

UNIVERSITÀ DEGLI STUDI DI SALERNO



**Dipartimento di Ingegneria dell'Informazione ed
Elettrica e Matematica applicata**

Corso di Laurea in Ingegneria Informatica

**APPUNTI DI INGEGNERIA DEL SOFTWARE
DI FRANCESCO PIO CIRILLO**

<https://github.com/francescopiocirillo>



"Non ti abbattere mai"

 Ehi, un attimo prima di iniziare!

Hai appena aperto una raccolta di appunti che ho deciso di condividere **gratuitamente** su GitHub, se ti sono utili fai **una buona azione digitale**:

-  **Lascia una stellina alla repo:** è gratis, indolore e fa super piacere!
-  **Condividerla con amici**, compagni di corso, o chiunque possa averne bisogno.

Insomma, se questi appunti ti salvano anche solo una giornata di studio... fammelo sapere con una **stellina!**

Grazie di cuore 

Testing

TESTING CONCEPTS

Nel contesto della qualità del Software possiamo parlare di:

- **Quality Assurance** (QA) (assicurazione della qualità): attività progettate per misurare e migliorare la qualità in un prodotto o in un processo {Fra: "è una cosa proattiva, che si fa a monte del lavoro"};
- **Quality Control** (QC) (controllo qualità): attività progettate per validare e verificare la qualità del prodotto rilevando malfunzionamenti e "correggendo" i difetti.



Possiamo definire la qualità come conformità ai requisiti e idoneità all'utilizzo.

Le attività di Quality Control sono la **Verifica** (il software è conforme ai requisiti? Il sistema è corretto?) e la **Validazione** (il Software viene in conto alle necessità degli utenti? Stiamo costruendo il sistema giusto?).

Failure/Erroneous State/Fault/Error



Un **Failure/Problem** (Fallimento) è una qualsiasi deviazione del comportamento osservato rispetto a quello specificato.



Uno **Stato Erroneo** significa che il sistema è in uno stato tale per il quale future operazioni porteranno sicuramente ad un Failure.



Un **Fault/Defect/Bug** (Difetto) è una condizione che potrebbe causare uno Stato Erroneo (o potrebbe continuare a funzionare).



Un **Error** fatto dal Software Engineer o dal Programmatore, è causa di un Fault.

Affidabilità (reliability) del Software



L'affidabilità è una misura del successo con il quale il comportamento osservato si conforma alle specifiche del suo comportamento.

Nel contesto del software possiamo definire l'affidabilità come la probabilità che un sistema software non causerà Failure per una certa quantità di tempo sotto specifiche condizioni.

Si può aumentare l'affidabilità di un sistema software in diversi modi:

- **Fault Avoidance** (evitamento del Fault): si prova a prevenire l'insorgere di Faults nel sistema prima che questo venga rilasciato.

Per ottenere questo obiettivo si può fare uso di molti Tools quali metodologie di sviluppo, design patterns, tecniche di refactoring and so on.

La Fault Avoidance asserisce al campo della Quality Assurance.

- **Fault Tolerance**: si decide di rilasciare un sistema con dei Faults. I Failures sono risolti a runtime.

Questa pratica fa uso di diverse tecniche quali l'handling delle exceptions, l'utilizzo della ridondanza and so on.

La Fault Tolerance asserisce alla Robustezza del Sistema.

- **Fault Detection**: vengono svolti degli esperimenti per identificare Stati Erronei e trovare i Faults che li causano prima di rilasciare il sistema.

Si fa Fault Detection durante lo sviluppo ma in alcuni casi anche dopo la release del sistema (è l'esempio del messaggio "invia report dell'errore" quando crasha un programma).

Questa pratica fa uso di diverse tecniche quali reviews del software e testing del software.

La Fault Detection asserisce al Quality Control.

Software Reviews

Una review del Software è l'ispezione manuale del sistema (di alcune o tutte le sue parti) senza eseguirlo. Di solito l'85% dei Faults sono trovati durante le code reviews.

- **Walkthrough** (procedura dettagliata) → Lo sviluppatore presenta informalmente la API, il codice e la documentazione del componente al team di review, di seguito il team di review fa dei commenti sulla mappatura dei requisiti e sull'architettura rispetto al codice.
- **Inspection** (ispezione) → Simile al walkthrough ma allo sviluppatore non è permesso presentare l'artefatto. Lo sviluppatore è solo presente in caso la review necessiti chiarificazioni.

Software Testing

La definizione debole è che il testing è il processo di dimostrare che non sono presenti Faults.

Tuttavia, dimostrare che non ci sono Faults non è possibile in sistemi di dimensioni realistiche, il testing può solo evidenziare la presenza di bugs ma non può dimostrare la loro assenza.

 La definizione forte è che il testing è l'insieme dei tentativi sistematici svolti in modo pianificato di trovare Faults in un software implementato allo scopo di correggere i faults e migliorare l'affidabilità del sistema.

È necessario per il testing che gli sviluppatori siano disposti a "smantellare" il loro lavoro.

La maggior parte delle attività di software development sono costruttive ma il Testing richiede un modus operandi diverso, la parola successo ha una accezione negativa durante il testing: successo è quando viene trovato un Fault.

Il sistema software si considera pronto alla consegna quando nessuno dei test svolti si dimostra in grado di romperne il funzionamento rispetto ai requisiti.

Concetti di Testing

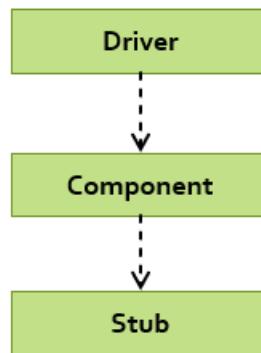
Test Component (componente di test): una parte del sistema che può essere isolata per il testing, può essere un oggetto, un gruppo di oggetti o uno o più sottosistemi.

Test Case: un set di input e di risultati attesi che "esercitano" un componente di test al fine di causare Failures e rilevare Faults.

Correzione: un cambiamento ad un componente mirato alla riparazione di un Fault (può però introdurre più Faults).

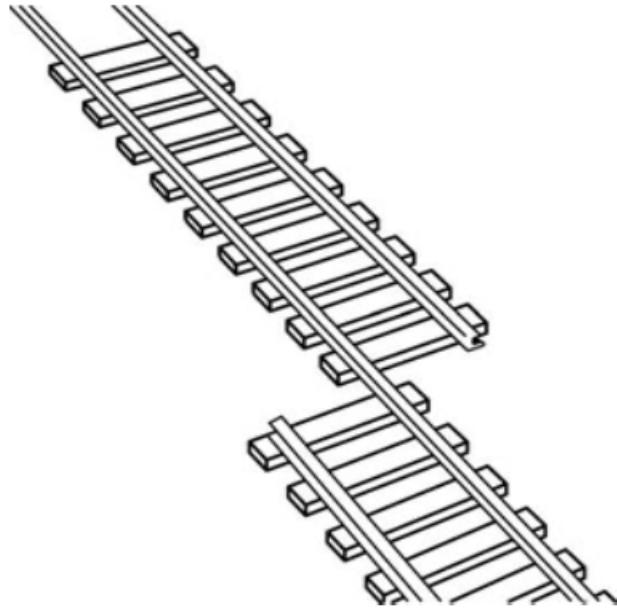
Test Stubs (ceppi di test): una implementazione parziale di componenti di basso livello dai quali le componenti testate dipendono.

