stato portato a termine, e in tal caso eventuali commenti a riguardo del cuoco che l'ha preparata. Solo lo chef che ha in carico un evento e i relativi servizi può modificare (aggiungendo, eliminando o cambiando) l'elenco dei compiti nei relativi fogli riepilogativi.
- il **tabellone dei turni**
  - riepiloga ciascun turno i compiti già assegnati indipendentemente dal servizio per cui sono assegnati. E' usato dallo chef per capire lo "stato" di un turno, e dai cuochi per sapere cos'hanno da fare. E' dunque pubblico; ogni qual volta uno chef modifica i compiti a partire dal proprio foglio riepilogativo, anche il contenuto del tabellone viene modificato.

Queste sono due visualizzazioni di una stessa informazione, l'utente inserirá l'informazione una volta sola.

- responsabilitá del sistema queste visualizzazioni

5.2.2 Primi UC

- Claudio
  1. crea foglio riepilogativo per un servizio di un evento **oppure** apre un foglie riepilogativo esistente (tra i servizi degli eventi di cui é stato incaricato)
  2. **opzionalmente** aggiunge preparazioni/ricette all'elenco
  3. ordina l'elenco per importanza e/o difficoltá
  4. **opzionalmente** consulta tabellone turni
  5. assegna un compito a un cuoco in un dato turno (sia sul tabellone dei turni che sul foglio riepilogativo) **oppure** modifica un assegnamento **oppure** elimina un assegnamento

6. **opzionalmente** specifica per il compito inserito nel tabellone una stima del tempo necessario
7. **opzionalmente** specifica per il compito inserito nel fogilo riepilogativo le quatitá/porzioni da preparare

ripete dal passo 4. fino a che soddisfatto

- Tony

1. crea foglio riepilogativo per un servizio di un evento **oppure** apre un foglie riepilogativo esistente (tra i servizi degli eventi di cui é stato incaricato)
2. **opzionalmente** apre piú fogli riepilogativi ripetendo il passo 1.
3. assegna compito a cuoco per dato turno (sia sul foglio riepilogativo che sul tabellone dei turni) **oppure** specifica che la ricetta/preparazione é giá pronta **oppure** assegna un compito a un turno senza specificare il cuoco
4. indica quantitá/porzioni per il compito inserito

ripete dal passo 3. fino a che soddisfatto torna al passo 2. oppure conclude

5.2.3 UC Combinato

1. Genera foglio riepilogativo **oppure** apre foglio esistente (relativo a eventi cui é incaricato)

se desidera ripete 1. per aprire piú fogli parallelamente se desidera continua con 2. altrimenti termina il caso d'uso

1. **opzionalmente** aggiunge preparazioni/ricette al foglio
2. **opzionalmente** ordina l'elenco
3. **opzionalmente** consulta tabellone dei turni
4. assegna un compito in un dato turno e **opzionalmente** a un cuoco **oppure** specifica se il compito é giá stato svolto **oppure** modifica un compito giá inserito **oppure** elimina un compito giá inserito
5. **opzionalmente** specifica tempo necessario al compito e/o quantitá/porzioni da preparare

ripete dal passo 4. fino a che soddisfatto

NB i passi 1. (per la generazione) e 4. (gestione delle 2 viste, *foglio servizio* e *tabellone turni*) sono responsabilitá del **Sistema**

5.2.4 Estensioni

5.3 Progettazione

Progettazione sullo strato di *domain*

- passaggio all'inglese per dividere il linguaggio prettamente tecnico e quello leggibile dai clienti
- domain modules
  - `MenuManagement`
  - `KitchenTaskManagement`
- *technical services*
  - persistence on `DB`
  - login

Gestione con `grasp controller` degli eventi tra `UI` e `Domain`
Il `Design Class Diagram` o `DCD`

- e' un documento unico per il progetto
  - riporta tutte le classi
- entro questo si puo' suddividere in moduli, ma questi rimangono interdipendenti tra loro
- questa e' la parte statica

Il `Detailed Sequence Diagram` o `DSD`

- la parte dinamica
- le interazioni tra gli oggetti per eseguire le operazioni necessarie
- a questo livello si vedono le chiamate e le risposte
  - e anche le notifiche tra `observed` e `observer`