Metodi di Verifica e Testing
A.A.2012/13 – 9CFU

Prof.Enrico Vicario
Corso di Laurea Magistrale in Ingegneria Informatica
Facoltà di Ingegneria - Università di Firenze

1. Verifica e Testing

Nell'ambito del ciclo di sviluppo di un sistema con componenti SW, la **verifica** è l'attività mirata ad accertare la capacità di una implementazione di soddisfare i requisiti prescritti. L'attività di verifica è inquadrata diversamente a seconda della natura del prodotto (in relazione al diverso livello di criticità, alla rilevanza della integrazione tra HW e SW, …) e al modello di SW development lifecycle applicato (e.g. waterfall, Unified Process, eXtreme Programming, V-model).

TBD: aggiungere un capitolo che inquadra la verifica nei diversi SW lifecycle models.

In maniera generale, il **testing** viene definito come un metodo di verifica nel quale la correttezza di una Implementation Under Test (IUT) viene accertata esercitandola e comparandone il funzionamento rispetto alle attese.
In questa prospettiva si dice che il testing è una tecnica di **verifica dinamica**, basata sulla sperimentazione, in opposizione a una tecnica statica nella quale le proprietà dell'implementazione sono derivate attraverso un processo di analisi.

TBD: il caso di Tick@ come esempio di combinazione di analisi e testing.

Si capisce fino da questo livello di generalità che il testing serve in prospettiva formativa a identificare e rimuovere difetti e in prospettiva di valutazione ad acquisire una ragionevole confidenza circa l'assenza di difetti residui non riscontrati nel corso dello sviluppo, senza però mai poterne garantire l'assenza (concetto anche noto ocme "tesi di Dijkstra"). Per questo limite in ambiente accademico è a volte sminuito, ma in realtà esso costituisce un passo fondamentale e insostituibile nella pratica dello sviluppo del SW, del tutto prevalente nel supporto della verifica e specificamente prescritto e disciplinato nella costruzione di sistemi safety critical (e.g. dispositivi elettromedicali, segnalamento ferroviario, applicazioni avioniche e aerospaziali, …) dove costituisce una delle voci prevalenti del costo di sviluppo .

1.1 Ontologia di errore, difetto e malfunzionamento

Intuitivamente il concetto di correttezza ha a che fare con l'assenza di errori. Per trattarne occorre attribuire un significato preciso al concetto.

Nel contesto applicativo del testing (), si distinguono tre concetti fondamentali:

- **failure** è un **malfunzionamento** nel comportamento del sistema, ovvero una manifestazione del sistema nella quale non sono soddisfatti i requisiti prescritti;
- **defect (aka fault)** è un **difetto** nella implementazione del sistema che, anche in combinazione ad altri difetti, può condurre se esercitato ad una failure;
- **error** è un **errore** nel processo di sviluppo che ha condotto al difetto, sia esso un fatto accidentale (e.g. il programmatore si è distratto) o sistematico (e.g. un errore nelle assunzioni circa il comportamento di un componente).

Un difetto non conduce necessariamente ad un fallimento, e che comunque questo può avvenire solo laddove uno o più difetti sono esercitati nel corso dell'esecuzione. Ad esempio se il codice `if(t>10) x=y+z;` è erroneamente sostituito da `if(t>10)`

`x=y*z;` se l'istruzione è raggiunta sotto la condizione t<=10 il difetto non viene esercitato; se `t>10, x==y==2`, il difetto è esercitato ma non produce un malfunzionamento realizzando una condizione di correttezza accidentale (incidental correctness).

Vale infine la pena di osservare che nel SW l'origine del difetto ha una sua specificità diversa da quella di altri contesti produttivi: nel SW un difetto non insorge per invecchiamento e nemmeno nella riproduzione di un modello in più copie.

Figura 1: ontologia dell'errore nella teoria del testing.

Nel contesto applicativo della **fault-tolerance** viene usualmente usata una diversa ontologia che attribuisce significato diverso agli stessi tre termini: il fault è ancora un **difetto**, che può condurre il sistema in uno **stato di errore** a partire dal quale è possibile arrivare ad un **fallimento** o ritornare in uno stato di operazione corretta. La formulazione toglie enfasi sulla origine del difetto e dà invece spazio alla possibilità che ha il sistema di recuperare un comportamento corretto dopo avere raggiunto una condizione di errore, che è appunto il concetto centrale nella disciplina. La formulazione anche introduce uno schema di ragionamento strutturato per cui il fallimento di un sottosistema costituisce un fault che conduce in uno stato di errore il sistema che lo contiene. Che a sua volta potrebbe essere un sottosistema di un sistema di scala più ampia.

Nella trattazione di questo corso faremo riferimento alla prima classificazione, quella propria del contesto applicativo del testing. Vale comunque la pena conoscere entrambe le formulazioni e per chiarirne le differenze è utile soffermarsi su un caso specifico.
Il caso del fallimento del lancio di ariane è un buon esempio. Sotto è riportata una breve descrizione. Il report esteso è facilmente reperibile in rete ricercando "**ARIANE 5**: Flight 501 Failure - Report by the Inquiry Board" by The Chairman of the Board Prof. J. L. LIONS, Paris, 19 July 1996. (Incluso in Appendice)

1996: il vettore Ariane 5 esplode al decollo, danni per 1 miliardo di euro

Il 4 giugno 1996 viene lanciato per la prima volta il vettore Ariane 5, punta di diamante del programma spaziale europeo. Dopo 39 secondi di volo interviene il sistema di autodistruzione, trasformando l'Ariane 5 e il suo carico pagante (quattro satelliti scientifici non assicurati, per un valore di 500 milioni di dollari) in quello che è stato definito "il più costoso fuoco d'artificio della storia".
Non ci sono vittime, dato che il missile non ha equipaggio e i frammenti dell'esplosione cadono in una zona disabitata della Guiana francese, ma i danni economici sono ingentissimi (1 miliardo di euro). Il disastro avviene perché un programma del sistema di navigazione tenta di mettere un numero a 64 bit in uno spazio a 16 bit.
Il sistema, progettato per l'Ariane 4 e riciclato perché dimostratosi affidabile, tenta infatti di convertire la velocità laterale del missile dal formato a 64 bit al formato a 16 bit. Tuttavia l'Ariane 5 vola molto più velocemente dell'Ariane 4, per cui il valore della velocità laterale è più elevato di quanto possa essere gestito dalla routine di conversione, che (a differenza di molte altre routine di conversione a bordo) è priva dei normali controlli che evitano problemi di gestione.
Risultato: overflow, spegnimento del sistema di guida, e trasferimento del controllo al secondo sistema di guida, che però essendo progettato allo stesso modo è già andato in tilt nella medesima maniera pochi millisecondi prima. Privo di guida, il vettore si autodistrugge.

Ironia della sorte, la funzione di conversione difettosa che causa lo spegnimento del sistema di guida non ha alcuna utilità pratica una volta che l'Ariane è partito: serve soltanto per allineare il vettore con le coordinate celesti (l'Ariane ha un sistema di navigazione inerziale). Ma i progettisti avevano deciso di lasciarla attiva per i primi 40 secondi di volo in modo da facilitare il riavvio del sistema se si fosse verificata una breve interruzione nel conto alla rovescia.
http://attivissimo.blogspot.com/2007/11/disastri-informatici-la-compilation.html

In entrambe le prospettive del testing e della fault tolerance il difetto del caso consiste nella incapacità della variabile che riceve la misura di accelerazione angolare di contenere appropriatamente il valore misurato.

Nella prospettiva del testing, il difetto deriva da vari errori: la non compresa necessità di ridisegnare la componente di misura della accelerazione angolare a seguito della reingegnerizzazione della componente energetica del razzo, la decisione di trascurare la necessità di proteggere da overflow la misura di accelerazione angolare; la decisione di spengere il processore in presenza di un riscontrato malfunzionamento nella misura della accelerazione angolare.
Il malfunzionamento si manifesta su diversi livelli di del sistema con diversa osservabilità rispetto ai processi di sviluppo: il componente SW che gestisce l'assetto manifesta un malfunzionamento quando non è capace di fornire una corretta valutazione dell'assetto; il componente SW di guida del lanciatore manifesta un malfunzionamento nel momento in cui entrambi i processori sono in stato di riavvio; il sistema complessivo manifesta un malfunzionamento nel momento in cui il razzo non riesce più a volare e si auto-distrugge.

Nella prospettiva di fault tolerance, il difetto conduce il sistema in uno stato di errore nel quale la misura di accelerazione angolare è inadeguata.
Le decisioni successive nella gestione dell'errore conducono il sistema al fallimento, che avrebbe potuto essere evitato con una diversa politica di fault-tolerance, banalmente con quella che ignorasse la misura.

1.2 Metodologia del processo di testing

Il processo di testing sottende un numero di attività collegate.

1.2.1 Test case selection

è l'attività di selezione dei casi di test che si ritengono adeguati a causare fallimenti nel caso in cui la IUT includa difetti o che comunque in assenza di fallimenti riscontrati si ritengono adeguati a garantire con accettabile confidenza l'assenza di difetti residui.
L'insieme dei casi di test costituisce una **test-suite**, che è tipicamente costruita attraverso un criterio di selezione.
Una test-suite, o anche il criterio che essa realizza, può essere caratterizzata sotto due profili opposti: la **fault-detection-capability** definisce quali sono i faults che generano una failure osservabile nell'esercizio della suite; la **complessità** definisce quanti casi di test compongono la suite ovvero quanti casi sono necessari per costruire un a test suite che realizza un criterio.
E' chiaro che salvo casi limite o ipotesi molto restrittive, la fault-detection capability di una test-suite non è mai completa, il che in altri termini ancora ripropone la iniziale osservazione per cui il testing è adeguato a trovare difetti ma non a garantirne l'assenza.

1.2.2 Input selection (aka sensitization)

consiste nell'identificare gli input che fanno sì che il sistema esegua il test case che abbiamo selezionato; in un sistema in tempo reale consiste anche nel decidere a che istante far arrivare l'input.
La sensitization è in generale un problema non decidibile essendo evidentemente riducibile al problema della terminazione: non è computabile la funzione che riceve in ingresso un qualsiasi codice e determina gli inputs che permettono al codice di raggiungere una specifica linea e/o un determinato valore di una qualche variabile.

Esiste un accoppiamento tra le attività di test-case selection e sensitization: la test case selection è in generale organizzata come la selezione di una test-suite che realizza un qualche criterio di copertura; lo stesso criterio può essere soddisfatto con test-suites diverse; e può avvenire che i casi di una test-suite siano più difficili da sensitizzare rispetto a quelli di un'altra; per questo è utile che la sensitization possa produrre un feedback sulla test-case selection.

Un caso limite nella soluzione dei problemi di test-case selection e sensitization è l'approccio di **random testing** nel quale gli inputs sono generati in maniera casuale, delegando alla fase di coverage analysis una valutazione a posteriori dell'efficacia dei test svolti.

1.2.3 Esecuzione
I problemi sono quelli legati alla creazione di un driver (anche detto scaffold o stub*)* che guida l'esecuzione del test e alla generazione di logs dell'attività.

L'attività del **driver** è spesso ostacolata dalla presenza di componenti non-deterministiche che sono solo parzialmente controllabili da parte del driver. La questione, che ha affinità con ha affinità con la teoria dei giochi, assume particolare rilievo nel contesto del real-time testing.

Il **log** non deve perturbare il comportamento del sistema, che può diventare un problema per esempio nel caso di applicazioni real time.

1.2.4 Oracle verdict
La funzione dell'oracolo è quella di mettere un verdetto circa la conformità tra il comportamento osservato nell'esecuzione di un test e quello atteso.
Il verdetto può essere **pass/fail/inconclusive**: pass e fail indicano che l'interpretazione del log delle attività evidenzia un comportamento corretto o un malfunzionamento rispetto ai requisiti attesi; un verdetto in conclusivo è espresso laddove l'analisi evidenzia che il sistema ha eseguito lungo una traiettoria diversa da quella pianificata nella test-case selection.

L'Oracolo è in generale esposto ad questioni di osservabilità dei malfunzionamenti: alcuni malfunzionamenti rimangono confinati nella IUT e non sono osservabili nel corso del test. Da un punto di vista concettuale è questionabile che questi siano riguardati come malfunzionamenti: se non sono osservabili allora non degradano la prestazione del sistema nel soddisfacimento dei suoi requisiti.

1.2.5 Debugging
E' il problema di tracciare ciascuna failure riscontrata dall'oracolo su uno o più fault.

1.2.6 Coverage analysis

La coverage analysis mira a fornire una misura della copertura realizzata dai test case effettuati rispetto all'insieme dei comportamenti possibili.
Essa fornisce una stima della confidenza nella assenza di difetti residui non riscontrati grado di copertura raggiunto rispetto alla complessità reale (es. rispetto al totale delle righe di codice, ai requisiti funzionali, etc.) del sistema.

1.3 Prospettive di astrazione

Per vari aspetti test case selection e coverage analysis si riducono ad uno stesso problema, differenziati per il fatto che uno opera a-priori e l'altro a-posteriori.
Entrambi fanno riferimento ad una **astrazione della IUT**. In un caso l'astrazione viene usata per selezionare i casi di test, nell'altro per valutare quanta parte della complessità del sistema è stata esercitata nei test effettuati.

L'astrazione può essere realizzata in diverse prospettive.

Nella prospettiva **funzionale** (**black box**), l'astrazione fa riferimento alla specifica dei requisiti, senza fare riferimento al modo con cui il sistema è realizzato. Ad esempio questo può essere il caso del diagramma degli use-cases o i requisiti che compaiono in una **SRS** (Sw Requirements Specification). Per quanto l'approccio è detto funzionale, i requisiti possono anche avere natura non funzionale ed essere riferiti ad attributi di qualità quali affidabilità, disponibilità, tempi di ripristino, accessibilità, performance, …
Per inciso: lo standard IEEE 830 Sw Requirements Specification (SRS) include una buona indicizzazione dei requisiti nella forma di requisiti funzionali, architetturali e di qualità. A loro volta i requisiti di qualità di un sistema SW sono formalizzati nello standard ISO9126.

Nella prospettiva **strutturale** (**white box**), l'astrazione fa riferimento a come in effetti il sistema è realizzato. Astrazioni in questa prospettiva possono essere il codice sorgente, il codice compilato, il diagramma UML delle classi dell'implementazione, …

Nella prospettiva **architetturale** (grey box), l'astrazione fa riferimento a come un insieme di componenti sono combinati tra loro, senza intervenire del merito di come i componenti sono internamente realizzati.

TBD: esempio di un real time task set o di una architettura publish & subscribe

In generale non è detto che test case selection e coverage analysis facciano riferimento alla medesima prospettiva. Anzi, l'approccio che tipicamente è ritenuto più virtuoso è quello nel quale si combinano una **test case selection in prospettiva funzionale** e una **coverage analysis in prospettiva strutturale**.

Si dice a volte che la selezione di casi di test in prospettiva strutturale incontra una qualche forma di **tautologia**. Il concetto può essere meglio chiarito nel seguito (e.g. a valle del data flow testing), ma cerchiamo comunque di esemplificarlo: consideriamo un frammento di codice che dovrebbe correttamente essere scritto come:

```
x=5;
…
if(x>3)
…
```

Un ragionevole criterio di copertura strutturale stabilisce che occorre testare il sistema nel percorso dalla definizione di x al punto in cui x viene usato in una guardia.
Se il codice ha un difetto ed è scritto nella forma

```
x=1;
…
if(x>3)
…
```

il caso di test implicato dal criterio probabilmente produrrà una qualche failure osservabile.
Se però il difetto fosse nella forma

```
y=5;
…
if(x>3)
…
```

allora il criterio non è più capace di osservare il difetto perché è il difetto stesso che elimina dalla test suite il caso che lo avrebbe potuto osservare.