Test Driver: un'implementazione parziale di un componente che dipende dal test component.



I Test Stubs e i Test Drivers permettono di isolare delle componenti dal resto del sistema al fine di svolgere il testing.

▼ Esempio



Cos'è? Un Failure? Uno Stato Erroneo? Un Fault?

Per stabilirlo bisogna comparare il comportamento atteso descritto nello use case e il comportamento osservato descritto dal test case.

Esempio di Use Case:

<i>Use case name</i>	DriveTrain
<i>Participating actor</i>	TrainOperator
<i>Entry condition</i>	TrainOperator pushes the "StartTrain" button at the control panel.
<i>Flow of events</i>	<ol style="list-style-type: none">1. The train starts moving on track 1.2. The train transitions to track 2.
<i>Exit condition</i>	The train is running on track 2.

Esempio di test case che porta il treno dalla Entry Condition dello Use Case ad uno stato nel quale si schianterà:

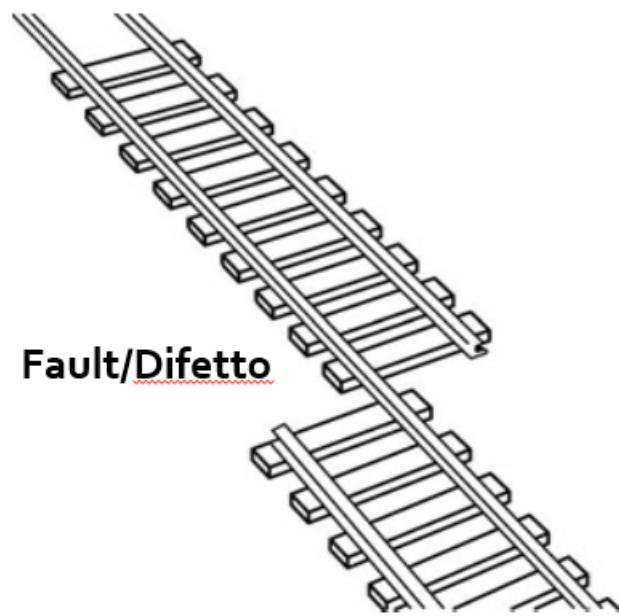
name	DriveTrainTest
location	http://www12.in.tum.de/TrainSystem/test-cases/test1
input	The StartTrain() method is called via a test driver StartTrain (contained in the same directory as the test case) Direction of trip and duration are read from a input file http://www12.in.tum.de/TrainSystem/test-cases/input The test stub Engine is needed for the test execution
oracle	The test passes if the train drives for 5 seconds and covers the length of at least two tracks

Ora possiamo dire che:



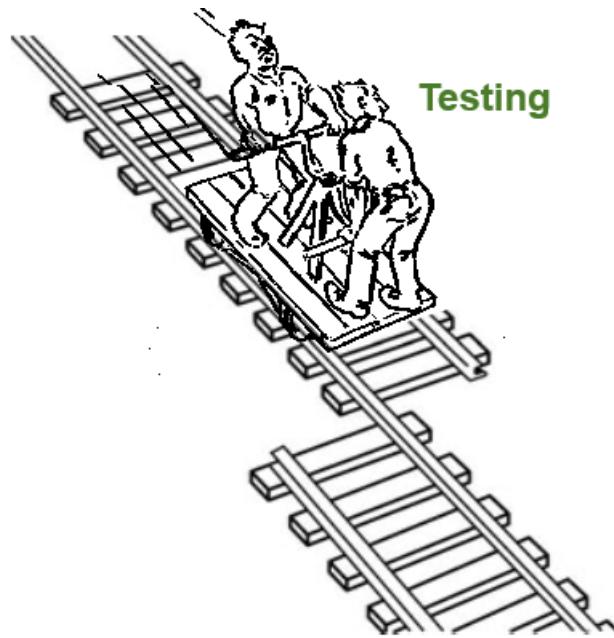


Stato Erroneo



Fault/Difetto

E infine:



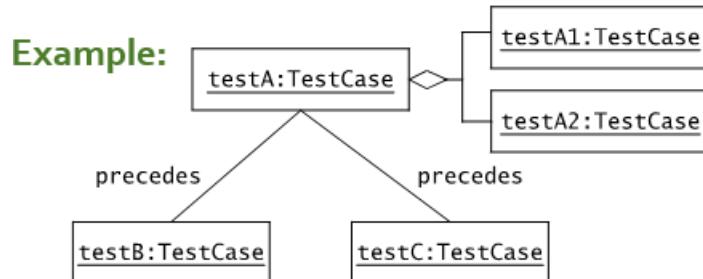
Test Cases

Un Test Case può essere espresso come un insieme di attributi

Attributes	Description
name	Name of test case
location	Full path name of executable
input	Input data or commands
oracle	Expected test results against which the output of the test is compared
log	Output produced by the test

Una volta identificati e descritti i casi di test è possibile identificare le relazioni tra questi:

- **Aggregazione:** usata quando un caso di test può essere decomposto in un insieme di sottotest;
- **Associazione Precedes (precede):** usata quando un caso di test deve precedere un altro.



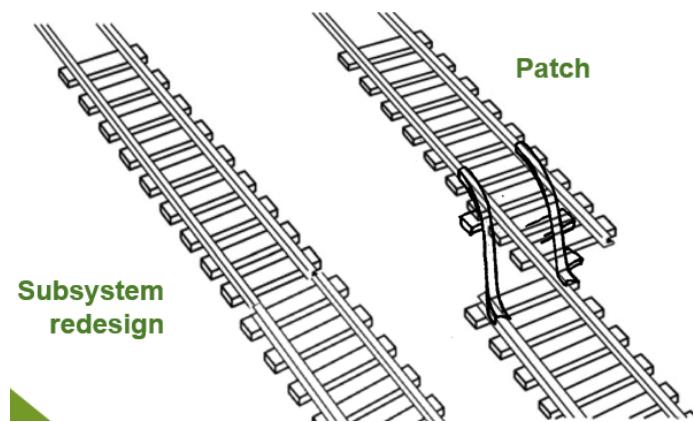
Sulla base di quale aspetto del sistema viene analizzato i casi di test sono classificati in:

- **test black box** → si concentrano sul **comportamento input/output** del componente e non si occupano degli aspetti interni. (par.10.3: derivati dai requisiti)
- **test white box** → si concentrano sulla **struttura interna** del componente e si assicurano che ogni stato dell'oggetto e ogni interazione tra oggetti venga testata. (par.10.3: derivati dal codice e dal design)

