Corso di Ingegneria del Software

Restaurant Management
Test Plan
Versione 1.0

Data: 20/01/2019

Partecipanti al progetto

Nome	Matricola
Andrea Cipriano	512104874
Gianmarco Cringoli	512104778
Manuel Flora	512104628

Scritto da:	Andrea Cipriano, Gianmarco Cringoli, Manuel Flora

Revision History

Data	Versione	Descrizione	Autore
19/01/19	1.0	Realizzazione template	Andrea Cipriano
19/01/19	1.1	Stesura documento	Andrea Cipriano

Indice

1. Introduzione

Il Test Plan del progetto *Restaurant Management* ha l'obiettivo di pianificare le attività di testing sul sistema che si intende realizzare, definendo l'obiettivo e l'approccio delle attività di testing, nonché lo schedule e le risorse da assegnarvi. In particolare, vengono descritti le funzionalità e le componenti di *Restaurant Management* di cui si ritiene opportuno testare il corretto funzionamento. Nel caso in cui delle attività di testing evidenziassero degli errori che possano causare comportamenti diversi da quelli attesi e che possano compromettere il buon utilizzo del sistema da parte degli utenti, quest'ultimo può essere sottoposto ad un processo di correzione degli errori individuati, fino a garantire di fornire agli utenti finali un prodotto software che rispecchi tutte le specifiche finora stabilite nelle precedenti fasi di sviluppo.

2. Relazioni con i documenti

Il Test Plan presenta diversi punti di correlazione con i documenti stilati durante le fasi precedenti dello sviluppo di *Restaurant Management*. In particolare, il presente documento fa riferimento ai requisiti funzionali descritti nel Requirement Analysis Document, ai sottosistemi individuati nel System Design Document ed alle componenti del sistema illustrate nell'Object Design Document. Ogni qualvolta si fa riferimento a qualunque di questi artefatti, si rimanda alla consultazione della relativa documentazione per ottenere informazioni più dettagliate.

3. Panoramica del sistema

Il sistema *Restaurant Management* fornisce tutte le sue funzionalità attraverso un sito web. Per assicurarsi che ciascuna funzione si comporti come previsto, bisogna quindi assicurarsi di testare ogni funzionalità offerta da ogni schermata che compone il sito. Queste, facendo riferimento alla suddivisione in sottosistemi di *Restaurant Management*, tra cui menù, camerieri, portate e la gestione della comanda. L'inserimento, la modifica e la rimozione di questi oggetti rappresentano le principali attività da testare, a cui si va ad aggiungere, naturalmente, al invio della comanda in cucina con la relativa conferma dell'ordine il quale presenta sole le portate consegnate. L'obiettivo dei test da compiere, infatti, deve concentrarsi principalmente nel dimostrare che i requisiti di affidabilità descritti nelle varie fasi di sviluppo vengano garantiti. L'importanza di questi requisiti è dettata dall'esigenza di *Restaurant Management* di permettere al gestore di organizzare nel modo migliore il proprio ristorante, il cameriere di prendere le comande nel modo più semplice e il cliente di prenotare un tavolo. L'individuazione dei test rispecchia in maniera piuttosto fedele sia i requisiti del sistema che la sua suddivisione in sottosistemi, i quali rappresentano in maniera adeguata gli obiettivi di *Restaurant Management* e le funzionalità che esso vuole offrire agli utenti. Tuttavia, alcune di queste attività vanno ulteriormente suddivise per accertarsi che ognuna di esse presenti un comportamento corretto sotto diverse situazioni, specialmente quelle al limite di validità. Nel caso in cui queste funzionalità si comportino nella maniera prevista, si può ritenere che il sistema *Restaurant Management* soddisfa gli obiettivi che si era prefissato.

4. Funzionalità da testare / non testare

4.1. Da testare

Le funzionalità da testare del sistema *Restaurant Management* sono le seguenti:

- Gestione utente:
  - Login utente (gestore, cameriere, cliente).
  - Registrazione.
  - Modifica dati personali.
- Gestione cameriere:
  - Inserimento, modifica cameriere.
- Gestione menù:
  - Inserimento, modifica menù.
  - Inserimento, modifica portata.
- Gestione comanda:
  - Inserimento, modifica portata comanda.
- Gestione prenotazione:
  - Inserimento prenotazione tavolo.
- Gestione attività:
  - Modifica dati attività.