1.4 Livelli del testing

TBD: Test di unità, di integrazione, in laboratorio e in ambiente di produzione, di accettazione, di collaudo, test interno e test contrattuale, …

2. Control flow testing

Il control flow testing mira a esercitare la IUT in modo da coprire i diversi percorsi del suo flusso di controllo. Per questo la IUT è astratta in un **control flow graph** (CFG) che rappresenta un insieme di locazioni logiche della IUT e la relazione di transizione tra le locazioni.
L'approccio si presta in modo naturale ad una interpretazione in chiave **strutturale**, identificando le locazioni logiche con posizioni del controllo nel codice. Per questo descriviamo l'approccio in chiave strutturale inquadrandolo come metodo di coverage analysis.
Nella parte finale illustreremo come il criterio può essere convenientemente applicato anche in prospettiva **funzionale** e nella selezione dei casi di test.

2.1 Control flow graph

I **vertici** rappresentano locazioni del controllo e possono essere identificati con diversi livelli di granularità.
Nel caso più fine un vertice è una singola **istruzione** in un qualche livello di compilazione (codice sorgente, assembler simbolico, linguaggio macchina).
Nella pratica più diffusa viene usato un livello più coarso ma ugualmente rappresentativo identificando i vertici con basic-blocks: un **basic block** è una sequenza di istruzioni che sono necessariamente susseguenti, nel senso che l'esecuzione di una di esse implica necessariamente l'esecuzioen di tutte le altre. Nella pratica, questo identifica il basic block come sequenza di istruzioni che non contengono alcun salto condizionale (conditional branch) né alcuna etichetta che possa essere destinazione di più di una istruzione di salto (si osservi che questo non esclude l'inclusione di salti incondizionati né di etichette che siano destinazione di un unico salto).
In una astrazione ancora più coarsa, mirata a ridurre la complessità, i vertici possono essere identificati con punti di chiamata delle **funzioni**.

Gli **archi** (edges) rappresentano salti condizionati o accessi a locazioni associate ad una label che possa essere destinazione di più salti. Nel primo caso, l'arco è etichettato con un predicato che esprime la condizione sotto la quale viene preso il salto.

2.2 Criteri di copertura dell'astrazione

Il CFG può essere coperto in relazione a diversi criteri. Ciascuno realizza diversi gradi di fault detection capability e comporta una diversa complessità, misurata come numero di casi di test che può essere necessario eseguire per realizzare la copertura.

2.2.1 All Nodes

Il criterio è soddisfatto quando sono stati coperti tutti i nodi. Il criterio è anche detto all-statements, facendo riferimento al codice anziché alla sua astrazione nel CFG. Questo risponde al razionale per cui non è possibile avere confidenza di un codice nel quale sono annidate istruzioni che non sono mai state eseguite.

Il criterio ha **complessità O(N)** rispetto al numero N dei vertici del grafo. E' evidente che con N casi di test è possibile raggiungere qualsiasi vertice. Che possa diventare impossibile farlo con meno di O(N) casi è dimostrato dalla famiglia di casi illustrati in Figura, dove il numero dei casi scala linearmente rispetto al numero dei vertici del grafo.

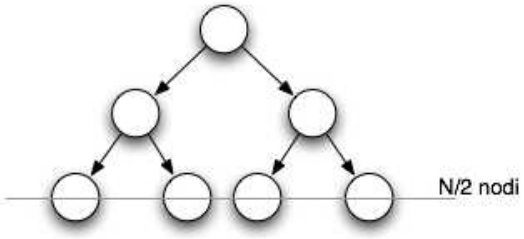


Figura 2: Per ogni foglia è necessario un diverso caso test. Su un grafo di N-1 vertici sono quindi necessari N/2 casi di test, ovvero il numero di casi di test è proporzionale alla dimensione del grafo.

La fault detection capability non è facilmente caratterizzabile. Per certo si può dire che il criterio garantisce che qualsiasi istruzione è stata eseguita almeno una volta, ma non esiste una diretta conseguenza tra questo e la natura dei difetti riscontrabili.
In presenza di confluenze nel CFG, il criterio non garantisce che la copertura di tutti gli archi, che nella pratica equivale a dire che non viene garantito che ciascun predicato in istruzioni condizionali (`if, while, do, for`) sia esercitato nelle diverse condizioni.
Ad esempio si consideri il frammento di codice che segue

```
float * V;
int N;
boolean alreadyAllocated;

…
if(alreadyAllocated==FALSE)
     V=(float *)malloc(sizeof(float)*N);
…
```

se la sezione di codice è raggiunta con la condizione `alreadyAllocated==TRUE`, tutti i nodi del CFG sono coperti, ma non è stato sperimentato cosa avviene nel caso in cui la `malloc()` non è eseguita. Questo costituisce una forte vulnerabilità perché potrebbe generare un mafunzionamento grave nel caso in cui la variabile `alreadyAllocated` non è trattata secondo l'intento che il suo nome sottende.

2.2.2 All Edges

Oltre a richiedere la visita di tutti i nodi, richiede anche l'attraversamento di tutti gli archi. Facendo riferimento al codice da cui deriva il CFG, è anche detto **all-decisions**, dove il termine decision denota il valore restituito da una guardia che controlla un branch di qualche tipo.

Una test suite che soddisfa all-edges soddisfa anche all-nodes, per il che si dice che all-edges **include** (o sussume come traduzione di subsumes) all-nodes.

Viceversa, è facile osservare che all-nodes include all-edegs se e solo se ogni vertice del CFG ha un unico predecessore (i.e. il CFG non include confluenze).

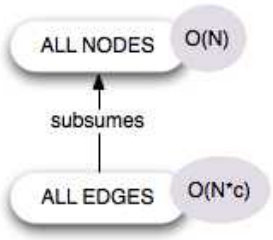


Figura 3: All-edges include (sussume) all-nodes.

Il criterio ha **complessità O(Nc)**, dove c è il massimo grado di uscita di ciascun vertice e vale quindi 2. Si osservi che questo è vero in modo evidente in cui i vertici del grafo sono singole istruzioni, e vale comunque anche nel caso in cui i vertici sono basic blocks.

2.2.3 All Conditions

Una **condizione** è una espressione che non contiene operatori booleani.
Il criterio all-conditions richiede che tutte le condizioni di tutti i predicati siano state testate con esito TRUE e FALSE. Le condizioni sono però **testate indipendentemente**, per cui non è garantito che alla fine il predicato sia testato con le condizioni TRUE e FALSE.

Illustriamo il concetto con un esempio:

```
if(x>10 && y<3)
```

Il predicato di guardia è composto da due condizioni. Se il frammento è testato con ingressi <x==20,y==4> e <x==5,y==1>, entrambe le condizioni sono state testate con esito TRUE e FALSE, ma la guardia non ha mai restituito una decisione TRUE.

Il passaggio offre lo spunto per una osservazione di merito che richiama la relazione tra test-case selection e sensitization.
Ai fini della test-case selection, la test suite dovrebbe essere espressa nella forma

Test case	x>10	y<3
1	T	F
2	F	T

In fase di sensitizzazione ciascun test case è associato ai valori di input che lo sensitizzano:

Test case	x>10	y<3	Input
1	T	F	<x==20,y==4>
2	F	T	<x==5,y==1>

La separazione dei due passi favorisce l'astrazione.

L'esempio chiarisce che all-conditions non include all-edges e neppure all-nodes. E' di per sé un criterio di poca utilità, introdotto in maniera strumentale ad altri che seguono.

2.2.4 Condition Decisions

E' definito come l'**unione** di all conditions e all decisions, ovvero è ottenuto come unione di due test suites che soddisfano separatamente i due criteri.
Nel caso dell'esempio già citato:

```
if(x>10 && y<3)
```

la copertura condition decision può essere ottenuta aggiungendo un ulteriore caso

Test case	x>10	y<3	x>10&&y<3	Input
1	T	F	F	<x==20,y==4>
2	F	T	F	<x==5,y==1>
3	T	T	T	<x==20,y==1>

Si osservi che la copertura così ottenuta non è ottima, nel senso che ne esiste una meno numerosa:

Test case	x>10	y<3	x>10&&y<3	Input
3	T	T	T	<x==20,y==1>
4	F	F	F	<x==5,y==4>

Per costruzione condition-decision **include all-decisions**.
Ha complessità **O (cNk)**, dove k è il massimo numero di condizioni in una guardia.

2.2.5 Multiple Conditions

Considera **tutte le combinazioni** di tutte le condizioni. Nel caso dell'esempio
Nel caso dell'esempio già citato:

```
if(x>10 && y<3)
```

la copertura multiple conditions si realizza con 4 casi:

Test case	x>10	y<3	x>10&&y<3	Input
1	T	F	F	<x==20,y==4>
2	F	T	F	<x==5,y==1>
3	T	T	T	<x==20,y==1>
4	F	F	F	<x==5,y==4>

Multiple conditions implica tutti gli altri ma comprende un numero di casi che è esponenziale rispetto al numero di condizioni: **$O(cN \cdot 2^k)$**

Alcuni linguaggi, tra cui notabilmente c, c++ e Java, compilano le espressioni Booleane con il meccanismo del corto-circuito (**short-circuit**). Nel riferimento dell'esempio precedente, se `x>10` restituisce valore falso, l'espressione `y<3` non viene calcolata. Per questo i casi 2 e 4 risultano nell'esecuzione dello stesso codice assembler per cui uno dei due può essere omesso senza modificare la copertura raggiunta.

Illustriamo il concetto in un caso più esteso:

```
if ((A && B) || (C && D))
```

La decisione è determinata da 4 condizioni, che in principio richiedono 2^4=16 casi di test, tuttavia molti di questi sono equivalenti per effetto dei corto circuiti. La tabella che segue illustra le condizioni di don't care sulle condizioni corto-circuitate, realizzando la copertura con 7 casi:

Test case	A	B	C	D	((A&&B)||(C&&D))
1	F	-	F	-	F
2	F	-	T	F	F
3	F	-	T	T	F
4	T	T	-		T
5	T	F	F	-	T
6	T	F	T	F	T
7	T	F	T	T	T

2.2.6 Modified Condition Decision Coverage (MCDC)

Modified Condition/Decision Coverage (MC/DC) è raccomandato nello standard RTCA DO178B applicato nello sviluppo di airborne-SW (SW avionico) . In particolare MCDC è prescritto per il test di unità di componenti Level A (Catastrophic).

Il criterio stabilisce che:
i) ciascuna decisione deve avere avuto i possibili esiti True e False;
ii) ciascuna condizione deve avere avuto i possibili esiti True e False;
iii) ciascun entry e exit point è stato invocato (il che esula dagli argomenti ora trattati);
iv) ciascuna condizione ha determinato in maniera indipendente l'esito della decisione.

La condizione iv) in sostanza prescrive che ciascuna condizione è stata testata con esito True e False in condizioni che producono diversi esiti della decisione complessiva. Questo ovviamente implica le condizioni i) e ii).

Il criterio MC/DC ha il pregio di essere applicabile a prescindere dal fatto che il linguaggio implementi un meccanismo di coro-circuito. Questo realizza una condizione più neutrale nel trattamento dei linguaggi che non hanno corto-circuiti tra cui è notevole il caso di Ada.

MC/DC Ha complessità **O(cN·2K)**, seppure il suo calcolo ha complessità esponenziale rispetto a k.

2.2.7 All Paths

Richiede la copertura di tutti i possibili cammini sul grafo.
La complessità è **O(2^N)** in assenza di cilci, e degenera in **O(∞)** se ci sono cicli.

2.2.8 Boundary Interior

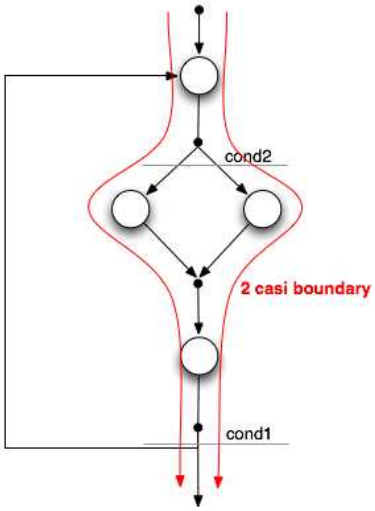
Siccome i cicli introducono un numero di path non limitato, esistono delle varianti all criterio all paths in cui il numero di casi di test è limitato raggruppando i path che differiscono solo per il numero di volte che interano sui loop e testando poi alcuni path rappresentativi per ogni gruppo. Il boundary-interior testing considera due classi di path per ogni gruppo di path simili rispetto ad ogni ciclo. I path della prima classe entrano nel

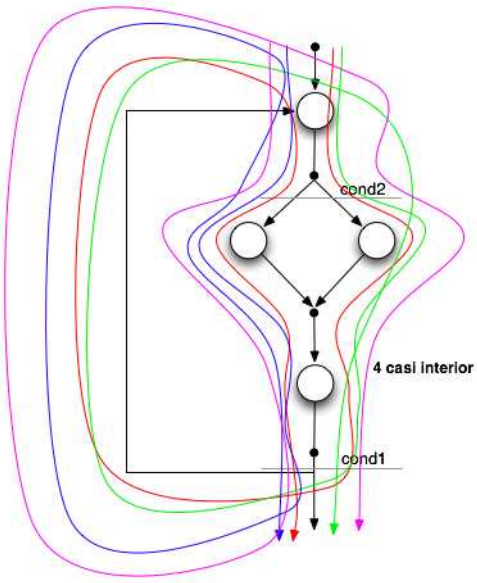
loop ma non iterano su di esso (*boundary tests*) mentre i path nella seconda classe iterano sul ciclo almeno una volta (*interior tests*). Se nel loop c'è un branch i casi boundary e i casi interior devono essere esercitati per distinguere il branch.

Per esempio se consideriamo un codice del tipo:

```
do{
…
if (condition2)
     {
     statement1
}
          else
          {
statement2
          }
…
} while (condition1)
```

Si possono individuare due casi boundary per questo ciclo (uno per ogni caso dell'if) e entrambi usciranno dal loop immediatamente. Il numero di casi interior invece è quattro; e tutti eseguiranno il corpo del loop una seconda volta. I quattro casi interior corrispondono alle quattro permutazioni dei rami nelle prime due esecuzioni dell'if-then-else (T-T, T-F, F-T, F-F) Dopo la seconda esecuzione del corpo del loop, ogni caso interior può uscire il loop o interarlo altre volte.





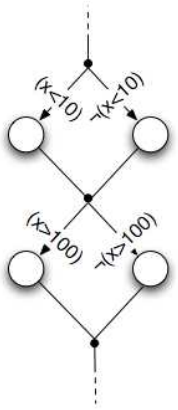
2.2.9 Osservazioni sui criteri di copertura

Dobbiamo notare però che non è detto che all paths garantisca la copertura di tutti gli errori: può sempre darsi che abbia "correttezza incidentale". Inoltre può accadere anche

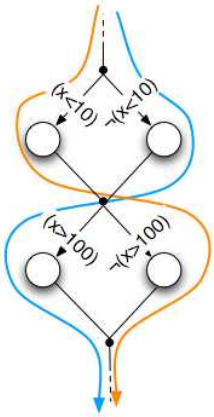
che all paths sia meno efficace di *all edges*. Pensiamo ad esempio che io abbia risorse sufficienti a fare M test, con i quali ottengo una copertura dell'80% di *all edges* e del 5 % di all paths; in questo caso avrò migliori risultati scegliendo di usare *all edges*.