Correzioni

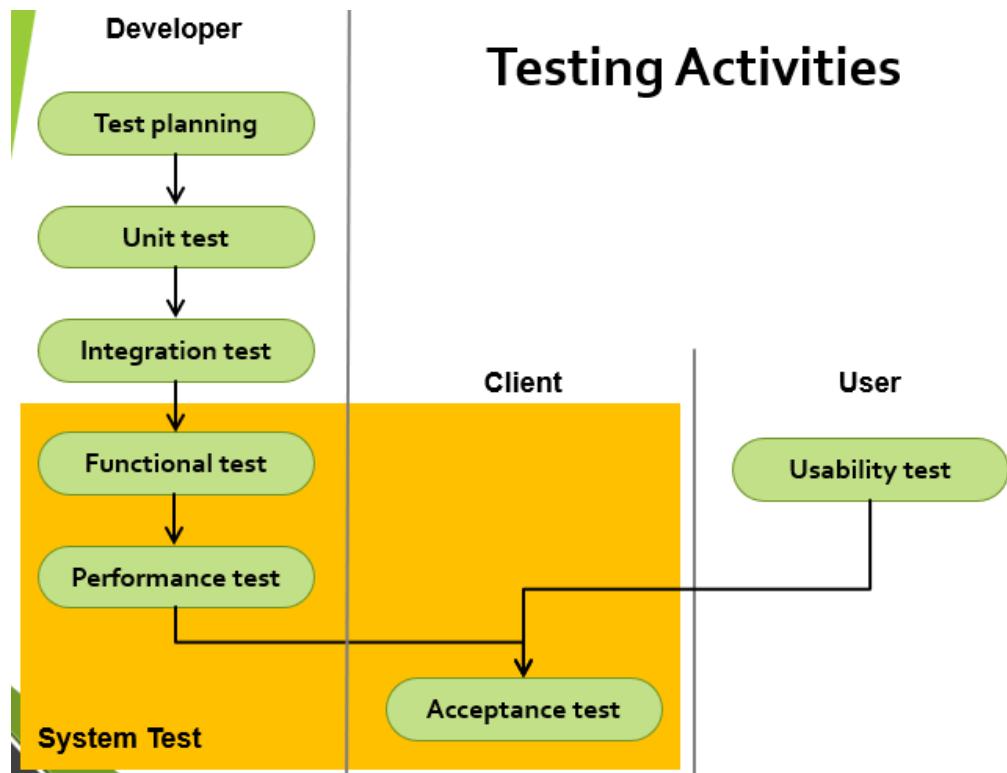
Una volta identificati i fallimenti (failures) gli sviluppatori cambiano il componente al fine di eliminare i difetti (faults) "sospettati".

Questa operazione può andare da una semplice modifica ad un componente ad un completo redesign della struttura dati del sottosistema.



La probabilità che lo sviluppatore introduca nuovi Faults (difetti) nel componente revisionato è alta ed è per questo che è importante il **Regression Testing** vale a dire la riesecuzione dei test svolti precedentemente al cambiamento fatto, questo assicura che le features che funzionavano prima della correzione continuano a funzionare.

RIASSUNTO DELLE ATTIVITÀ DI TEST



Le Attività di Test possono essere divise in:

- **Test Planning** (pianificazione del test): consiste nell'allocazione delle risorse e nella schedulazione del testing; gli sviluppatori dovrebbero progettare i Test Cases appena i modelli che validano diventano stabili.
- **Usability testing** (test di usabilità): consiste nel provare a trovare Faults (difetti) nel progetto della UI del sistema; spesso i sistemi non riescono a raggiungere gli obiettivi prefissati perché la UI confonde gli utenti. TROPPO COSTOSO
- **Unit testing**: consiste nel cercare Faults (difetti) negli oggetti e/o nei sottosistemi partecipanti rispetto allo Use Case.
- **Integration testing**: consiste nel cercare Faults (difetti) testando componenti individuali in combinazione ad altri componenti. NON FORMALIZZATO MA FATTO COMUNQUE
- **System testing**: si testano tutte le componenti insieme come un unico sistema, include:
 - **Functional testing** (testing funzionale): controlla i requisiti funzionali;
 - **Performance testing**: controlla i requisiti non funzionali; NON NE AVEVAMO
 - **Acceptance testing**: controlla il sistema rispetto agli accordi relativi al progetto (fatto dal cliente).

Unit Testing

Si concentra sui "mattoncini" che compongono un sistema, vale a dire oggetti e sottosistemi (i sottosistemi dovrebbero essere testati solo dopo che ogni loro componente è stato testato

individualmente). Si sceglie cosa testare tramite il class diagram.

Lo Unit Testing permette di **lavorare in parallelo** e focalizzandosi su unità piccole riduce la complessità delle attività generali di test. Assumendo di concentrarsi su poche componenti lo Unit Testing facilita la ricerca e la correzione di Faults.

Esistono quattro tecniche principali di Unit Testing:

1. Testing di Equivalenza (Equivalence Testing);
2. Testing dei Confini (Boundary Testing);
3. Testing dei Percorsi (Path Testing); NON CI SERVIVA
4. Testing del Polimorfismo (Polymorphism Testing). NON CI SERVIVA

1. Equivalence Testing

È una tecnica di testing Black-Box che minimizza il numero di Test Cases dividendo i possibili input in **"Classi di Equivalenza"**. Vengono usati solo pochi membri da ogni classe usata per il testing.

L'assunzione è che il sistema si comporti in maniera simile per tutti i membri di una classe.

Il testing di equivalenza prevede due passi:

1. Identificazione delle classi di equivalenza;
2. Selezione degli input di test.

Per determinare le **classi di equivalenza** si seguono tre criteri:

- **Copertura** (Coverage): ogni possibile input appartiene ad una delle classi;
- **Disgiuntura** (Disjointedness): nessun input appartiene a più di una classe;
- **Rappresentazione**: se l'esecuzione dimostra uno Stato Erroneo quando un particolare membro di una classe è utilizzato, allora lo stesso stato erroneo può essere rilevato usando un qualsiasi altro membro.

Per ogni classe sono selezionati almeno due input:

- Un input **tipico**, che esercita il caso comune;
- Un input **invalido**, che esercita l'handling di una exception.

▼ Esempio: testare un metodo che restituisce il numero di giorni in un mese, dato il mese e l'anno.

```
class MyGregorianCalendar {  
    ...  
    public static int getNumDaysInMonth(int month, int year) {...}  
    ...  
}
```

Classi di equivalenza per il parametro "month":

- Mesi con 31 giorni (1, 3, 5, 7, 8, 10, 12);
- Mesi con 30 giorni (4, 6, 9, 11);

- Febbraio, che può avere 28 o 29 giorni.

Valori invalidi: interi non positivi e interi più grandi di 12.

Classi di equivalenza per il parametro "year":

- Anni bisestili (2020, 2024, 2028 ecc...)
- Anni non bisestili.

Invalid values: interi negativi.

I Test Cases sono ottenuti selezionando un valore valido per ogni classe di equivalenza e combinando gli input sono i seguenti:

Equivalence class	Value for month input	Value for year input
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Month with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non-leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

2. Boundary Testing

Il Testing dei Confini è un caso speciale di test di equivalenza nel quale gli elementi sono scelti dai "margini" delle classi di equivalenza, l'assunzione è che gli sviluppatori spesso non si concentrano abbastanza sui casi speciali.