4.2. Da non testare

Le funzionalità da non testare del sistema *Restaurant Management* sono tutte quelle riferita alla visualizzazione, e sono le seguenti:

- Visualizza prenotazioni.
- Visualizza bilancio.
- Visualizza menù.
- Invia comanda.
- Anteprima ordine.
- Conferma ordine.

5. Pass/Fail criteria

Un caso di test ha esito positivo se l'output osservato è differente dal risultato previsto dall'oracolo; al contrario, un caso di test ha esito negativo se l'output osservato coincide con il risultato previsto dall'oracolo. Pertanto, le attività di test hanno successo nei casi in cui riescono ad individuare dei comportamenti anomali nell'esecuzione delle funzionalità del sistema. Nel caso in cui uno o più casi di testi riscuotono successo, è possibile attuare un'opportuna procedura di correzione del difetto riscontrato e, successivamente, ricorrere ad un test di regressione per testare nuovamente la funzionalità modificata ed accertarsi che il problema sia stato risolto.

6. Approccio

Le attività di testing, da effettuare sul sistema si dividono in tre tipologie:

- Testing di sistema, che si occupa di testare la conformità delle funzionalità del sistema con i rispettivi requisiti funzionali e non funzionali specificati dal Requirement Analysis Document.
- Testing di integrazione, che si occupa di testare l'interoperabilità dei diversi sottosistemi, assicurandosi che il loro comportamento sia conforme a quanto specificato nel System Design Document.
- Testing d'unità, che si occupa di testare il comportamento dei singoli componenti del sistema assicurandosi che questo sia conforme alle specifiche dell'Object Design Document.
  Molte di queste attività di testing si servono di opportune classi di equivalenza per ridurre tutti i possibili input ad un numero contenuto di campioni rappresentativi dei dati.

6.1. Testing di unità

Anche in questo caso, la metodologia scelta per effettuare i test d'unità è il category partition. I vantaggi forniti da questa tecnica consistono in:

- Esplicitare le relazioni tra le variabili da testare.
- Evitare di selezionare casi di test non utili.
- Selezionare anche casi di test in cui ci siano dei legami tra le diverse variabili.

I test d'unità vengono eseguiti seguendo la suddivisione prevista per il sistema, partendo dal livello *Model* fino ad arrivare a quello *Controller;* in verità anche se le classi *Manager* sono contenute nel package *Model* esse contengono metodi i quali hanno lo stesso nome delle singole servlet. Quindi possiamo eseguire il testing partendo dal livello *Model* fino ad arrivare al quello *Manager.*
Il testing relativo a questo package prevede di testare la classe DriverManagerConnectionPool per garantire la stabilità della connessione e, successivamente, le singole classi BeanDAO. Ciascuna di queste ultime utilizza una o più istanze (fittizie) delle corrispondenti classi Bean che, tuttavia, vengono considerate come già testate visto che sono composte semplicemente dalle proprie variabili d'istanza e dai relativi metodi getter e setter associati.

Nel package Model sono presenti varie classi, quali:

- Classe DriverManagerConnectionPool.
- Classi Bean.
- Classi BeanDAO.

Per ogni classe *DAO*, quindi, vengono testati i seguenti metodi:

- UtenteBeanDAO:
  - doSave;
  - doRetrieveByKey;
  - doRetrieveByOneKey;
  - doUpdate;

    - doDelete;

- MenùBeanDAO:
    - doSave;
    - doRetrieveByKey;
    - doRetrieveByAll;
    - doUpdate;
    - doDelete;

- PortataBeanDAO:
    - doSave;
    - doRetrieveByKey;
    - doRetrieveByAll;
    - doRetrieveByCond;
    - getIdByNome;
    - doDelete;

- PrenotazioneBeanDAO:
    - doSave;
    - doDelete;
    - doRetrieveAllByKey;
    - doRetrieveAll;

- OrdineBeanDAO:
    - doSave;
    - doRetrieveByDay;
    - doRetrieveByMonth;
    - doRetrieveByYear;
    - doRetrieveAll;

Per testare le classi e i metodi elencati viene utilizzato il framework JUnit disponibile per l'ambiente di sviluppo Eclipse, già utilizzato per lo sviluppo del sistema.

Come detto in precedenza possiamo anche non testare il package Controller e testare direttamente le classi *Manager;* esse sono:

- UtenteManager.
- MenùManager.
- CamerieреManager.
- ComandaManager.
- PrenotazioneManager.
- AttivitàManager.