Dobbiamo inoltre notare che non sempre è possibile effettuare un criterio di copertura al 100 %. Talvolta si può pensare di effettuare anche *all nodes* ad una percentuale minore. Ci spossono essere infatti dei problemi nel realizzare interamente questi criteri. Per esempio non è detto che un cammino che si incontra sull'astrazione sia in effetti eseguibile sul codice.
Es.

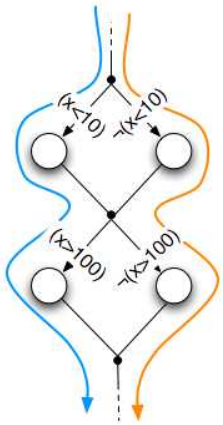
```
if (x<10)
…
else
…
if (x>100)
…
else
…
```



Se scelgo i seguenti due cammini, sono in grado di eseguirli:

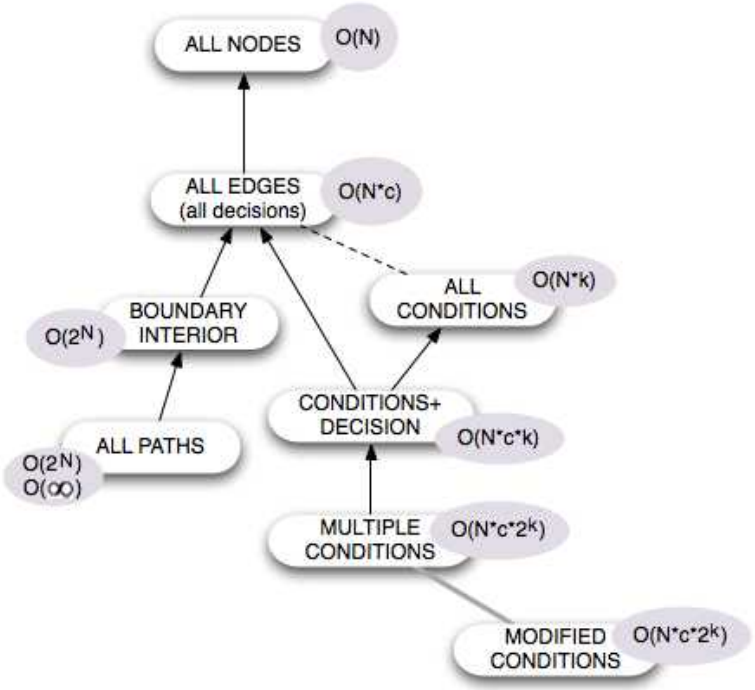


Se voglio però fare *all edges* e coprire anche questi due cammini:



Non troverò gli input per coprirli (problema dei non feasible paths).
Questo ci fa capire come sia importante la correlazione tra le diverse attività del testing.

Altro inconveniente alla copertura dei criteri al 100% è il fatto che talvolta l'esecuzione dipende anche da fattori esterni al software che stiamo testando, per esempio allo stato del sistema in cui il nostro componente viene eseguito (per esempio il caso in cui finisce la memoria dipende dalla memoria del sistema e dai criteri secondo cui viene allocata da esso).



3. Data flow testing

Il *data flow testing* è un'estensione del control flow testing che cerca di inserire qualcosa tra *all edges* e i criteri a complessità $O(2^N)$.
Il concetto fondamentale del data flow testing è quello di catturare le dipendenze tra le variabili.

Es:

Codice corretto:

```
x=10
```

Codice con errore:

```
x=100
```

Finché x non viene usata, non mi posso accorgere dell'errore. Se da qualche parte prendo una decisione che dipende da x, allora mi posso accorgere della differenza. Es: if (x<12) …

Notiamo che se ho un errore del genere:

Codice corretto:

```
1: x=10
2: y=x
3: if (y<12)…
```

Codice con errore:

```
1: x=100
2: y=x
3: if (y<12)…
```

La riga `y=x` mi trasferisce l'errore dallo stato della x a quello della y; anche se x non viene usata direttamente nell'if, x nella riga 2 viene usata per una computazione.

N.B. Se il codice fosse stato `if (x<150)` non mi sarei in ogni caso accorto dell'errore!

Per costruire il grafo su cui si basa il data flow testing (*Data Flow Graph*) si parte dal CFG, aggiungendo delle annotazioni.

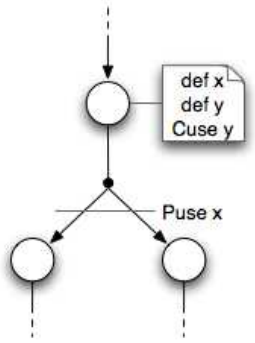
Definizione di una variabile: se in un nodo viene definita una variabile x, si aggiunge l'annotazione def x accanto a quel nodo. Attenzione: si ha definizione di una variabile non solo quando le viene assegnato un valore, ma ogni volta che tale variabile subisce un side effect.

***Cuse* di una variabile**: si aggiunge l'annotazione Cuse x in ogni nodo in cui la variabile x è referenziata in una <expr>.

***Puse* di una variabile**: si aggiunge l'annotazione Puse x ogni volta che la variabile x è referenziata in una <expr> e tale <expr> è contenuta in una delle <condition> di un costrutto condizionale (cioè la variabile viene usata per prendere una decisione). A differenza delle annotazioni def e Cuse, le annotazioni Puse vengono poste sugli archi del grafo.

Es:

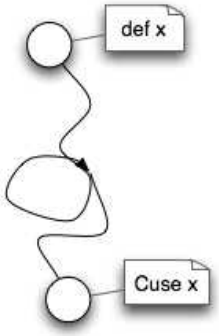
```
…
x=y++;
if (x>3)…
…
```

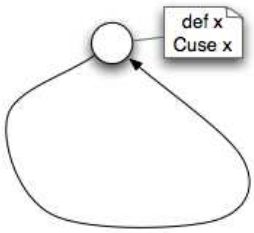


Definizioni:

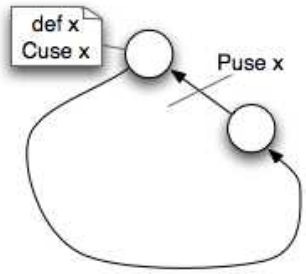
- *def clear path with respect to x*: è un cammino nel Data Flow Graph nel quale x non subisce def (side effects)
- *def use path with respect to x*: è un cammino nel grafo che inizia in un nodo dove ho un def x e:
  - termina su un Cuse x o un Puse x
  - non contiene def x intermedi (è def clear rispetto alla x)
  - "non contiene cicli" (non esegue più di una volta lungo il cammino lo stesso basic block)

non sono ammessi:





è ammesso:

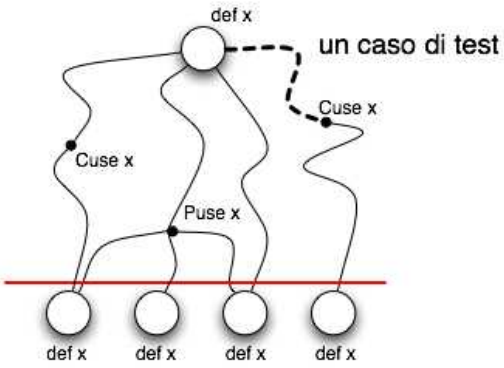


Il Puse si trova sull'arco, non sul nodo.

Criteri di copertura

- ALL DEF

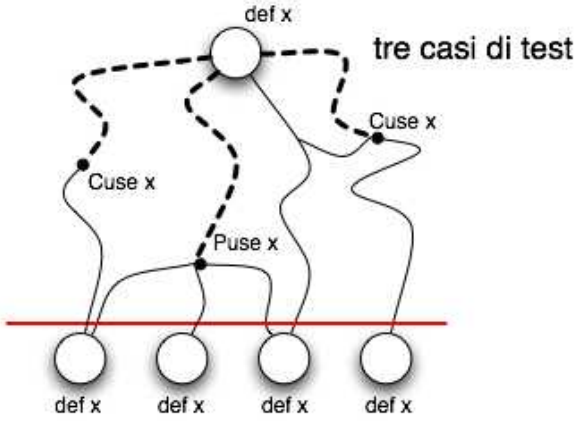
Questo criterio, per ogni def x, costruisce tutti i cammini (che partono da quel nodo) che sono def clear rispetto ad x (ovvero terminano con un altro def x) ed esercita per quel def almeno un use. Definisce in pratica una specie di "frontiera", oltre la quale il def x considerato non ha più importanza.



Potrebbero esistere più cammini che arrivano allo stesso use: basta che ne eserciti uno tra questi.

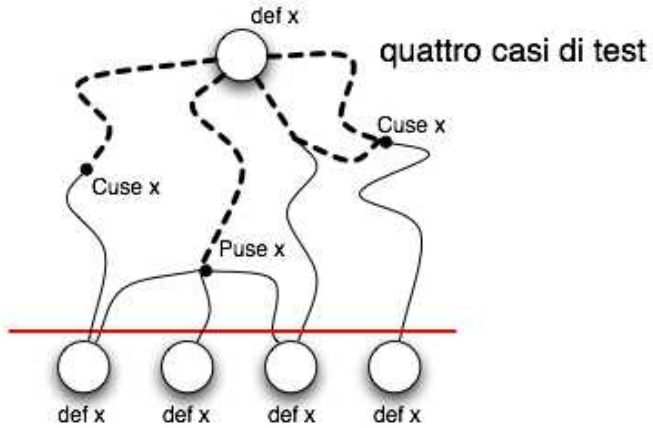
- *ALL USES*

Questo criterio esercita almeno un def use path (du path) per ogni uso.



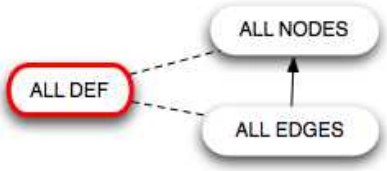
- ALL DU PATHS

Per ogni def esercito tutti i du paths.



N.B. Se ho un ciclo non lo esercito (i du paths non sono ciclici).

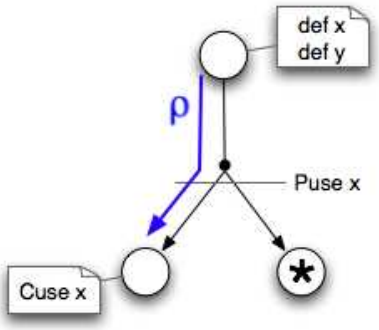
Confrontiamo questi criteri con quelli del data flow testing.



All def e *all nodes*/*all edges* non sono comparabili.
Dimostriamo che *all def* non include *all nodes*.

Consideriamo il seguente esempio:

```
x=10;
y=20;
if (y>100)
  z=x;
else
z=y;
```

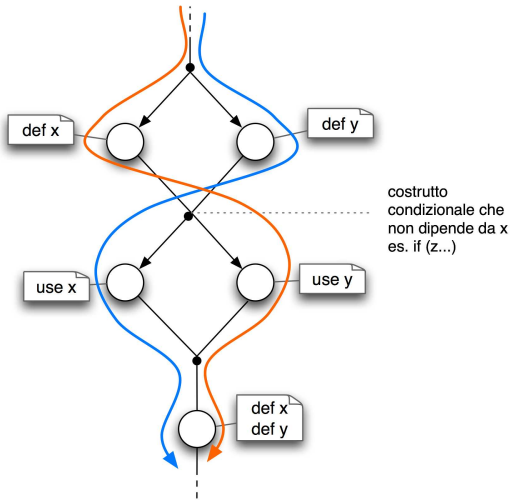


Se consideriamo il cammino ρ, esso realizza la copertura *all def*, ma non copre il nodo (*).

Visto che *all def* non include *all nodes*, esso non include nemmeno *all edges*.

Vediamo ora che *all edges* non include *all def*.
Es:

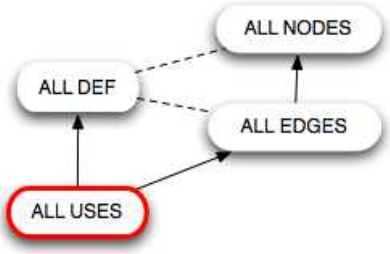
I due casi realizzano *all edges* ma non *all def*.



PROGRAM SLICING: consiste nell'identificare alcune variabili critiche e "tagliare" il codice andando ad osservare solo l'interazione tra quelle variabili (nell'esempio visto qui sopra non considero la variabile z nel costruire il grafo).

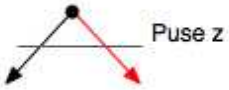
SLICING FUNZIONALE: si parte identificando le funzioni del sistema e si fa lo slicing del codice in relazione ai casi d'uso. Si usa per esempio quando si deve reingegnerizzare un sistema (andando a lavorare su codice scritto da altri).

Confrontiamo adesso *all uses* con *all edges* e *all nodes*.

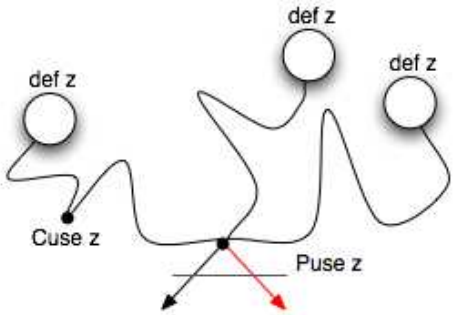


All uses include *all edges* (e quindi anche *all nodes*). Esso inoltre include ovviamente *all def*.

Consideriamo un edge, facendo l'ipotesi che nella condizione ci sia un'espressione che fa riferimento a almeno una variabile: if (…z…)

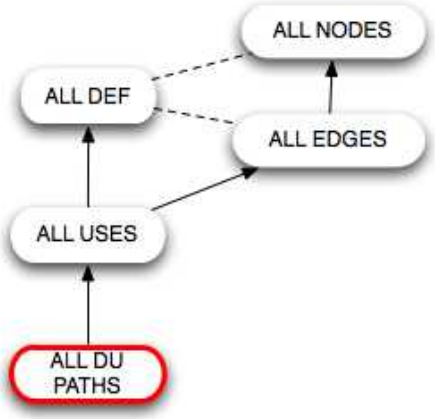


Andando a ricercare all'indietro troveremo un def riferito a quella stessa variabile (facciamo l'ipotesi che non si usi mai una variabile prima di averne fatto un def).

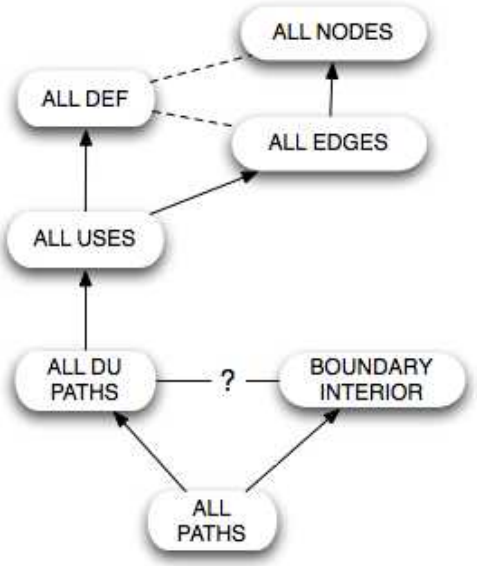


Dato che sto facendo *all uses*, devo per forza aver considerato anche quell'arco perché c'è un Puse della variabile che sto considerando.