▼ Esempio:

- Gli anni che sono multipli di 4 sono anni bisestili;
- Gli anni che sono multipli di 100, però, non sono anni bisestili a meno che non siano anche multipli di 400;
- Il 2000 fu un anno bisestile ma il 1900 no;
- Il 2000 e il 1900 sono buoni casi "boundary" "confine" per il parametro "year";
- 0 e 13 sono ai margini della classe di equivalenza per il parametro "month".

Casi boundary addizionali per il metodo getNumDaysInMonth():

Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

3. Path Testing

In questa fase si identificano i Faults (difetti) nell'implementazione del componente, è una tecnica di testing White-Box. Il Path Testing richiede conoscenza del codice sorgente e delle strutture dati utilizzate.

L'assunzione è che, esercitando tutti i possibili percorsi attraverso il codice almeno una volta, la maggior parte dei Faults causeranno dei Failures (rendendosi visibili).

Il punto di partenza è il **flow graph** (penso diagramma di flusso), un flow graph consiste di nodi (nodes) che rappresentano blocchi eseguibili e di edge (margini) che rappresentano i flussi di controllo. Corrisponde all'activity diagrams in UML ed è costruito a partire dal codice.

- ▼ Esempio: implementazione di getNumDaysInMonth() con dei Faults.

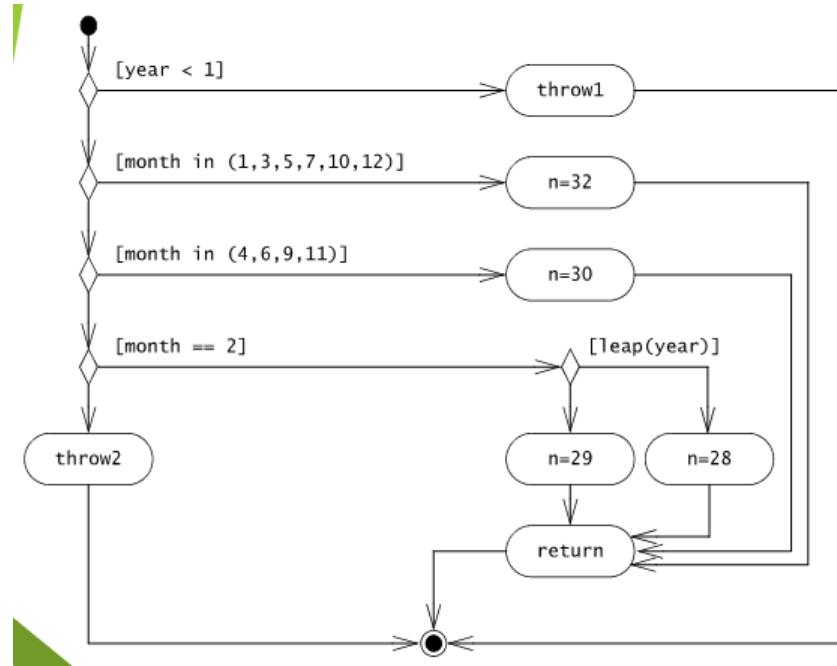
```

public class MonthOutOfBoundsException extends Exception {...};
public class YearOutOfBoundsException extends Exception {...};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
        throws MonthOutOfBoundsException, YearOutOfBoundsException {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBoundsException(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
            month == 10 || month == 12) {
            numDays = 31;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBoundsException(month);
        }
        return numDays;
    }
}

```

Da cui ricaviamo il seguente flow graph:

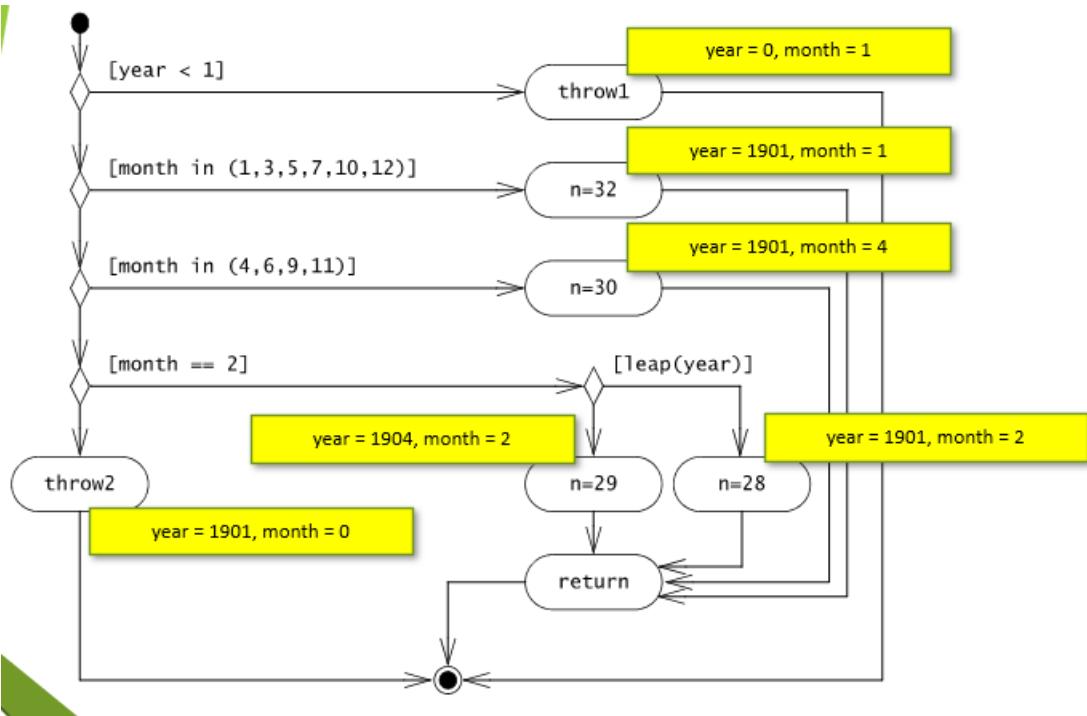


Vanno poi definiti i Test Cases:

ogni edge nell'activity diagram deve essere attraversato almeno una volta e per ogni condizione bisogna selezionare un input per il ramo (branch) vero e un altro per il ramo falso.

Test case	Path
(year = 0, month = 1)	{throw1}
(year = 1901, month = 1)	{n=32 return}
(year = 1901, month = 2)	{n=28 return}
(year = 1904, month = 2)	{n=29 return}
(year = 1901, month = 4)	{n=30 return}
(year = 1901, month = 0)	{throw2}

E infine:



NOTA 1:

- L'implementazione di `isLeapYear()` non prende in considerazione gli anni divisibili per 100;
- Il Path Testing può solo rilevare Faults (difetti) che si ottengono esercitando un certo path nel programma e NON può rilevare omissioni.