Tutte queste classi presentano dei metodi, i quali sono di seguito elencati:

- UtenteManager:

  - login;
  - registrazione;
  - modificaDatiPersonali;

- MenùManager:
  - inserimentoMenù;
  - modificaNomeMenù;
  - rimozioneMenù;
  - inserimentoPortata;
  - modificaPortata;
  - rimozionePortata;
  - getPortate;
  - getMenù;
  - getIdMenuByNome;
  - getPortateByMenuTipo;
  - getPortataByNome;

- CamerieреManager:
  - inserimentoCameriere;
  - modificaCameriere;
  - rimozioneCameriere;

- ComandaManager:
  - aggiornaListaTavoli;
  - creaComanda;
  - inserimentoPortataComanda;
  - rimozionePortataComanda;
  - modificaPortataComanda;
  - modificaPortataComandaStato;
  - inviaComanda;
  - inserimentoOrdine;
  - getOrdini;
  - getOrdiniByLastDay;
  - getOrdiniByLastMonth;
  - getOrdiniByLastYear;

- PrenotazioneManager:
  - prenotaTavolo;
  - rimozionePrenotazione;
  - visualizzaPrenotazioni;

- AttivitàManager:
  - modificaDatiAttività;

Infine, per il package View, non verranno effettuati casi di test specifici; questo perché, rappresentando delle semplici pagine visuali HTML o JSP, non contengono metodi che possono essere testati come per le classi precedenti. Tuttavia, gli unici test che verranno effettuati sono soltanto quelli che rappresentano i pochi metodi di validazione presenti nelle JSP.

6.2. Testing d'integrazione

La struttura del sistema *Restaurant Management* è organizzata in modo gerarchico. In questo caso, esistono le seguenti alternative per lo svolgimento del testing d'integrazione:

- Big bang.
- Bottom-up.
- Top-down.
- Sandwich.

Siccome dovremmo utilizzare sia la strategia top-down che bottom-up, è opportuno usa quella s*andwich.*

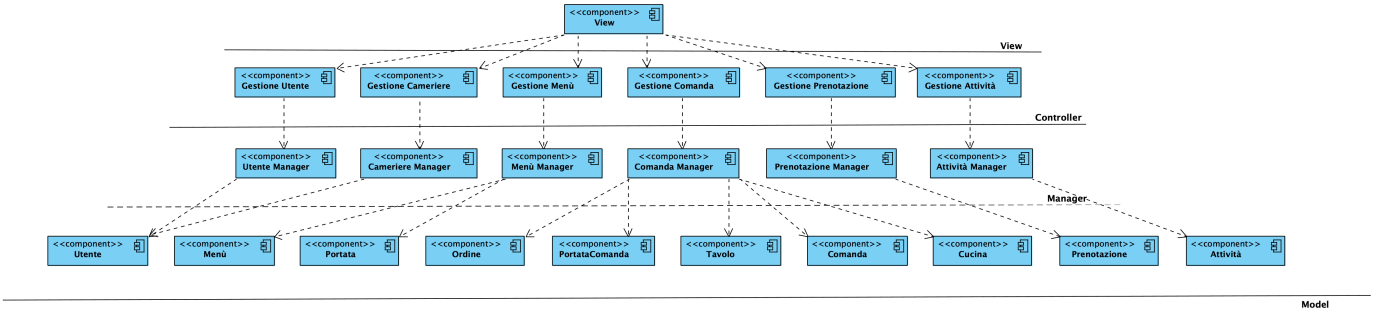
Una breve spiegazione:

Il **top-down** consiste nel testare prima i livelli superiori della gerarchia, per poi combinare tutti gli altri sottosistemi chiamati da quelli che sono stati già testati. Questo processo viene iterato finché non si arriva al livello più basso del sistema. Per questa strategia, sono necessari degli stub che riescano a simulare le componenti dei livelli inferiori. Il top-down risulta una strategia utile per i sistemi decomposti funzionalmente, anche se realizzare tutti gli stub necessari può essere piuttosto complicato nel caso in cui i livelli inferiori siano particolarmente complessi.