Vediamo ora come si inserisce all du paths rispetto agli altri. Esso include *all uses*.

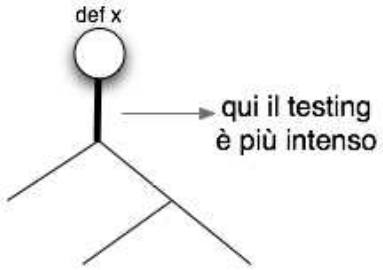


Possiamo ricostruire il quadro d'insieme:



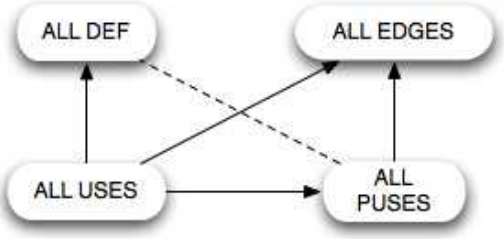
All paths include all du paths perché include anche i path ciclici.

- Consideriamo infine ALL PUSES. Si tratta di una variante di *all uses* che copre tutti gli usi nei predicati. *All uses* ha il difetto che le righe in cui sono presenti i def vengono esercitate molte volte; esso copre il codice in modo disomogeneo.



L'idea di all Puses è di cercare di ridurre il numero dei casi.

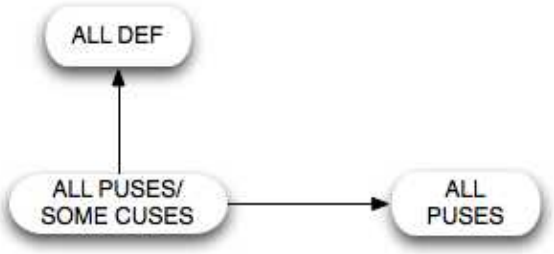
Esiste anche all Cuses, ma ovviamente, dovendo scegliere, i Puses sono più critici.



All Puses non include *all def*. Per esempio se c'è un def che non arriva mai a un Puse non viene esercitato.

Un'alternativa può essere ALL PUSES/SOME CUSES: per quei def che non hanno Puse, esercita almeno un Cuse.

All Puses/Some Cuses implica *all def*.



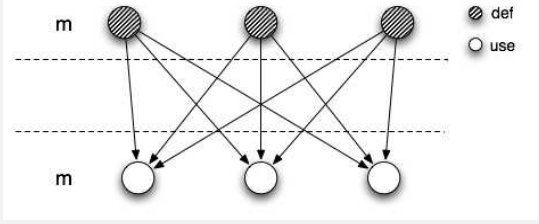
Analizziamo la complessità di questi criteri.
All def ha complessità O (N·k) dove k è il numero massimo di variabili definite in uno stesso nodo (deve essere generato un caso di test per ognuna).
All uses ha complessità O (N^2·k).
All du paths ha complessità O (2^N).

Alla fine se se fa data flow testing si usa *all uses*.

Dimostrazione che *all uses* ha complessità O (N^2·k)

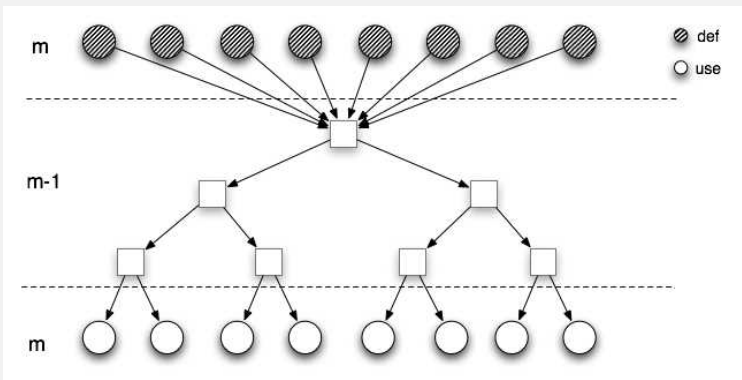
Vogliamo dimostrare che *all uses* è O (N^2·k), dove N è il numero dei nodi del CFG e la complessità è il numero di test per coprire il criterio.

In teoria potrei considerare come caso peggiore un grafo del tipo:



N=2m=O(m)
#du=m^2=O(m^2)

Ma devo tener conto che il grado di uscita può essere al massimo pari a 2 (i due rami di un costrutto if).
Allora il caso peggiore sarà qualcosa del tipo:



N=3m-1=O(m)
#du=m^2=O(m^2)

In conclusione i criteri da tenere in considerazione sono:

- *all uses*
- *all edges*
- *all nodes*

Ci possiamo domandare quale criterio sia più efficace tra *all uses* e *all edges*. Si possono fare degli esperimenti.
Sia *all uses*, che *all edges*, che qualsiasi criterio in generale, definiscono un criterio che può essere realizzato attraverso test suite diverse (è soddisfatto da più test suite).
Inoltre dobbiamo considerare che nella pratica un cammino non si traduce in un unico caso, in realtà posso realizzarlo con valori diversi; quindi ogni test suite può essere eseguita con una diversa sensitization (tutte sono equivalenti sull'astrazione ma non nella pratica). Un criterio di test è "buono" se, per qualsiasi test suite che lo soddisfa, esso trova i fault. Ovviamente nella pratica questo non esiste; ma una misura di quanto un criterio è buono può essere costituita da quante test suite trovano i faults.
Più precisamente possiamo definire la misura di *qualità di un criterio* C come il rapporto tra il numero di test suite che soddisfano C e trovano i fault e il numero di test suite che soddisfano C.

Questo modo di misurare la qualità di un criterio ha a che fare con la ripetibilità, con il fatto che non mi affido al caso.
Una volta definita una misura, posso fare degli esperimenti.
Ho bisogno però di definire un benchmark, perché in qualche modo la misura dipende dal codice.
I test effettuati hanno portato alla conclusione che *all uses* funziona meglio (in alcuni casi con confidenza statistica) di *all edges* ma *all uses* è molto fragile rispetto all'incompletezza del test. Cioè se entrambi vengono effettuati al 100% *all uses* funziona meglio, ma se abbasso la percentuale *all uses* risulta più fragile. *All uses* è più lento.

A questo punto risulta evidente il limite del testing strutturale, cioè la sua natura tautologica.

Consideriamo un esempio:

Codice corretto:

```
A. x=10;
…
B. if (x<15)…
```

Codice con errore:

```
A. x=100;
…
B. if (x<15)…
```

Il data flow testing (*all uses*) mi garantisce che ho eseguito un cammino che va dal punto A al punto B (con *all edges* non ho questa garanzia).

Ipotizziamo però un errore del tipo:

Codice corretto:

```
A. x=10;
…
B. if (x<15)…
```

Codice con errore:

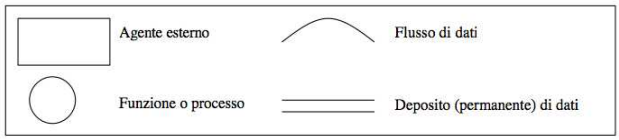
```
A. y=100;
…
B. if (x<15)…
```

In questo caso il data flow testing non trova l'accoppiamento! Il concetto è che non si può testare quello che non c'è.

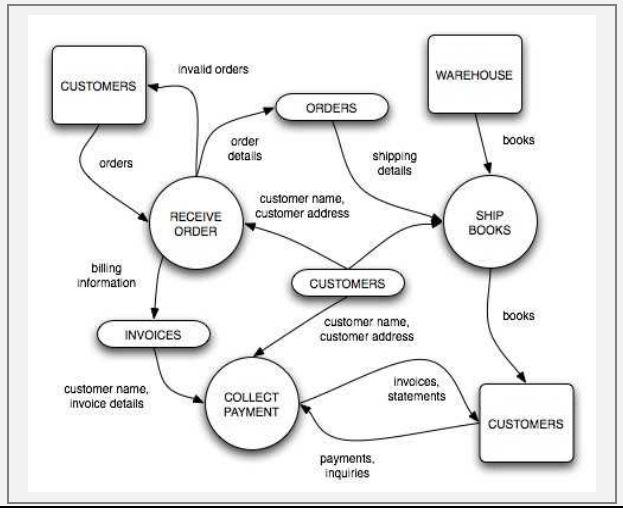
3.1 Control Flow Testing e Data Flow Testing in prospettiva funzionale

Ci possiamo chiedere se i criteri di testing visti fino ad ora si possano applicare anche in prospettiva funzionale. Per fare questo devo basarmi su delle specifiche, invece che sul codice; le specifiche allora devono essere fatte in modo che si possa costruire un CFG a partire da esse. Questa caratteristica non è molto comune; si può avere per esempio in casi di applicazioni di workflow o se lavoro su un data flow diagram, che mette in evidenza proprio le dipendenze tra i dati.

Data Flow Diagrams
Il Data Flow Diagram (abbreviato in DFD) è un tipo di diagramma definito nel 1978 da Tom De Marco nel testo Structured Analysis and Systems Specification per aiutare nella definizione delle specifiche.
É una notazione grafica molto usata per i sistemi informativi e per la descrizione del flusso di dati in quanto permette di descrivere un sistema per livelli di astrazione decrescenti con una notazione di specifica molto "intuitiva".
Attraverso i Data Flow Diagram si definiscono soprattutto come fluiscono (e vengono elaborate) le informazioni all'interno del sistema, quindi l'oggetto principale è il flusso delle informazioni o, per meglio dire, dei dati. Motivo per il quale diventa fondamentale capire dove sono immagazzinati i dati, da che fonte provengono, su quale fonte arrivano, quali componenti del sistema li elaborano. Oltre alle funzioni (rappresentati con cerchi), i data flow diagram possono includere agenti esterni al sistema, rappresentati mediante elementi rettangolati e non modellati ulteriormente, depositi di dati da usare come sorgente o destinazione di informazione permanente, rappresentati con coppie di linee parallele, e flussi di dati, rappresentati mediante archi orientati, scambiati tra funzioni oppure tra una funzione e un componente di tipo diverso.



Esempio



4. Finite State Testing

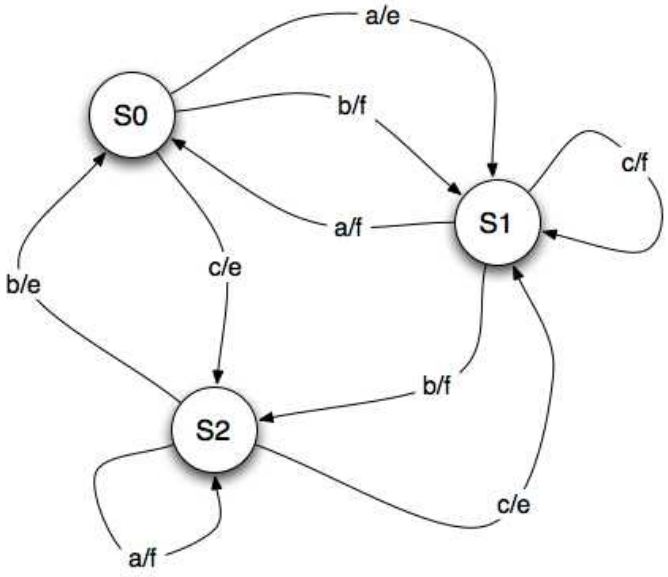
Il *Finite State Testing* è un modello usato per i sistemi reattivi. Un sistema reattivo è un sistema che deve fornire una risposta continua nel tempo agli eventi di un ambiente che è solo parzialmente decidibile.

Molto spesso la specifica di un sistema reattivo ha come parte fondamentale una *macchina a stati*.
Vediamo un esempio di macchina a stati.
Abbiamo:

- un insieme di stati $\{S_0, S_1, S_2\}$, di cui uno è lo stato iniziale (S_0)
- un insieme di input {a, b, c}
- un insieme di output {e, f}

Facciamo alcune ipotesi aggiuntive:

- la macchina che consideriamo è completamente specificata, cioè in ogni stato si possono ricevere tutti gli input
- la macchina è osservabile: per ogni input la macchina emette un output
- la macchina è deterministica: non è possibile che, a partire dallo stesso stato, l'applicazione dello stesso input produca nella macchina due diverse uscite.



Nel caso del finite state testing abbiamo due macchine a stati:

- S rappresenta la specifica
- I rappresenta l'implementazione

L'implementazione, comunque sia fatta, dal punto di vista astratto può essere sempre vista come una macchina a stati (cioè posso immaginare di avere una macchina a stati che rappresenta lo stato dell'implementazione).

4.1 Conformance testing

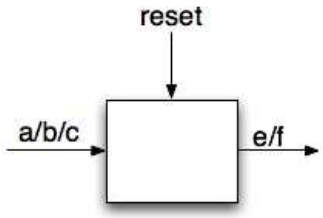
Il *Conformance Testing* consiste nel confrontare I con S e vedere se esse sono tra loro equivalenti. È un caso di testing funzionale. Potremmo effettuarlo anche in prospettiva strutturale, nel caso in cui si implementi il codice come un macchina a stati (per esempio secondo il pattern state).

Definiamo il concetto di *equivalenza di macchine a stati finiti*: se V è un insieme di sequenze di ingressi, due stati sono *V-equivalenti* se restituiscono la stessa sequenza di uscite per ogni sequenza in V.
Due *stati* sono *equivalenti* se sono V-equivalenti per qualsiasi insieme V.
Due *macchine* sono *equivalenti* se i loro stati iniziali sono equivalenti.

Quindi il problema diventa stabilire se lo stato iniziale di I è equivalente allo stato iniziale di S.

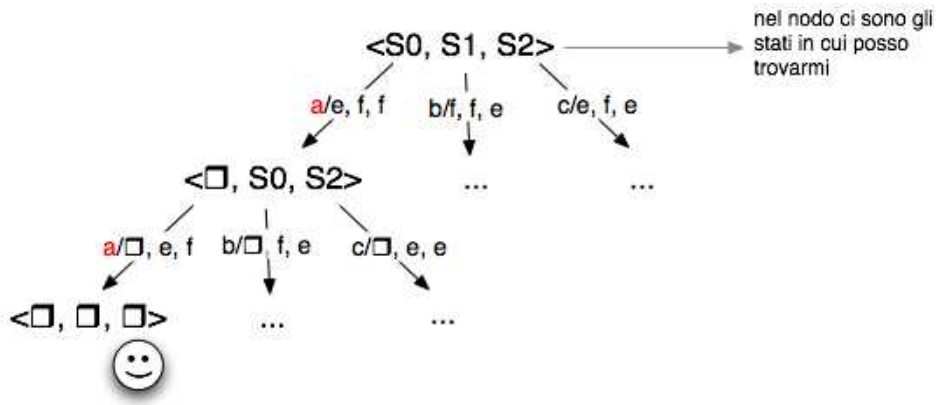
Una tecnica con cui si risolve questo problema ha il nome di *W-method*.

L'implementazione è una black box: posso soltanto osservare l'output che emette. Quindi il problema è scoprire in che stato si trova la macchina I in ogni istante. Tutto quello che so è che dopo aver resettato mi trovo nello stato iniziale.