NOTA 2:

- Nessun metodo di testing può garantire la scoperta di tutti i Faults;
- Né l'Equivalence Testing né il Path Testing possono rilevare il fault legato al mese di Agosto.

```

public class MonthOutOfBoundsException extends Exception {...};

public class YearOutOfBoundsException extends Exception {...};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
        throws MonthOutOfBoundsException, YearOutOfBoundsException {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBoundsException(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
            month == 10 || month == 12) {
            numDays = 31;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBoundsException(month);
        }
        return numDays;
    }
}

```

NOTE 1

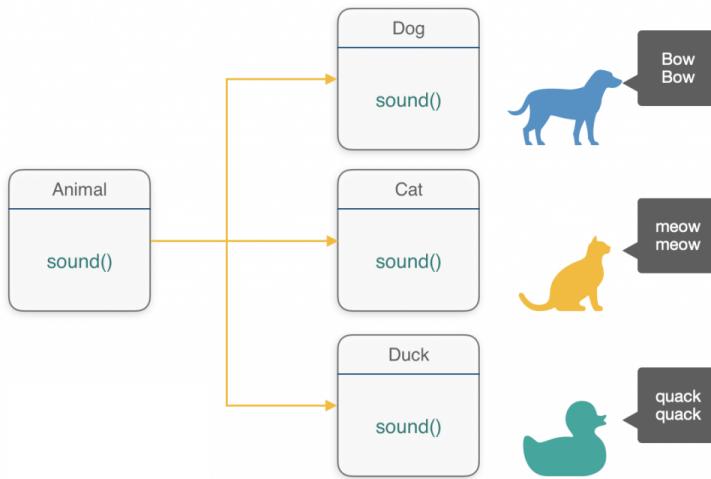
NOTE 2



4. Testing del Polimorfismo

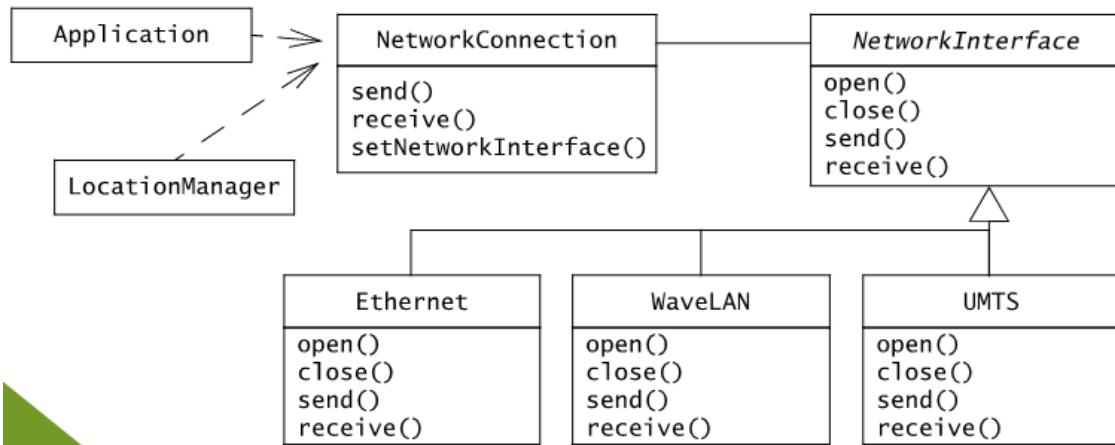
Il polimorfismo introduce una nuova difficoltà nel testing. Il polimorfismo permette a dei messaggi di essere legati a diversi metodi sulla base della classe "target" e questo introduce più casi di test.

Tutti i possibili bindings (legami) dovrebbero essere identificati e testati.



▼ Esempio:

- Il polimorfismo è usato per schermare NetworkConnection dalla strategia concreta (Ethernet, WaveLAN, UMTS);
- NetworkConnection.send() chiama la NetworkInterface.send() per inviare dati;
- A run-time, l'invocazione di NetworkInterface.send() può essere legata a Ethernet.send(), WaveLAN.send(), UMTS.send().



Quando si applica il Path Testing abbiamo bisogno di considerare tutti i possibili bindings (legamenti), è un po' come espandere il codice sorgente con degli if-else innestati per ogni sottoclasse (intesa qui come classi che implementano) di NetworkInterface.

```

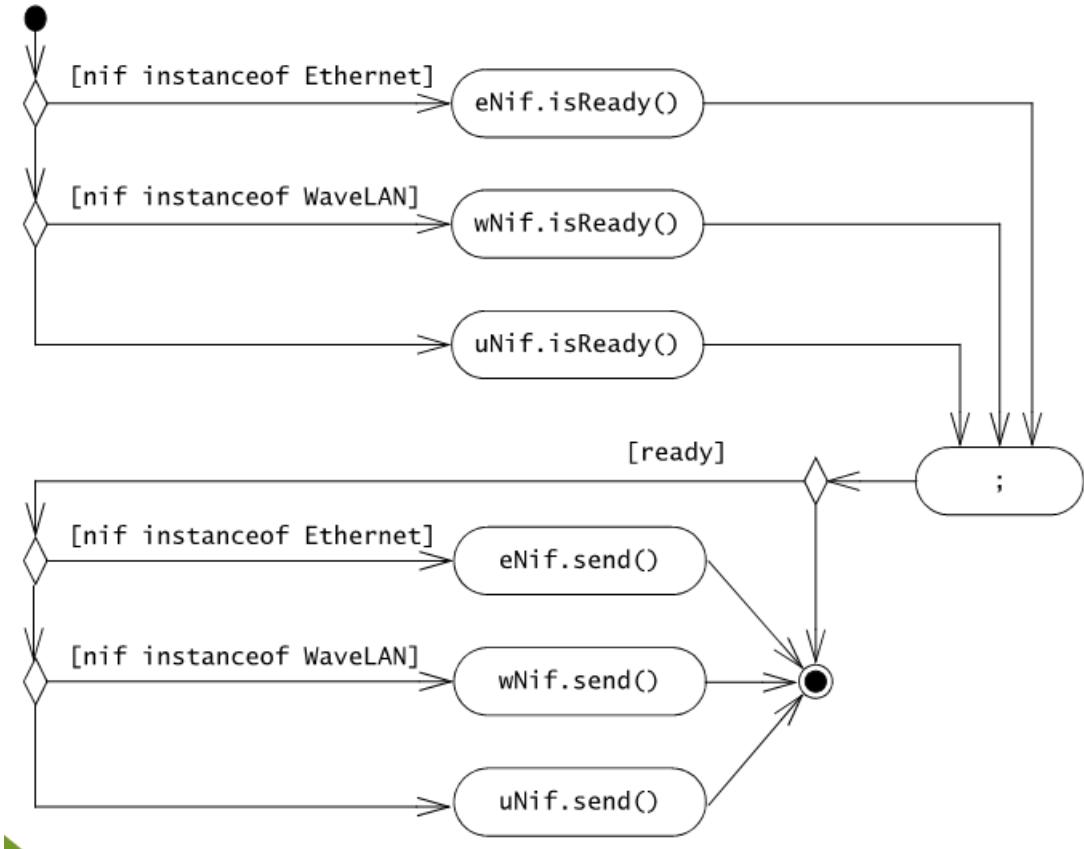
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    if (nif.isReady()) {
        nif.send(queue);
        queue.setLength(0);
    }
}
}

public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    boolean ready = false;
    if (nif instanceof Ethernet) {
        Ethernet eNif = (Ethernet)nif;
        ready = eNif.isReady();
    } else if (nif instanceof WaveLAN) {
        WaveLAN wNif = (WaveLAN)nif;
        ready = wNif.isReady();
    } else if (nif instanceof UMTS) {
        UMTS uNif = (UMTS)nif;
        ready = uNif.isReady();
    }
    if (ready) {
        if (nif instanceof Ethernet) {
            Ethernet eNif = (Ethernet)nif;
            eNif.send(queue);
        } else if (nif instanceof WaveLAN){
            WaveLAN wNif = (WaveLAN)nif;
            wNif.send(queue);
        } else if (nif instanceof UMTS){
            UMTS uNif = (UMTS)nif;
            uNif.send(queue);
        }
        queue.setLength(0);
    }
}
}

```




Infine il Flow Graph deve coprire (nel senso di trattare, considerare) il codice espanso:



ALTRÉ ATTIVITÀ DI TESTING

Integration Testing

Una volta che i Faults in ogni componente sono stati rimossi le componenti sono pronte ad essere integrate **ma potrebbero ancora contenere Faults**, infatti Stubs e Drivers sono solo approssimazioni delle componenti che simulano.