Il **bottom-up** consiste nel testare prima le componenti presenti nel layer più basso della gerarchia e successivamente testare tali componenti insieme ai sottosistemi presenti nel livello successivo, in base alle relative associazioni tra le componenti dei vari layer. Questo passaggio viene iterato finché non sarà testata l'intera gerarchia del sistema. In questo caso, si rende necessario utilizzare dei driver per simulare le componenti dei layer superiori che non sono ancora stati integrati. Il bottom-up risulta una strategia adatta ai sistemi object-oriented, ma non risulta molto efficiente per i sistemi decomposti funzionalmente. L'approccio stabilito per il testing d'integrazione del sistema proposto è, dunque, quello del bottom-up. Questa strategia risulta quella più utile sia perché il sistema proposto è prettamente orientato agli oggetti, sia a causa dell'elevata complessità del layer più basso (contenente lo Storage), che renderebbe l'utilizzo della strategia top-down eccessivamente complicata a causa dell'elevato numero di stub da realizzare.

Il **sandwich: c**onosciuto anche con il nome di test misti, i test a sandwich sono una metodologia di test che combina le metodologie di lavoro dei test di integrazione top down e bottom up. La bellezza di questo test è che i vantaggi di entrambe le metodologie vengono elaborati all'unisono per ottenere il massimo dalla verifica del prodotto per i difetti. È popolarmente conosciuto con il nome di test di integrazione ibridi. Dal momento che la strategia di lavoro prevede l'incorporazione di due diverse forme di test, il sistema / prodotto da testare è visto come avente tre strati.

- Uno strato di destinazione nel mezzo.
- Uno strato sopra l'obiettivo chiamato come strato superiore.
- Uno strato sotto l'obiettivo chiamato come livello inferiore.



Le singole componenti del sistema vengono innanzitutto testate singolarmente nell'attività di testing d'unità; in questi test, le componenti si servono, quando necessario, degli opportuni stub e degli opportuni driver. Successivamente, si procede ad integrare le varie componenti seguendo la strategia bottom-up. Pertanto, esse vengono testate nell'ordine seguente:

- Utente con UtenteManager.
- Menù con MenùManager.
- Portata con MenùManager.
- Prenotazione con PrenotazioneManager.
- Attività(Ristorante) con AttivitàManager.
- PortataComanda con ComandaManager.
- Tavolo con ComandaManager.
- Ordine con ComandaManager.
- Cucina con ComandaManager.
- Utente, Menù, Portata, Prenotazione, Attività(Ristorante), PortataComanda, Tavolo, Ordine, Cucina.

6.3. Testing di sistema

Le attività di testing di sistema vengono effettuate tramite un approccio black-box per testare la correttezza delle funzionalità definite a partire dai requisiti funzionali e dai casi d'uso esposti dal Requirement Analysis Document. Lo strumento utilizzato per implementare i diversi casi di test è Selenium, il quale fornisce un insieme di librerie Java utili a codificare i vari passaggi necessari a testare il sistema. Nel caso in cui le funzionalità da testare prevedessero degli input da parte dell'utente, è stata utilizzata la strategia del category partition, in cui delle classi di equivalenza partizionano opportunamente l'insieme dei possibili dati di input. I test case così formulati vengono illustrati nel dettaglio nel documento Test Case Specification -Restaurant Management.

7. Sospensione e ripresa

Le attività di testing pianificate devono portarsi fino a quando tutti i test effettuati non presentino esito negativo. Nel momento in cui un test presenta un esisto positivo evidenziando un potenziale problema, si procede pianificando e mettendo in atto un'opportuna soluzione. Successivamente, una volta eliminato il difetto rilevato, si effettua un test di regressione relativo al test precedente e a quelli strettamente correlati, per assicurarsi che la soluzione adottata abbia effettivamente risolto il problema e non ne abbia causato degli altri. In ogni caso, le attività di testing possono fermarsi se si ritiene che stiano ritardando in maniera eccessiva il completamento del progetto; in questo caso, si può evitare di risolvere completamente i test che presentano una bassa priorità.

8. Materiali di testing (requisiti hardware / software)

Le risorse che vengono utilizzate dalle attività di testing comprendono i documenti di progetto *Requirement Analysis Document - Restaurant Management*, *System Design Document - Restaurant Management* ed *Object Design Document - Restaurant Management*, a partire dai quali vengono individuate le componenti da testare, rispettivamente, nel testing di sistema, nel testing di integrazione e nel testing d'unità. Per l'esecuzione di queste attività, invece, vengono utilizzati gli strumenti Selenium e JUnit su Eclipse, insieme ad altri tool qualora sia necessario.

9. Casi di test

La specifica dei test case formulati si trova nel documento *Test Case Specification - Restaurant Management,*

10. Programma di test

Le fasi di testing verranno svolte nel seguente ordine:

1. Realizzazione Test Case;
2. Testing di unità con jUnit;
3. Test di sistema con Selenium;