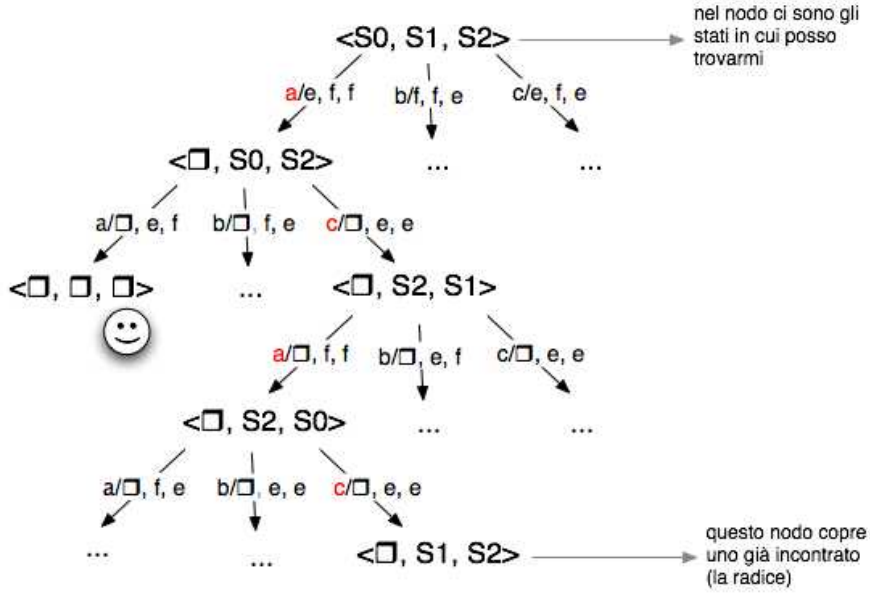
Il primo problema è quindi quello di trovare una serie di ingressi che mi permettano di identificare lo stato.
Bisogna trovare una sequenza di ingressi che se applicata a uno stato mi permette a posteriori di capire lo stato in cui ero.
Nel nostro esempio se applico l'ingresso a ottengo e se ero nello stato S_0, mentre ottengo f sia che fossi nello stato S_1 che S_2. Quindi se ottengo e ho identificato lo stato in cui ero (S_0), ma se ottengo f non sono in grado di identificare in quale stato ero tra S_1 e S_2; il solo ingresso a non è sufficiente per distinguere tra tutti e tre i casi.

Posso andare avanti ad applicare altri ingressi, costruendo un albero i cui nodi contengono gli stati in cui via via posso trovarmi.



Tutte le volte che uno stato emette un'osservazione diversa da tutti gli altri, allora quello stato risulta identificato.
Nell'esempio la sequenza aa identifica univocamente qualsiasi stato.

N.B. Può esistere il caso di una sequenza infinita che non disambigua mai tutti gli stati. Nell'esempio abbiamo la sequenza {acac}*.



Possiamo fare due considerazioni riguardo all'esempio:

- le sequenze aa e ab identificano tutti gli stati
- esistono sequenze infinite che non identificano lo stato iniziale

Si può però dimostrare che l'algoritmo di costruzione dell'albero termina sempre; questo perché arriverò ad uno stato che "copre" uno già incontrato, dove l'algoritmo si ferma (con questa ipotesi si dimostra che l'albero è finito).

Posso anche pensare di applicare due sequenze diverse. Per esempio se applico le due sequenze a e b (e tra l'una e l'altra riporto la macchina allo stato precedente) identifico comunque tutti gli stati. Per riportare la macchina allo stato precedente la resetto e ripeto poi la sequenza di ingressi che mi aveva portato a quello stato (ho la certezza che torno allo stesso stato perché la macchina è deterministica)

Dobbiamo definire il *fault model*.
L'implementazione può presentare due tipi di fault:

- output fault: la macchina trasferisce correttamente il controllo ma emette l'output sbagliato
- transfert fault: la macchina emette l'output corretto ma trasferisce il controllo allo stato sbagliato
- i due errori possono anche essere combinati

Dobbiamo notare che può presentarsi anche un altro tipo di errore: l'implementazione potrebbe avere degli extra states, cioè degli stati non previsti nella specifica.

4.2 W-Method

Il W-Method si basa su una serie di concetti:

- *Characterization Set* W

  $W \in 2^{I^*}$ è un insieme di sequenze di ingressi (di lunghezza non prefissata), sufficienti insieme a disambiguare lo stato iniziale (con I insieme degli ingressi e I* le possibili sequenze di ingressi).
  Es. W={a,b} due sequenze lunghe 1
  W={aa}
  W={ab} una sequenza lunga 2

  Non è detto a priori se sia meglio usare più sequenze corte o una sola sequenza lunga, bisogna vedere quant'è difficile resettare il sistema.
- *State Cover Set* Q

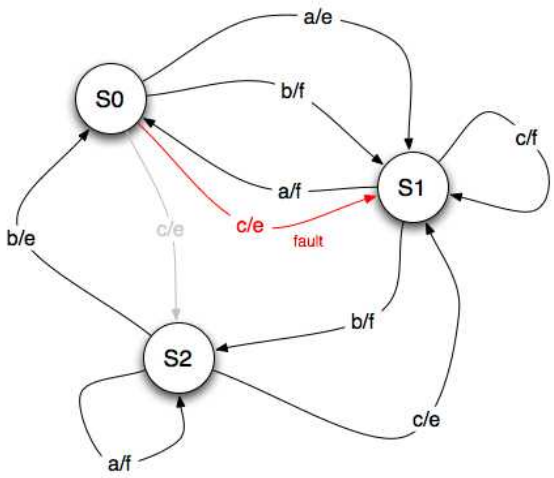
  Q è un insieme di sequenze che copre tutti gli stati (dopo un reset)
  Per esempio nella macchina che abbiamo esaminato può essere Q={λ, b, c}: con λ (input nullo) arrivo in S_0, con b in S_1 e con c in S_2 (se l'implementazione è conforme).
- Q×W è la concatenazione di tutti gli elementi di Q con tutti gli elementi di W (l'insieme di tutte le sequenze formate da prefisso e suffisso dove il prefisso appartiene a Q e il suffisso appartiene a W).
  Es {λ, b, c}×{a, b}={a, b, ba, bb, ca, cb}

Criteri di copertura

- ALL NODES

Se eseguo le sequenze di ingressi di Q×W, il prefisso di ogni sequenza (cioè l'elemento di Q) mi garantisce di andare in un certo stato, dopodichè il suffisso (l'elemento di W) mi permette di dire in che stato sono arrivato, cioè se arrivo o no nello stato in cui devo arrivare se l'implementazione è conforme.
In altre parole, applicando Q, se la macchina è conforme, ho portato la macchina in tutti gli stati; ma io devo proprio verificare se la macchina è conforme, allora in coda ad ogni elemento applico tutte le sequenze del characterization set; dopo questi due esperimenti sono in grado di dire se effettivamente l'ingresso applicato mi porta allo stato previsto.

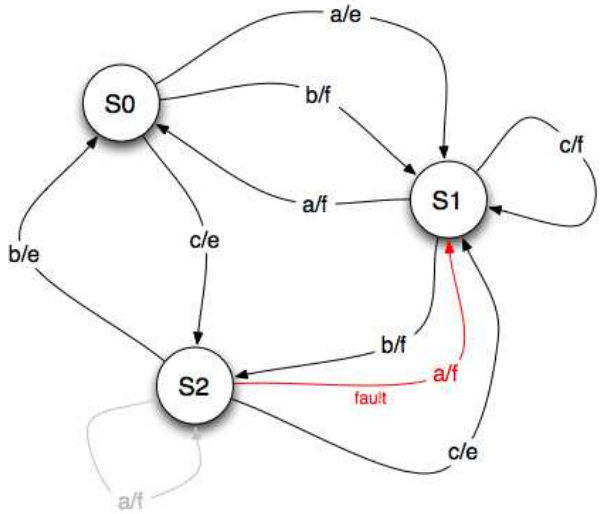
Per esempio supponiamo di avere il seguente errore nell'implementazione:



Immaginiamo di applicare Q; per costruzione in Q deve esserci una sequenza che mi porta in S_2 (c); ma siccome c'è un errore, quando, dopo aver applicato c, applico la sequenza di W, avrò in uscita la sequenza di uscita caratteristica di S_1 invece che di S_2.
Es. applico le due sequenze ca e cb:

Su I:	Su S:
c->e	c->e
a->f	a->f
c->e	c->e
b->f	b->e

Ho però trovato l'errore perché corrispondeva a una delle sequenze di Q. Immaginiamo che l'errore fosse stato "dopo":



Questa test suite non è in grado di osservare questo errore perché essa arriva in tutti i nodi e verifica dove arriva, ma non è detto che vada a vedere tutti gli archi quindi potrebbe non trovare un errore su un arco.

- ALL EDGES

Definiamo il *Transition Cover Set* P: è l'insieme delle sequenze di ingresso che garantisce di coprire tutti gli archi (e i nodi) della macchina. Es {λ, a, b, bc, ba, bb, c, ca, cc, cb}
Come test suite utilizzo in questo caso P×W. Questo set è in grado di trovare l'errore dell'esempio precedente. Per costruzione ogni edge è coperto da almeno un caso in P.
P×W={…, caa, cab, …}

Su I:	Su S:
c->e	c->e
a->f	a->f
a->f	a->f
c->e	c->e
a->f	a->f
b->e	b->f

Questa tecnica, sotte le ipotesi che abbiamo fatto, è esaustiva; si parla di *full fault coverage*: qualsiasi implementazione del W-method (con copertura all edges) copre tutti i fault.

Questa tecnica mostra un limite di complessità (ma è polinomiale, può essere accettabile); bisogna però considerare che in alcuni casi (per esempio in sede contrattuale) è un'operazione che si fa per lo più "a mano", la fa una persona, quindi può essere un problema se i casi di test sono troppi.

4.3 Wp-method

Si possono fare delle ottimizzazioni rispetto alla tecnica base. Un esempio è il Wp-method, dove p sta per partial. Possiamo partire dall'osservazione che in realtà, dopo che ho applicato una sequenza in Q, non è necessario applicare tutto il characterization set: posso scegliere le sequenze da applicare in base allo stato che sto considerando. Allora nel Wp-method invece di considerare Q×W, facciamo Q⊗W, dove ⊗ significa "concatenato speciale": concateno Q con un insieme di sequenze che scelgo per i singoli stati.
Nel nostro esempio:
Q⊗W={{λ}×{a}, {b}×{a,b},{c}×{b}}={a, ba, bb, cb}

Quando realizzo *all nodes* risparmio qualche caso, ma dove è più evidente il vantaggio è quando vado a fare copertura *all edges*. Nell'esempio P={λ, a, b, bc, ba, bb, c, ca, cc, cb}

P=	{λ,	a,	b,	bc,	ba,	bb,	c,	ca,	cc,	cb}=
	S_0	S_1	S_1	S_1	S_0	S_2	S_2	S_2	S_1	S_0

={	{λ, ba, cb}, portano in S_0	{a, b, bc, cc}, portano in S_1	{bb, c, ca} portano in S_2	}

Con P⊗W, concateno ogni sottoinsieme con l'insieme che caratterizza lo stato in cui portano i suoi elementi:
P⊗W={{λ, ba, cb}×{a},{a, b, bc, cc}×{a,b}, {bb, c, ca}×{b}}=
={a, baa, cba, aa, ab, ba, bb, bca, bcb, cca, ccb, bbb, cb, cab}

Risparmio 6 casi rispetto al W-method, e continuo a mantenere la condizione di full fault coverage.

Nel W-method ho un characterization set W (W={a,b} nell'esempio); nel Wp-method invece ho un identification set W_i per ogni stato: W_i è un insieme di sequenze di ingressi sulle quali lo stato S_i da una risposta unica.
Nell'esempio:
W_0 ={a}
W_1 ={a,b}
W_2 ={b}
Nel Wp-method il characterization set W invece di essere un insieme è un insieme di insiemi.
W={ W_0 , W_1 , W_2 }={{a},{a,b},{b}}

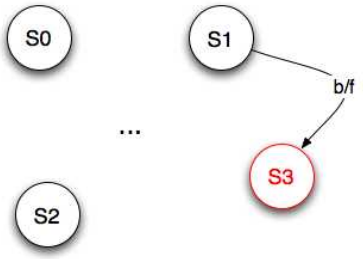
Il Wp-method riduce i casi perché invece di fare il prodotto fa la somma dei casi: riduce di 1 la dimensione.

4.3.1 Ulteriori considerazioni

Noi abbiamo ipotizzato che siamo in grado di fare il reset della macchina; tutto questo si basa sull'assioma che l'operazione di reset sia corretta. La vera ipotesi che facciamo è che il reset ci porti sempre nello stesso stato; anche se questo stato non è S_0, scopriamo comunque che c'è l'errore, anche se dipende dal reset. Il reset mi deve garantire di aver "tolto memoria".

In conclusione il Wp method fa fault coverage completo
Esso richiede però alcune ipotesi:

- Reset corretto
- FSM deterministica, sia la specifica (ok) che l'implementazione (più problematico)
- Osservabilità:
  Se ci sono delle transizioni che non mi danno output (evento non osservabile), ho a che fare con un *sistema caotico*, cioè non sono in grado di osservare l'istante in cui è stata presa la scelta. Se ci sono degli eventi che non si osservano il sistema potrebbe essere "andato a giro" per gli stati senza che me ne accorgessi. Un sistema non deterministico non è necessariamente caotico.
- Non ci sono extra-states
  Potrei avere un'implementazione con degli stati in più rispetto alla specifica.



Lo stato S_3 potrebbe comunque essere equivalente a S_2; in tal caso l'implementazione I sarà comunque equivalente alla specifica S, anche se I non sarà minima. Ma S_3 potrebbe anche non comportarsi come S_2.
Il problema è che essere arrivato sugli extra-states allunga la sequenza dei comportamenti: il characterization set non è in grado di distinguere gli extra-states. Potrebbe esistere uno stato S_3 in grado di rispondere allo stesso modo di S_2 al characterization set ma che non è equivalente a S_2 (la dimensione del characterization set ha a che fare con il numero dei nodi).
Chiamiamo N il numero di stati di S e M il numero di stati di I.
Se M<N la macchina non è in grado di rappresentare lo stesso linguaggio e quindi mi accorgo dell'errore.
Se M>N anziché fare $P \times W$ si fa $P \times I^{M-N} \times W$: con P avevo la garanzia di coprire tutti gli archi, ma siccome ci sono degli extra state, con P ho adesso la garanzia di "rompere la macchina", cioè di portarmi fuori dagli stati previsti; adesso gli concateno tutti i possibili ingressi (sequenze lunghe quanto il numero di extra state) in tutti i possibili ordini. Dopo che ho fatto questo ho coperto tutto quello che c'è nella macchina; se c'è un output fault me ne accorgo, altrimenti devo applicare W per vedere se sono arrivato nello stato giusto.
I^{M-N} è esponenziale rispetto al numero degli extra state con base pari al numero degli input. Il Wp-method può ridurre un po' il numero dei casi, ma arrivati a questo punto la riduzione è praticamente irrilevante.
Il problema però è che il realtà non sappiamo quanti sono gli extra state; per applicare questo metodo devo ipotizzare un numero di extra state massimo.

- La macchina è completamente specificata, cioè è in grado di accettare qualsiasi input in qualsiasi stato.