Nell'integration testing due o più componenti vengono integrate e testate e se non vengono rilevati nuovi Faults vengono aggiunti nuovi componenti.

L'Integration Testing può essere orizzontale o verticale.

Integration Testing Orizzontale

Nell'integration testing orizzontale le componenti sono integrate a strati.

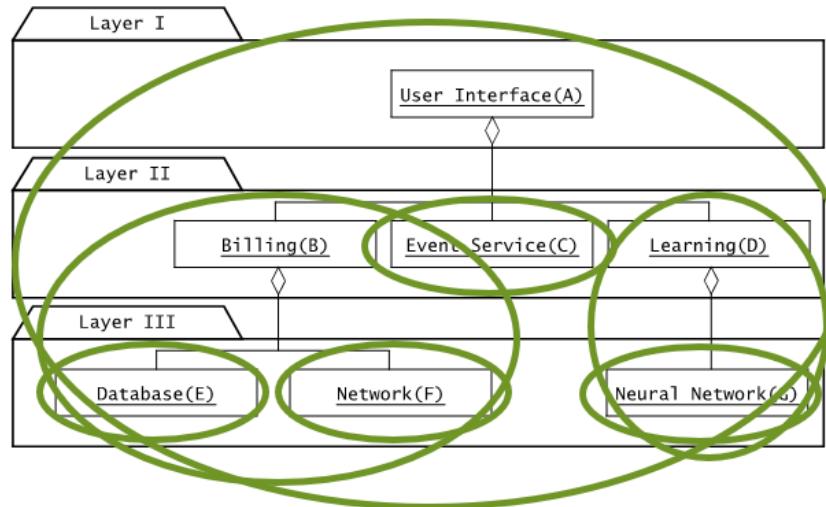
Le assunzioni sono che il sistema può essere **decomposto gerarchicamente**, che ogni componente appartiene ad uno strato gerarchico e che gli strati sono ordinati secondo l'associazione "chiamata".

Si possono seguire tre strategie principali:

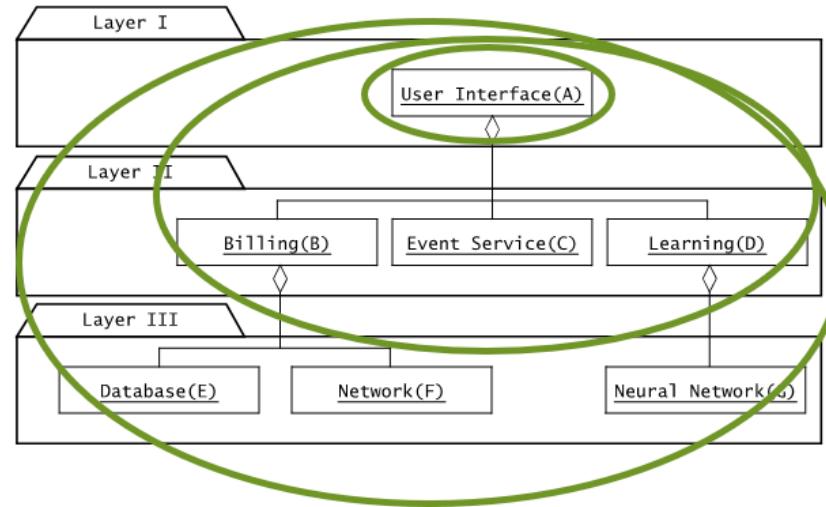
- Bottom-Up (può richiedere la creazione di driver);
- Top-Down (può richiedere la creazione di stub);

- **Sandwich** (da sopra e da sotto contemporaneamente e prima o poi si uniscono al centro).

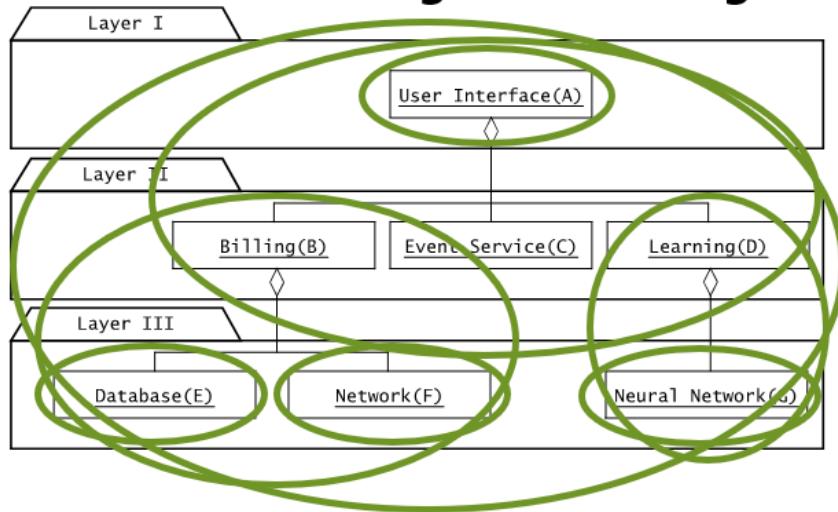
Bottom-Up Integration Testing



Top-Down Integration Testing



Sandwich Integration Testing



Integration Testing Verticale

Nell'Integration Testing Verticale le componenti sono integrate secondo funzioni.

Mentre con l'integration testing orizzontale un sistema operazionale può essere rilasciato solo molto tardi nello sviluppo l'integration testing verticale si concentra sull'integrazione "early" (fatta presto nello sviluppo).

Per una certa User Story si identificano, sviluppano e testano (con l'Integration Testing) le parti di ogni componente che sono necessitate.

Alla fine di ogni iterazione è prodotta e mostrata al cliente una nuova release.

Il Progetto del Sistema si evolve incrementalmente ma questo può portare alla riapertura di grandi decisioni di System Design (progettazione del sistema).

Regression Testing

Quando si modifica un componente gli sviluppatori progettano nuovi test che esercitano le feature introdotte, inoltre aggiornano e rieseguono gli Unit Test fatti in precedenza.

Queste attività possono portare alla comparsa di effetti collaterali in altre componenti e porta alla necessità di nuovi Integration Test che vengono chiamati Regression Tests.

La tecnica di Regression Testing più **robusta** è la riesecuzione di tutti gli Integration Test fatti in passato ma questo è time-consuming ed è conveniente solo se è già disponibile una infrastruttura per il testing automatico.