5. OO Testing

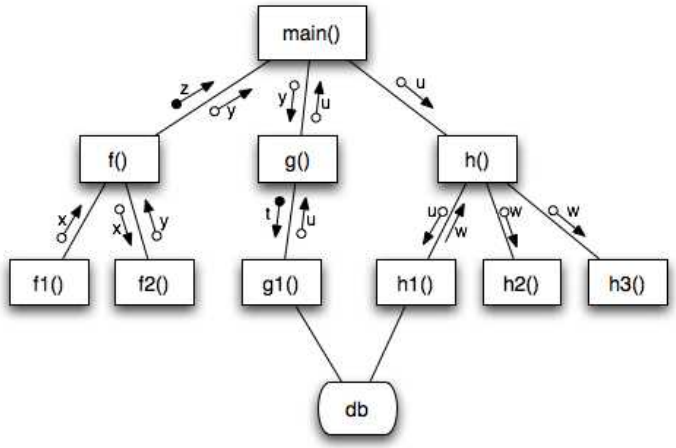
L'*OO Testing* è un caso speciale di testing, con proprie particolari caratteristiche. La teoria e l'esperienza relative all'OO testing sono meno consolidate rispetto all'ambito della programmazione strutturata (*SP-Structured Programming*); questo sia per il fatto che la programmazione a oggetti è più recente, sia per il fatto che essa è meno rilevante nei contesti safety critical dove il testing è prescritto, contesti nel quale la transizione da C a C++ è ora in atto.
Altra caratteristica peculiare è il ruolo speciale del testing in eXtreme Programming (XP). L'XP si basa sul principio di Test First; sono offerti strumenti specifici di supporto come Junit, CppUnit. XP comprende concetti come il supporto al regression testing, refactoring, proprietà condivisa, integrazione frequente etc, ma non porta con se una specifica metodologia di test selection.
La programmazione a oggetti aggiunge alla programmazione strutturata vari fattori di complessità:

- il flusso di controllo è molto più complesso in OOP
- gli oggetti hanno uno stato, ed i valori che compongono questo stato determinano il comportamento dell'oggetto stesso
- lo stesso oggetto può offrire servizi a diversi altri oggetti (concorrenza)
- si ha visibilità globale all'interno di un oggetto
- il polimorfismo complica ulteriormente le cose

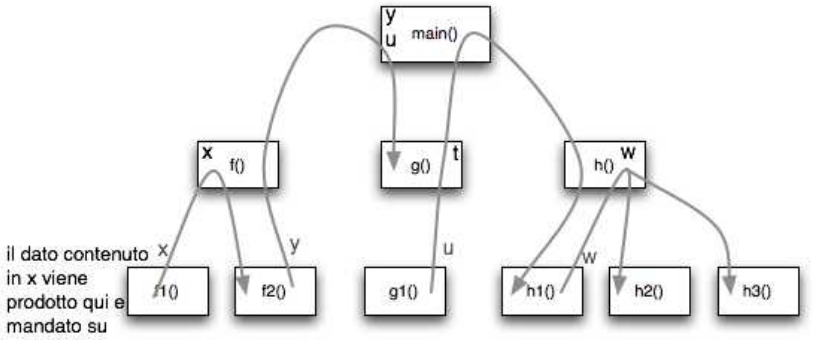
5.1 Fattori di complessità della programmazione a oggetti

5.1.1 Flusso di controllo

In SP il codice è organizzato in una gerarchia di funzioni; la gerarchia realizza una funzione ("trasformazione") o un insieme di funzioni smistato da uno o più "centri di transizione". Spesso si realizza una confluenza sui livelli bassi (riuso sul livello di libreria). La carta strutturata fornisce una chiara visione di come gira il controllo ed esiste un modello di riferimento su come far girare il controllo.

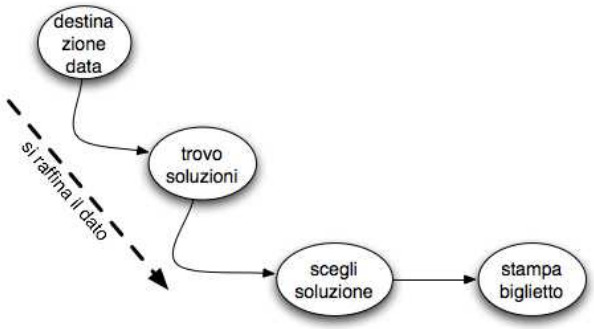


Il disegno rappresenta la gerarchia di un programma C: g() è la funzione principale, essa ha bisogno del dato contenuto nella variabile y; il modulo f() fornisce il dato di cui g() ha bisogno. Il modulo f() non ha validità funzionale, ma solo strutturale.
Il flusso tende a muoversi in questo modo:

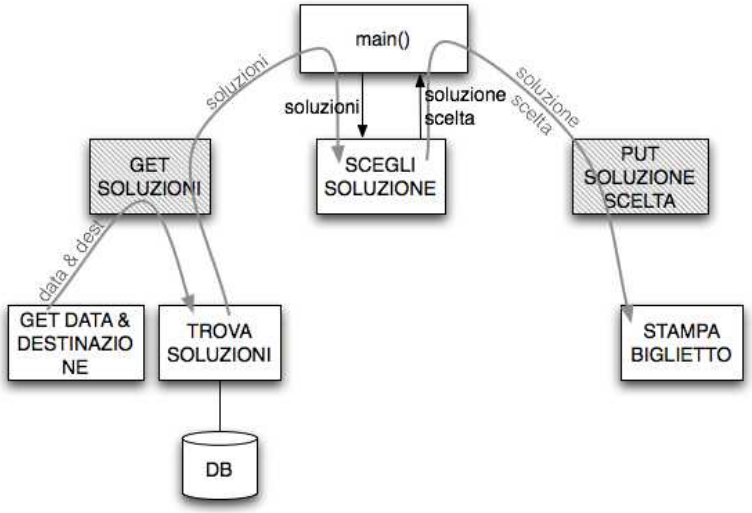


Ogni variabile è allocata nel punto più alto in cui viene usata; così per esempio y e u sono allocate al livello del main.

Vediamo un esempio concreto; consideriamo un programma che produce un biglietto di un aeroplano. Attraverso il seguente diagramma determiniamo le responsabilità:



Il main program deve chiamare la funzione più importante, cioè quella che ha il più alto livello di astrazione, in questo caso “scegli la soluzione” è il livello più logico.



Il modulo “GET SOLUZIONI” non ha responsabilità funzionale, non calcola niente; ha responsabilità strutturale: mette in fila i moduli che hanno responsabilità funzionale; il modulo che ha responsabilità funzionale è quello che fa la query sul database per esempio.

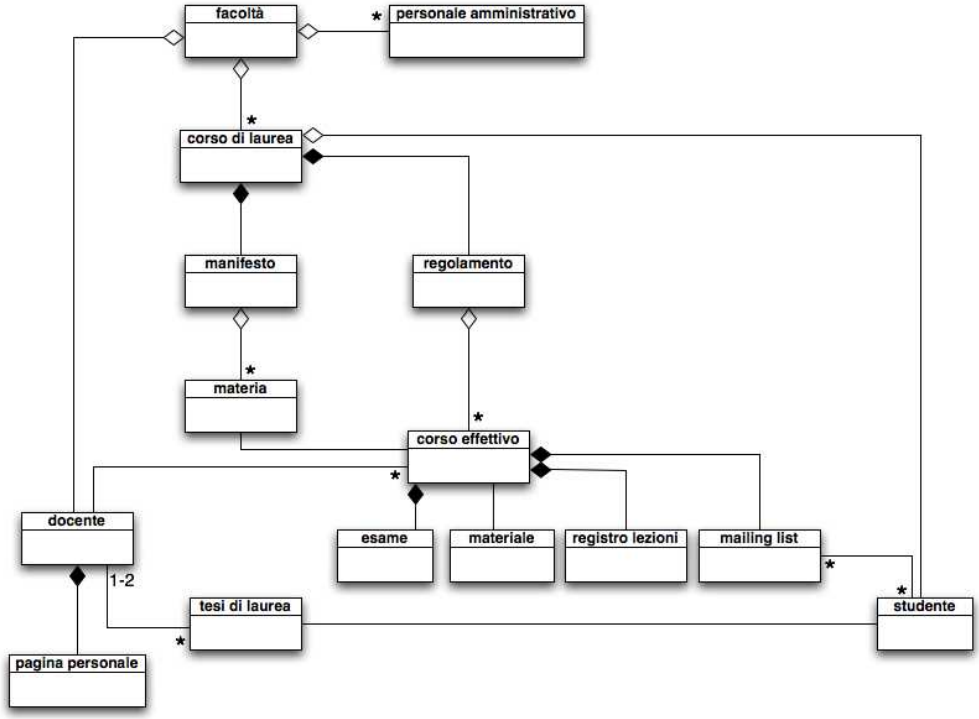
Per esempio il codice di “GET SOLUZIONI” potrebbe essere qualcosa del genere:

```
void getSoluzioni(≈* soluzioni)
                    {
                  ≈ data;
                ≈ destinazione;
      getDataDestinazione(&data, &destinazione);
    trovaSoluzioni(data, destinazione, soluzioni);
                    }
```

Anche il modulo “PUT SOLUZIONE SCELTA” ha responsabilità strutturale.

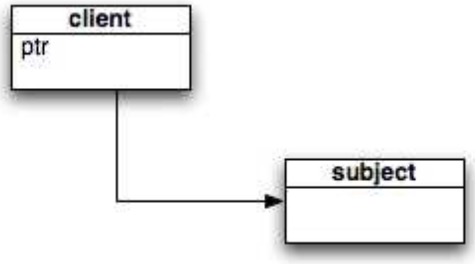
In un’applicazione strutturata la massima aspirazione per una gerarchia è quella di realizzare una *coesione funzionale*; questo significa che sono in grado di identificare *la* funzione realizzata dalla gerarchia.

Quando si passa agli oggetti la situazione si complica. L’astrazione più corretta per comprendere la programmazione a oggetti è il modello client-server; con la OOP, non abbiamo più una gerarchia ma invece una *rete* di oggetti (classi), ciascuno dei quali assume delle *responsabilità*. La rete non è progettata per realizzare una sola funzione, ma anzi è in grado di realizzarne diverse, e il flusso gira in modo diverso a seconda della funzione.



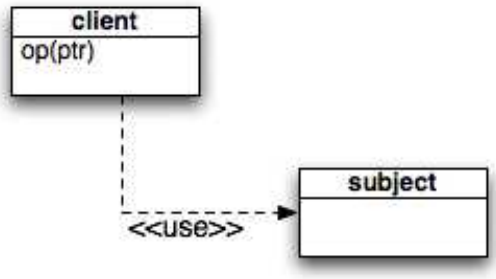
La rete in figura può svolgere più computazioni, corrispondenti a più scenari d’uso; ciascuno corrisponde ad un diverso sequence diagram (per esempio aggiungiEsame, trovaDocente, elencaMaterie, etc) e non è strettamente specificato dalla vista statica del class diagram.
Il controllo non gira secondo un modello predefinito; ciascun oggetto trasferisce il controllo all’oggetto che detiene la responsabilità delegata e non ad un subordinato (come nel caso SP). Tra il delegante e il delegato spesso esiste una relazione d’uso dinamica e non strutturale.

Relazione strutturale:



Nell'oggetto client esiste un attributo che rappresenta questa relazione (l'oggetto mantiene nel suo stato questa relazione).

Relazione d'uso:



Un altro oggetto chiama un metodo pubblico dell'oggetto client e gli passa un puntatore all'oggetto subject.

Spesso la dipendenza è una relazione d'uso, che non è rappresentata in modo forte nel codice (non è facile vederla staticamente).

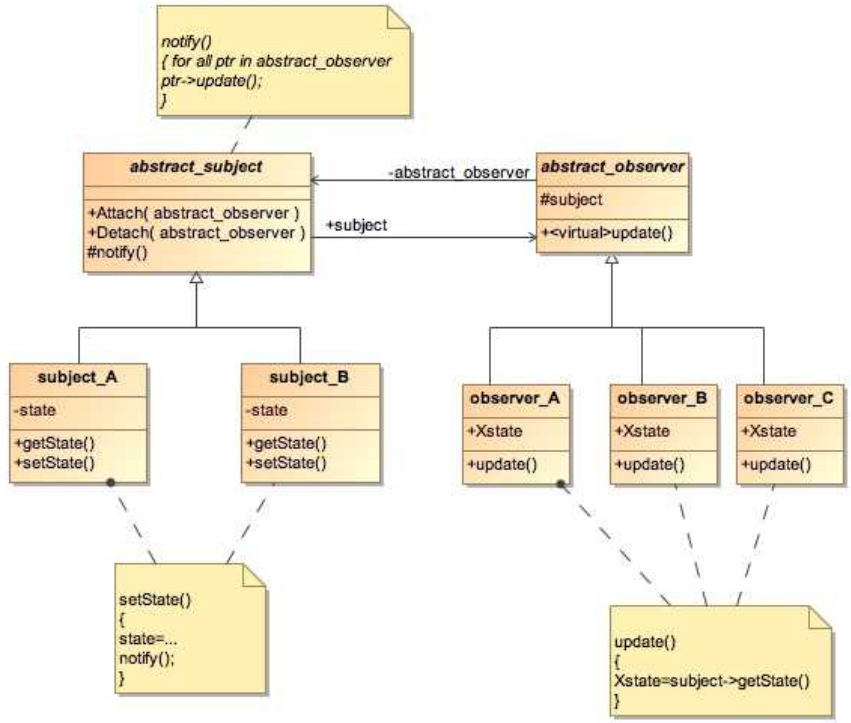
È dimostrato che la programmazione a oggetti favorisce l'evolvibilità, il riuso, la produttività:

- un'architettura è capace di assorbire una varietà di responsabilità coesive sui dati che tratta, e non una singola funzione
- è sempre possibile aggiungere una responsabilità ad una classe, o aggiungere una classe
- è sempre possibile attraversare la rete degli oggetti in qualche modo diverso per rispondere ad un nuovo caso d'uso

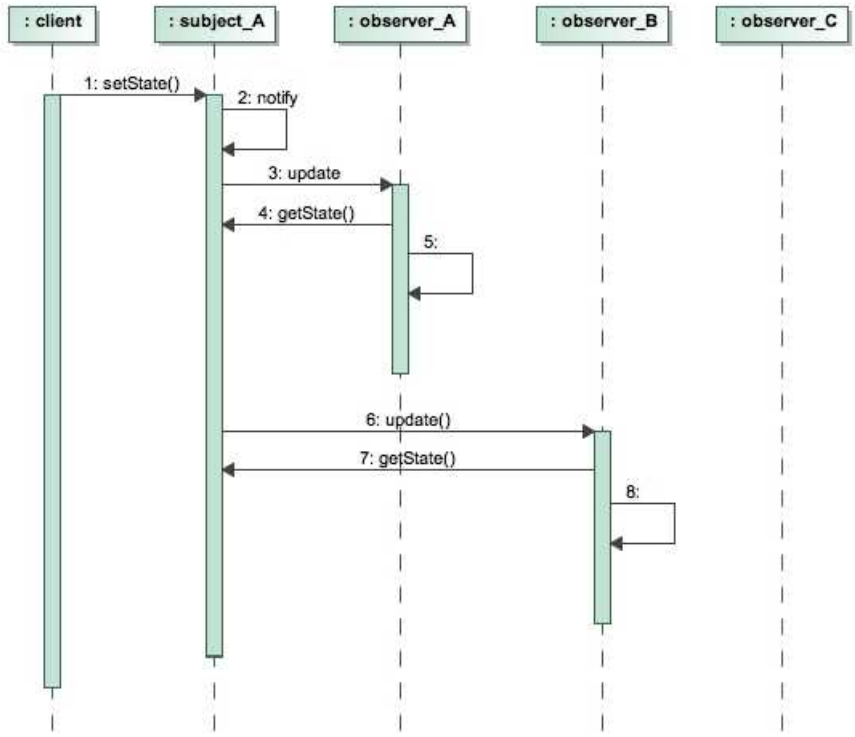
Queste caratteristiche rendono la programmazione a oggetti molto più flessibile rispetto alla programmazione strutturata, dove l'intera gerarchia è finalizzata a svolgere una funzione con responsabilità coesive in senso funzionale. Ma tutto questo evidentemente complica il testing.

La programmazione a oggetti è inoltre spesso caratterizzata da meccaniscmi di interazione complessi, che complicano ulteriormente le cose; per esempio la callback: un oggetto chiama un metodo di un secondo oggetto passandogli il proprio indirizzo, che il secondo oggetto userà per chiamare un metodo del primo; in ultimo si tratta di una ricorsione.

Esempio: il pattern *observer* definisce una dipendenza uno-a-molti per cui quando un oggetto modifica il suo stato tutti i suoi dipendenti ne ricevono notifica.



Si ha installazione dinamica con responsabilità a carico dell'observer: l'observer si registra sul subject con attach/detach; il subject notifica i suoi cambi di stato agli osservatori registrati tramite il notify; in ricezione del notify, l'observer invoca `getState()`.



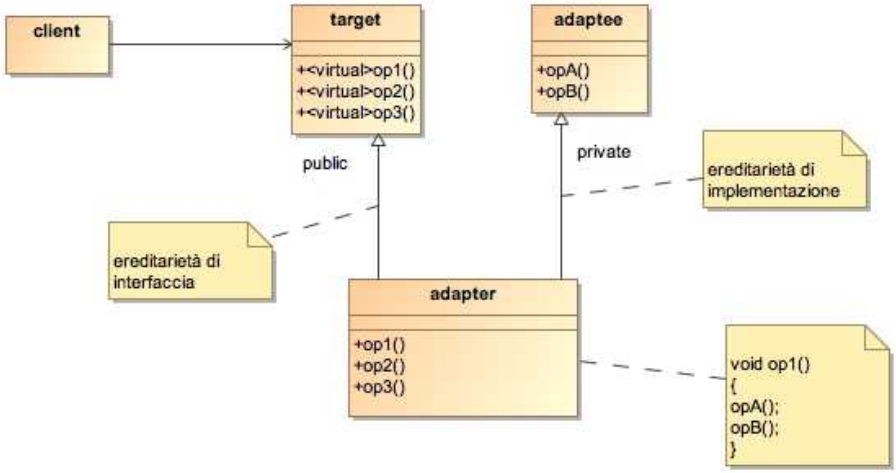
Da questo esempio capiamo che il flusso di controllo può essere molto complicato.

5.1.2 Topologia della rete degli oggetti

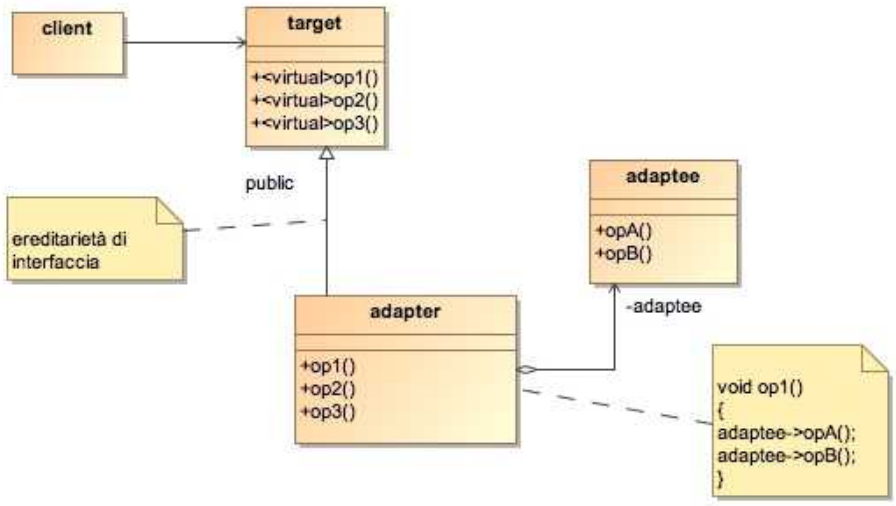
Nella programmazione strutturata la gerarchia è sostanzialmente statica (la funzione principale ne chiama un'altra etc, tutto è definito staticamente nel codice); nella programmazione a oggetti troviamo una topologia dinamica, largamente dipendente da scelte prese al tempo di esecuzione: gli oggetti vengono istanziati in modo dinamico, possono avere tipi diversi e essere configurati e composti in modo diverso. La programmazione nello stile dei "design patterns" crea astrazione proprio sul modo con cui viene poi configurata la rete degli oggetti.
Il prevalere dell'uso della delega rispetto all'ereditarietà esacerba il problema.

Esempio: il pattern *adapter* converte l'interfaccia di una classe già implementata per adattarla all'interfaccia attesa da un cliente; per esempio può essere il caso di un adattamento verso un'interfaccia già disponibile realizzata da una precedente applicazione.

Class Adapter: realizza questo adattamento attraverso l'ereditarietà.



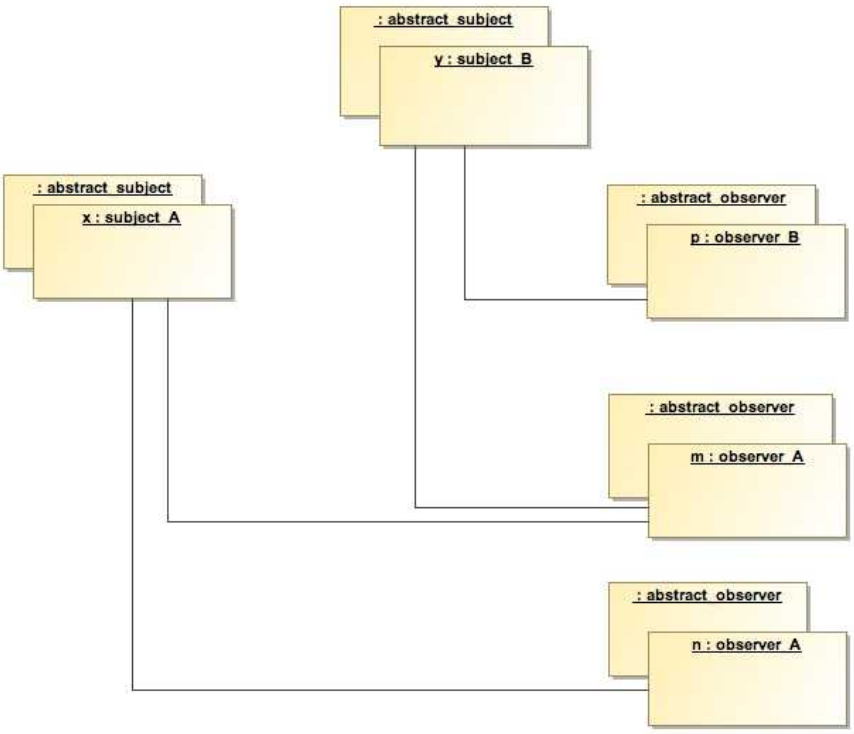
Object Adapter: realizza l'adattamento attraverso la composizione.



Il Class Adapter è più limitato dal punto di vista della progettazione; per esempio l'adapter non eredità un eventuale override dell'adaptee. L'Object Adapter può utilizzare implementazioni diverse dell'adaptee, l'adaptee può anche essere sostituito dinamicamente (in principio); l'Object Adapter risulta però più complicato per il debugging.

5.1.3 Stato e concorrenza

Nella programmazione a oggetti gli oggetti hanno uno stato e subiscono concorrenza.
Nella programmazione strutturata le variabili locali ad una funzione terminano il tempo di vita con la funzione: la funzione non ha stato (a parte un limitato uso della direttiva static). Nella programmazione a oggetti un oggetto incapsula attributi che mantengono il loro valore anche quando il controllo non risiede in uno dei metodi dell'oggetto (stato). Inoltre gli attributi hanno visibilità globale entro una classe (sono variabili globali all'interno della classe). Lo stesso oggetto può essere usato da oggetti diversi e anche di tipo diverso (concorrenza).
Per esempio nel pattern observer, gli oggetti observer sono rappresentati dalla lista degli observer (a cui si aggiungono chiamando attach) e sono oggetti di tipo diverso tra loro (anche i subject sono diversi).



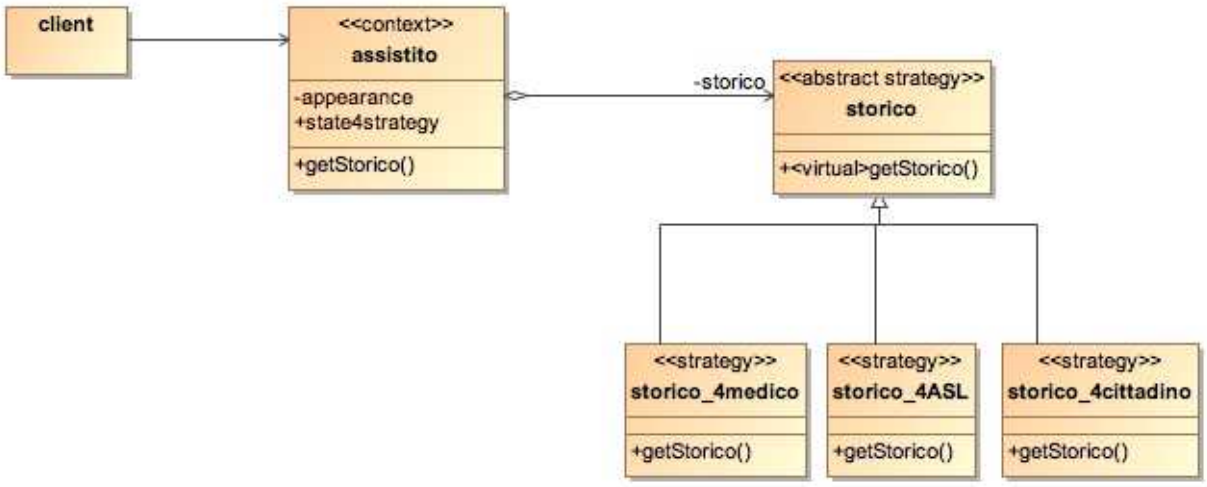
Gli oggetti hanno sostanzialmente visibilità globale, è sufficiente conoscerne l'indirizzo; in sostanza si realizza un uso pervasivo di quello che in C sarebbe un puntatore a funzione.

5.1.4 Polimorfismo

La stessa operazione può avere implementazioni polimorfiche; questo significa che oggetti che stanno dietro alla stessa interfaccia possono esibire comportamenti diversi: un client chiama un metodo di un oggetto ma non sa quello che c'è dietro (questo deriva direttamente dalla ricerca di un disaccoppiamento).

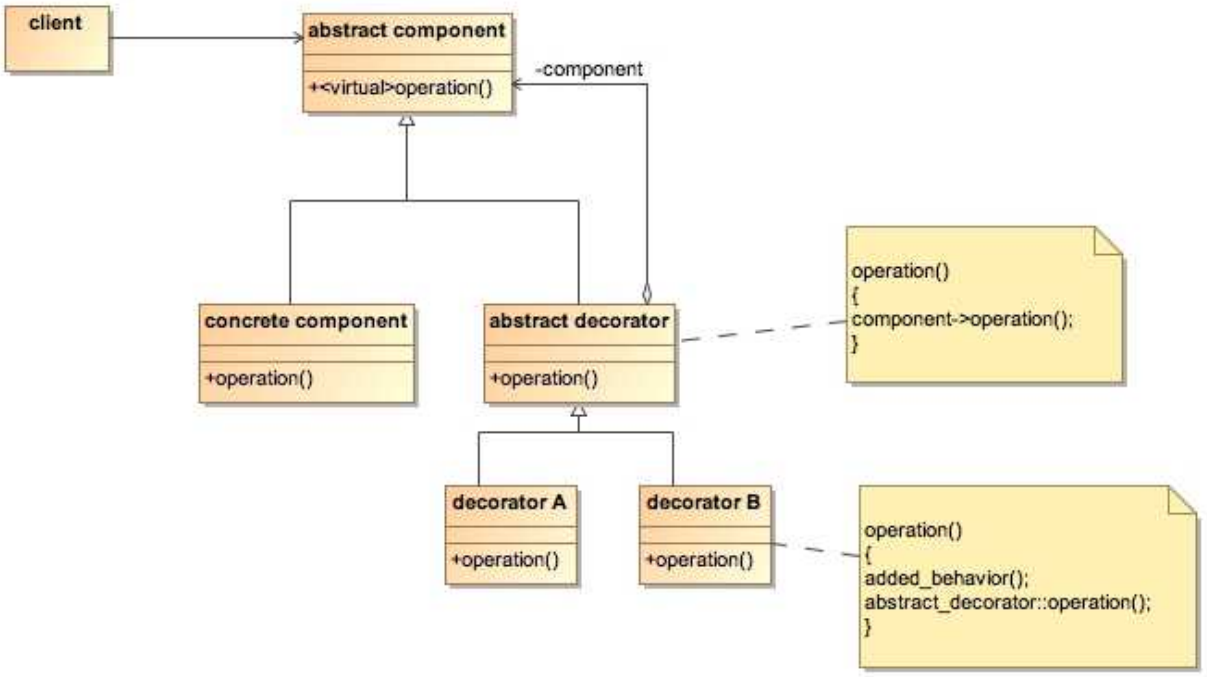
Il primo meccanismo del polimorfismo permette di istanziare la classe che implementa una funzione in forme diverse e a seconda di come la istanzio la stessa classe si comporta diversamente.

Esempio: il Pattern Strategy permette di isolare una funzione all'interno di un oggetto. Il pattern Strategy è utile in quelle situazioni dove è necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione. Per esempio se devo fornire un servizio, il metodo che lo fa mette in vita un oggetto o un altro a seconda delle risorse disponibili al momento (per esempio lo stato della rete nel caso offra un servizio tramite Internet).



Nell'esempio il problema è che il tipo assistito ha la responsabilità di recuperare lo storico delle prestazioni sanitarie di un assistito; l'operazione è implementata in modo diverso a seconda che il client sia il cittadino stesso, il suo medico attuale, l'amministrazione della ASL competente, etc.

Il secondo meccanismo del polimorfismo permette di cambiare anche dinamicamente l'oggetto che implementa la funzione. Per esempio il Pattern Decorator permette di avere diverse configurazioni delle deleghe.



5.2 Testing con OOP

Questa complessità tipica dell'OOP può essere in qualche modo limitata seguendo il principio detto *Design for testability*; questo significa tenere conto dei problemi del testing nella fase di progetto. Alcuni esempi possono essere non usare allocazione dinamica di oggetti, usare vettori di oggetti al posto delle liste, etc. In quest'ottica si rischia però di perdere la potenza stessa della programmazione a oggetti. Si tratta allora di trovare una giusta "via di mezzo".