Possono essere applicate anche altre tecniche.

Retest delle componenti dipendenti

- Le componenti che dipendono da quella modificata sono quelle che più probabilmente avranno problemi
- Selezionare questi test massimizzerà la probabilità di individuare Faults

Retest degli Use Case "rischiosi"

- Assicurarsi che i Faults più catastrofici siano identificati è più critico che identificare il numero più alto di Faults possibile

Retest degli **Use Case frequenti**

- Gli utenti si aspettano che le features che hanno funzionato in passato continuino a funzionare
- Per massimizzare questa probabilità gli sviluppatori si concentrano sugli Use Cases che sono usati più spesso

Test di Usabilità

Viene testata la comprensione del sistema che hanno gli utenti

- non compara il sistema rispetto alle specifiche
- si concentra sulla ricerca delle differenze tra il sistema e le aspettative degli utenti

I tipici obiettivi di questo testing includono:

- la comparazione degli stili di interazione di diversi utenti
- l'identificazione di features utili
- quando è necessario aiuto
- che tipo di "informazioni di training" sono necessarie

Gli obiettivi di test sono valutati in esperimenti. I partecipanti sono allenati a svolgere dei tasks predefiniti.

Gli sviluppatori osservano i partecipanti e raccolgono dati per:

- identificare le preferenze degli utenti
- misurare le performance degli utenti (tempo per svolgere un task, rate di errore ecc)
- identificare specifici problemi con il sistema
- raccogliere dati per migliorare il software

L'obiettivo è ottenere informazioni qualitative su

- come risolvere problemi di usabilità
- come migliorare il sistema

Tipi di Usability Tests

Test di Scenario

Agli utenti finali viene presentato uno scenario realistico del sistema (ad esempio un set di mock-ups)

Gli sviluppatori identificano:

- quanto rapidamente gli utenti sono in grado di comprendere lo scenario
- quanto accuratamente lo scenario rappresenta il modello di lavoro degli utenti
- quanto positivamente gli utenti reagiscono al sistema

Il vantaggio del test di scenario è che è economico da realizzare e ripetere ma lo svantaggio è che manca di interazione con il sistema e ha i dati fissi (dati fissi nel senso che se ho fatto un mockup è tutto fisso).

Test di Prototipo

Agli utenti finali viene presentato un pezzo di software che implementa gli aspetti chiave del sistema.

Si fa differenza tra:

- **Prototipo verticale:** implementa completamente uno use case (usato per valutare i requirements)
- **Prototipo della UI:** presenta una interfaccia per la maggior parte degli Use Cases (senza fornire funzionalità)
- **Prototipo del Mago di Oz:** è un prototipo della UI nel quale un operatore umano dietro le quinte agisce per creare l'illusione di un sistema con delle funzionalità

Il vantaggio del test di prototipo è che fornisce una visione realistica del sistema, lo svantaggio è che richiede più sforzo per essere costruito rispetto ai Test di Scenari.

System Testing

Una volta che i componenti sono stati integrati il System Testing permette di verificare che l'intero sistema aderisca ai requisiti.

Il System Testing comprende:

- **Functional Testing:** testa i requisiti funzionali
- **Performance Testing:** testa i requisiti non funzionali
- **Pilot Testing:** è un test fatto con un gruppo selezionato di utenti finali
- **Acceptance Testing:** è un test fatto dal cliente nell'ambiente di sviluppo

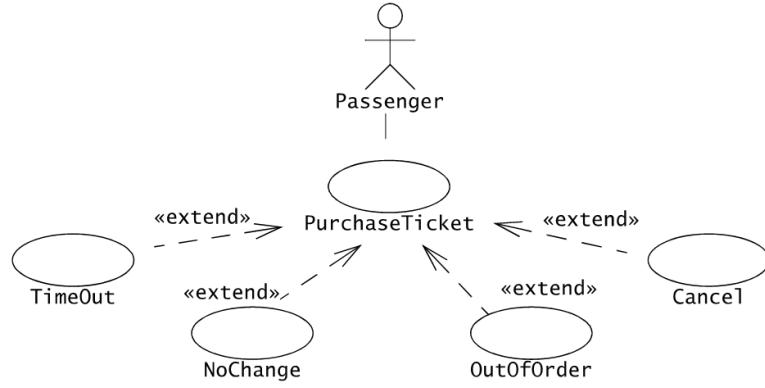
Functional Testing

Vengono testate le differenze tra i requisiti funzionali e il sistema, i test cases sono ricavati dagli use cases. Si tratta di una tecnica di testing **Black-Box**.

Come identificare i test funzionali:

- si ispezionano gli use case model
- si identificano istanze degli use case che più probabilmente possono andare in contro a fallimento
- si ricorda di esercitare sia gli use cases comuni che quelli eccezionali

Ex: distributore dei biglietti del treno



Lo use case PurchaseTicket descrive la normale interazione tra i passeggeri e il distributore

<i>Use case name</i>	PurchaseTicket
<i>Entry condition</i>	The Passenger is standing in front of ticket Distributor. The Passenger has sufficient money to purchase ticket.
<i>Flow of events</i>	1. The Passenger selects the number of zones to be traveled. If the Passenger presses multiple zone buttons, only the last button pressed is considered by the Distributor. 2. The Distributor displays the amount due. 3. The Passenger inserts money. 4. If the Passenger selects a new zone before inserting sufficient money, the Distributor returns all the coins and bills inserted by the Passenger. 5. If the Passenger inserted more money than the amount due, the Distributor returns excess change. 6. The Distributor issues ticket. 7. The Passenger picks up the change and the ticket.
<i>Exit condition</i>	The Passenger has the selected ticket.

Vengono rilevate le features che possono fallire con più probabilità:

- vengono premuti più tasti "zona" prima di inserire il denaro
- viene selezionato un altro tasto di "zona" dopo aver iniziato ad inserire il denaro
- viene inserito più denaro del necessario

A questo punto è possibile definire un test case che esercita le tre features individuate

<i>Test case name</i>	PurchaseTicket_CommonCase
<i>Entry condition</i>	The Passenger standing in front of ticket Distributor. The Passenger has two \$5 bills and three dimes.
<i>Flow of events</i>	<ol style="list-style-type: none"> 1. The Passenger presses in succession the zone buttons 2, 4, 1, and 2. 2. The Distributor should display in succession \$1.25, \$2.25, \$0.75, and \$1.25. 3. The Passenger inserts a \$5 bill. 4. The Distributor returns three \$1 bills and three quarters and issues a 2-zone ticket. 5. The Passenger repeats steps 1–4 using his second \$5 bill. 6. The Passenger repeats steps 1–3 using four quarters and three dimes. The Distributor issues a 2-zone ticket and returns a nickel. 7. The Passenger selects zone 1 and inserts a dollar bill. The Distributor issues a 1-zone ticket and returns a quarter. 8. The Passenger selects zone 4 and inserts two \$1 bills and a quarter. The Distributor issues a 4-zone ticket. 9. The Passenger selects zone 4. The Distributor displays \$2.25. The Passenger inserts a \$1 bill and a nickel, and selects zone 2. The Distributor returns the \$1 bill and the nickel and displays \$1.25.
<i>Exit condition</i>	The Passenger has three 2-zone tickets, one 1-zone ticket, and one 4-zone ticket.

Performance Testing

Consiste nel trovare le differenze tra il sistema e i requisiti non funzionali.

Esistono diverse tecniche di performance testing:

- **Stress testing:** controlla se il sistema può rispondere a molte richieste simultanee
- **Volume testing:** prova a trovare faults associati a grandi quantità di dati
- **Security testing:** prova a trovare Faults relativi alla sicurezza nel sistema, per questa tecnica sono coinvolti i Penetration Testers cioè individui che con la loro esperienza e conoscenza di tipici difetti di sicurezza provano a penetrare nel sistema
- **Timing testing:** prova a trovare comportamenti che violano i timing constraints

Pilot Testing

Il sistema viene installato e usato da un set selezionato di utenti che **esercitano il sistema senza delle linee guida esplicite**. Dopo il test gli utenti "pilota" danno feedback agli sviluppatori.

Possiamo differenziare il Pilot Testing in:

- **Alpha Testing:** gli utenti pilota esercitano il sistema nell'ambiente di sviluppo
- **Beta Testing:** gli utenti pilota esercitano il sistema nell'ambiente target (obiettivo)
 - il beta testing sta diventando sempre più comune e addirittura alcune aziende lo usano come metodo di testing principale offrendo una versione beta del loro software a chiunque sia interessato a testarla (o per buona volontà o per avere agevolazioni nell'ottenimento del software finito)

Acceptance Testing

Include diverse pratiche

- **Benchmark testing:** il cliente esercita il sistema, i test cases rappresentano tipiche condizioni sotto le quali il sistema dovrebbe operare
- **Competitor testing:** il sistema è testato rispetto ad un prodotto di un competitor
- **Shadow testing:** il sistema è testato rispetto al sistema esistente che dovrà sostituire eseguendo i due sistemi in parallelo e confrontando i loro output. Può essere considerato un caso speciale di Competitor Testing

Dopo il completamento dell'Acceptance Testing se il cliente è soddisfatto il sistema viene accettato, altrimenti il cliente riferisce al Project Manager quali requisiti esistenti non sono stati soddisfatti ed eventualmente quali requisiti vanno modificati o aggiunti.

Questo forma le basi per un'altra iterazione del ciclo di vita del software.

Test Planning

Al fine di ridurre i costi e i tempi del testing si decide di:

- **progettare i test cases presto**
- **parallelizzare i test**

I casi di test possono essere progettati non appena i modelli che validano diventano stabili.

Quando gli use cases sono completi bisogna sviluppare i test funzionali.

Quando le interfacce dei sottosistemi sono definite bisogna sviluppare Unit Test, Stubs e Drivers,

Seguendo queste linee guida l'esecuzione dei test inizia non appena i componenti sono disponibili.

Quanto costa il testing?

Il testing rappresenta una parte sostanziale delle risorse del progetto, una linea guida generale è che per i test bisogna allocare il 25% delle risorse (Unified Process) ma questo numero può salire sulla base dei requisiti riguardanti la sicurezza e l'affidabilità.

Documentazione del Testing

Le attività di testing vanno documentate in 4 diversi tipi di documenti:

- Il Test Plan
- Il Test Case Specifications
- Il Test Incident Reports
- Il Test Summary Report

Test Plan

Il Test Plan si concentra:

- sugli aspetti manageriali (scope, approccio, risorse e schedule delle attività di testing)
- sui Requisiti e sui Componenti da testare

Test Plan

1. Introduction
 2. Relationship to other documents
 3. System overview
 4. Features to be tested/not to be tested
 5. Pass/Fail criteria
 6. Approach
 7. Suspension and resumption
 8. Testing materials (hardware/software requirements)
 9. Test cases
 10. Testing schedule
-

Test Case Specifications

Questo documento documenta ogni singolo test case

Test Case Specification

1. Test case specification identifier
 2. Test items
 3. Input specifications
 4. Output specifications
 5. Environmental needs
 6. Special procedural requirements
 7. Intercase dependencies
-

Ex:

1. Test Case Specification Identifier

Si tratta del nome del test case, usato per distinguerlo dagli altri test cases. Le Convenzioni (come chiamare i test cases sulla base delle features/componenti che testano) permettono agli sviluppatori di riferirsi più facilmente ai test cases.

2. Test items

Lista dei componenti testati e delle features che vengono esercitate.

3. Input specifications

Lista degli input necessari per i test cases.

4. Output specifications

Lista degli output attesi. Questi output sono computati manualmente o con l'aiuto di un sistema concorrente (come ad esempio un vecchio sistema che sta per essere rimpiazzato)

5. Environmental needs

Lista delle piattaforme hardware e software necessarie per eseguire i test, inclusi test drivers o stubs.

6. Special procedural requirements

Una lista che contiene tutti i constraints necessari per eseguire il test come il timing, il carico o l'intervento di un operatore.

7. Inter-case dependencies

Lista delle dipendenze da altri test cases.

Test Incident Reports

Questi documenti documentano ogni esecuzione di ogni test registrando i risultati ottenuti e le differenze di questi dagli output attesi. Dovrebbe contenere abbastanza informazioni da poter (volendo) riprodurre i failures ottenuti.

Test Report Summary

Si tratta di una lista di tutti i Failures scoperti durante il testing.

Grazie a questo documento gli sviluppatori analizzano e prioritizzano ogni Failure e poi pianificano i cambiamenti nei sistemi e nei modelli. Questi cambiamenti portano poi a nuovi test cases.

Assegnare le Responsabilità

Il testing dovrebbe essere fatto da degli sviluppatori non coinvolti nello sviluppo del componente testato.

Di solito si crea un **Testing Team dedicato** al quale vengono dati i modelli, il codice e il sistema, che vengono usati per sviluppare ed eseguire i test cases.

Il Testing Team invia poi al Development Team (team di sviluppo) i Test Incident Reports e i Test Report Summaries.

Dopo una revisione il Team di Sviluppo invia il sistema revisionato al testing team che lo ritesta per controllare se i Failures originali sono stati risolti **e per assicurarsi che non siano stati inseriti nuovi**

Faults nel sistema.

DISCLAIMER

Questi appunti sono stati realizzati a scopo puramente educativo e di condivisione della conoscenza. Non hanno alcun fine commerciale e non intendono violare alcun diritto d'autore o di proprietà intellettuale.

I contenuti di questo documento sono una rielaborazione personale di lezioni universitarie, materiali di studio e concetti appresi, espressi in modo originale ove possibile. Tuttavia, potrebbero includere riferimenti a fonti esterne, concetti accademici o traduzioni di materiale didattico fornito dai docenti o presente in libri di testo.

Se ritieni che questo documento contenga materiale di tua proprietà intellettuale e desideri richiederne la modifica o la rimozione, ti invito a contattarmi. Sarò disponibile a risolvere la questione nel minor tempo possibile.

In quanto autore di questi appunti non posso garantire l'accuratezza, la completezza o l'aggiornamento dei contenuti e non mi assumo alcuna responsabilità per eventuali errori, omissioni o danni derivanti dall'uso di queste informazioni. L'uso di questo materiale è a totale discrezione e responsabilità dell'utente.