Quello che possiamo fare è applicare i principi del testing per la programmazione strutturata adattandoli a diversi aspetti del problema.
Possiamo effettuare il testing a livelli diversi, con diverse combinazioni di approccio funzionale e stutturale:

- Si fa *unit testing* sui singoli metodi, applicando le stesse tecniche viste per la programmazione strutturata; si fa *unit testing* sulla singola classe. Lo unit testing sulle classi e sui metodi funziona bene se lo fa direttamente il programmatore.
- Il *system testing* è eseguito a livello più "alto": non si possono testare tutti i singoli accoppiamenti di variabili, etc. Lo si fa in prospettiva funzionale, di solito a partire dai diagrammi di casi d'uso. Lo fa un soggetto diverso da chi ha progettato e implementato il sistema. Può essere accompagnato da coverage analysis.
- Esiste un livello intermedio, che non esiste invece nel caso della programmazione strutturata: *integration testing* su microarchitettura di classi. Una microarchitettura di classi è un insieme di classi molto accoppiate tra loro che presentano in qualche modo un'interfaccia comune verso il resto (per esempio le classi che implementano un pattern costituiscono di solito una mocroarchitettura). In questo ambito si distinguono i casi in cui si lavora sul class diagram che specifica il progetto o sul class diagram estratto dal codice

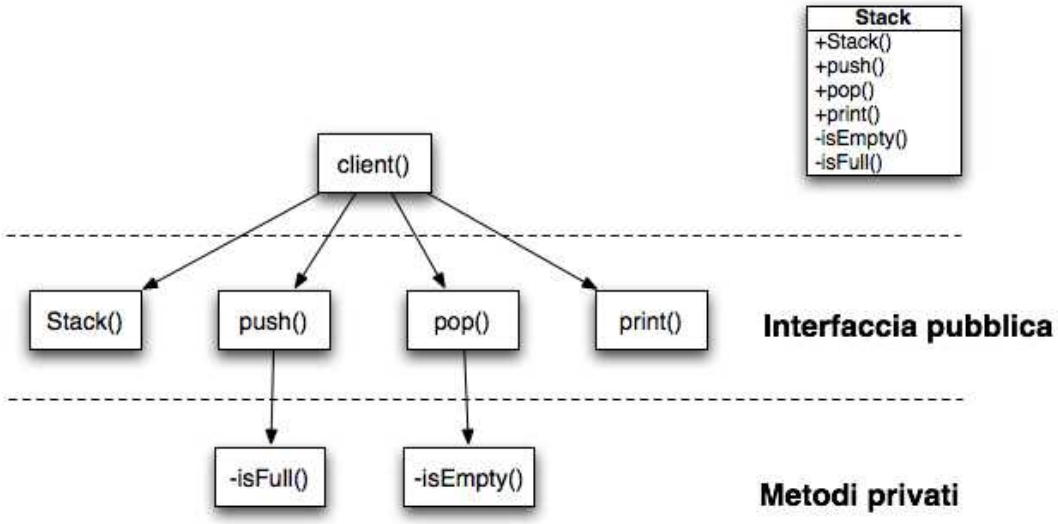
La pratica di sviluppo dipende anche dalla disponibilità della documentazione e dalla scelta testing manuale/computer aided.

5.2.1 Dal Control Flow Testing

Nell'ambito dello unit testing su singola classe, possiamo adattare le tecniche provenienti dal control flow testing andando a costruire un grafo, detto call graph, che rappresenta i metodi e le loro dipendenze d'uso.

Vengono distinti i metodi pubblici da quelli privati; sui metodi pubblici può avvenire un uso concorrente da parte di più client esterni, i metodi privati possono essere usati da più metodi pubblici; bisogna controllare cosa succede nei metodi chiamanti quando cambia lo stato dell'oggetto.

Esempio: consideriamo una classe che implementa uno stack.



Criteri di copertura possono essere:

- Invocare almeno una volta ogni metodo (funcion coverage). Si possono considerare solo metodi pubblici, anche metodi privari o protetti.
  Es: stack()->push()->isFull()->pop()->isEmpty()->print()
- Dove un metodo pubblico (privato) è usato da più clienti esterni (metodi) esercitare le diverse associazioni. Si possono testare i diversi ordini di interleaving nelle attivazioni.

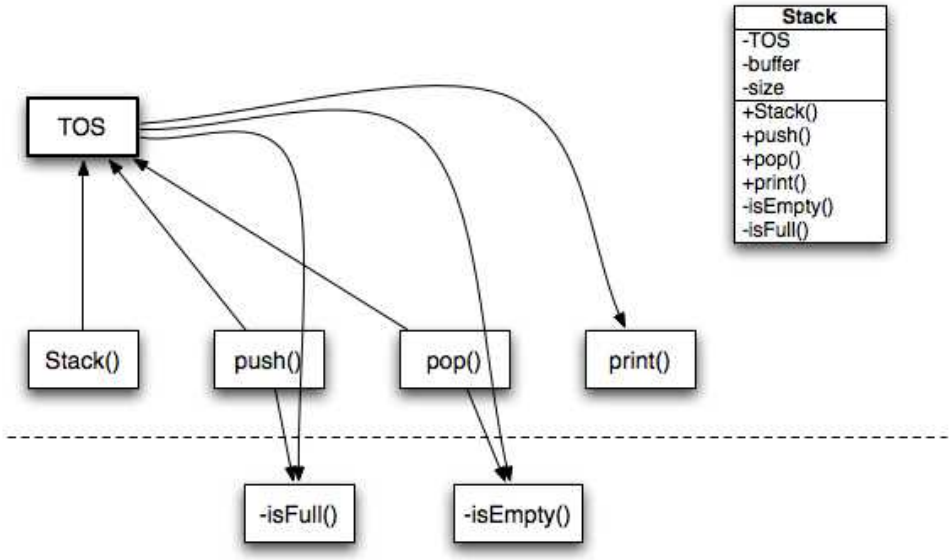
Nota: nella pratica di solito non si testano nemmeno tutti i metodi.

5.2.2 Data Flow Testing basato sugli attributi

Si può effettuare anche data flow testing usando come variabili gli attributi delle classi.
L'approccio consiste nell'andare a esercitare i def/use path rispetto agli attributi di un oggetto. Il call graph viene annotato con relazioni def-use sugli attributi che formano lo stato dell'oggetto. Non si possono però andare a testare tutti gli attributi di tutte le classi: si fa uno slicing sugli attributi più rilevanti rispetto all'interazione. Non si distingue tra Puse e Cuse perché vorrebbe dire andare a identificare il livello di statement.
Questa tecnica è applicabile:

- nello unit testing di una classe: risponde al problema di condivisione degli attributi; nella programmazione a oggetti aumenta l'accoppiamento rispetto alla programmazione strutturata, ma l'accoppiamento è localizzato
- nell'integration testing di microarchitetture: risponde al problema del mantenimento dello stato e della concorrenza nell'uso di uno stesso oggetto.

Riprendiamo il caso dello stack, come esempio di unit testing di una classe.



Gli attrubuti condivisi sono: buffer, TOS, size.
Effettuiamo un analisi dataflow sul TOS:

- Def: Stack(), push(), pop()
- Uses: print(), -isEmpty(), isFull()

Osservazioni:

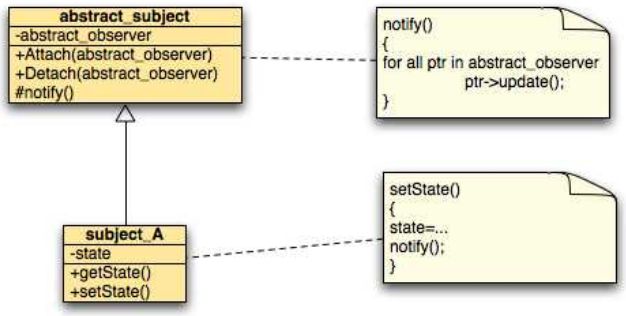
- non distinguiamo p-uses e c-uses
- il def in `push()` avviene dopo l'uso in `isFull()`

Una copertura *all uses* mi detta una serie di casi di test:

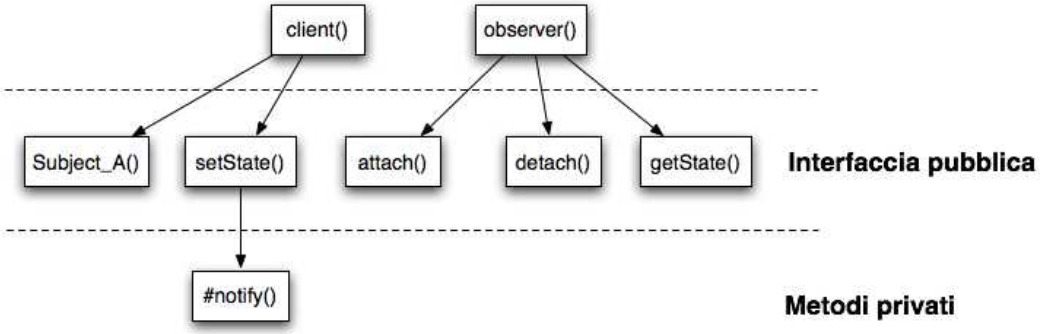
- Stack(), print()
- Stack(), push()[isFull()], print()
    - Prolungo fino a print() per osservare: l'uso in isFull() precede la def in push()
- Stack(), pop()[isEmpty()], print()
    - L'uso in isEmpty() precede la def in pop()
- …push(), push()[isFull()], print()
    - La def del secondo push() è successiva a isFull()
- …push(), pop()[isEmpty()],print()
- …pop(), push()[isFull()],print()
- …pop(),pop()[isEmpty], print()
    - stack() dopo push() o pop() non è feasible

N.B. Ogni caso di test deve iniziare con il costruttore.

Consideriamo adesso il caso di un observer come esempio di integration testing su microarchitettura.
Vogliamo testare il subject; esso subisce concorrenza perché viene chiamato dal client e dagli observer.

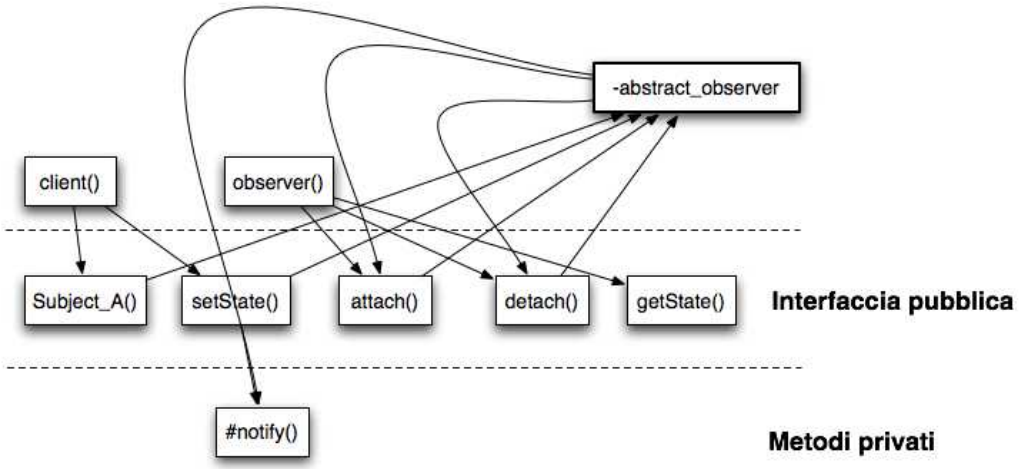


Costruiamo il call graph:



Il metodo notify() è privato, ma può essere esercitato attraverso setState().

La variabile critica di questo sistema è lo stato del subject (`state`); ma un buon criterio è quello di scegliere le variabili sulle quali è più probabile commettere errori: in questa ottica la variabile più critica è la lista degli observer (`abstract_observer`). Fare slicing sulla lista degli observer significa testare le operazioni di delega (interclass testing), e non la complessità interna della classe (intraclass testing, quello che sostanzialmente costituisce lo unit testing); i parametri critici sono quelli che hanno a che fare con l'integrazione tra le classi.
Annotazione dataflow del call graph:



Copertura all-def sulla lista degli observer (non dice molto):

- Subject(), setState()[Notify()]
- Subject(), attach(), setState()[Notify()]
    - setState() e notify() rendono osservabile l'esito del test
- Subject(), detach(), setState[Notify()]

Copertura all-uses (funziona):
- Subject(), setState()[notify()]
- Subject(), attach(), setState()[notify()]
- Subject(), detach(), setState()[notify()]
- … attach(), attach(), setState()[notify()]
- … detach(), attach(), setState()[notify()]

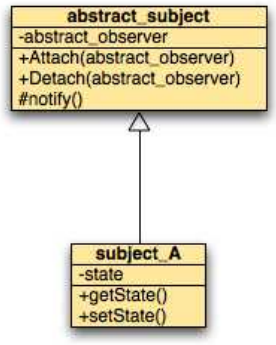
La copertura all-uses trova cose interessanti; per esempio prova a fare un notify() su una lista vuota o un detach() senza avere fatto prima un attach().

Si può apportare un ulteriore miglioramento: notiamo che la sequenza attach(), attach() e attach(), detach() dovrebbe essere distinta a seconda di come è risolta la condizione interna al secondo metodo (cioè se esso agisce sullo stesso oggetto su cui ha agito il primo). Per rendere la cosa osservabile sul call graph dovrei avere due funzioni sotto la guardia dell'if.

Il testing def-use da soluzione al problema dall'accoppiamento tra gli attributi (che comprende stato e concorrenza). Altra cosa è il discorso della topologia degli oggetti.

5.3 Il problema delle configurazioni della topologia

Quando si esegue un programma, ogni classe, che appare come singola entità in un diagramma UML delle classi, ha a run time più istanze, che possono avere dipendenze tra loro etc. Questo si limita fortemente se si programma usando l'ereditarietà.
Nel meccanismo dell'ereditarietà, tra la classe base e la classe derivata c'è una specifica ripartizione delle responsabilità, e se la programmazione è fatta bene, questa suddivisione risulta ortogonale.
Per esempio nell'observer:



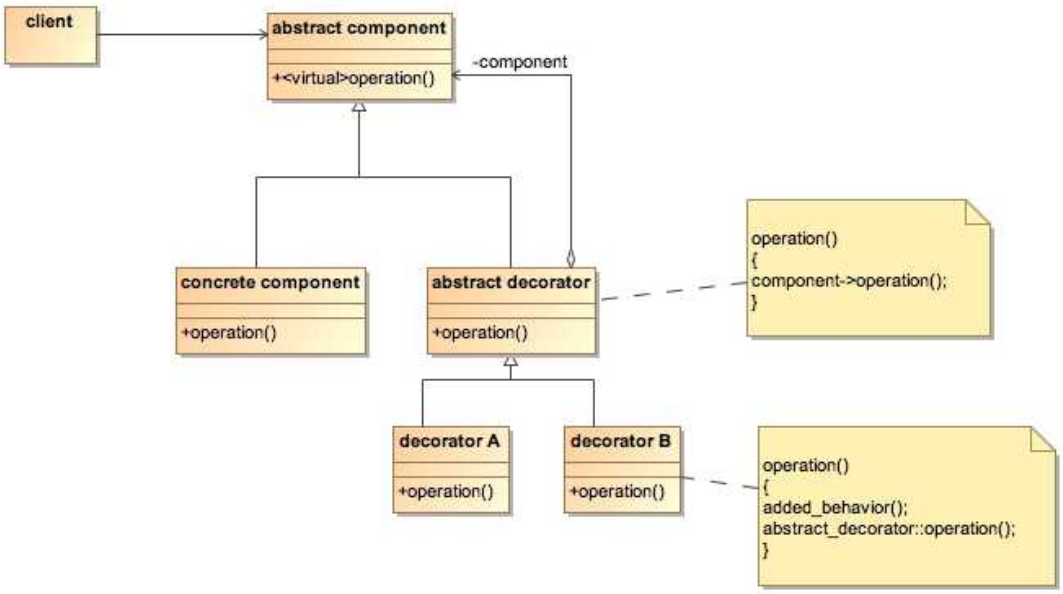
La classe derivata subject_A gestisce la business logic (es file system). La classe base abstract_subject ha invece la responsabilità di gestire il fatto di essere osservata (che non ha niente a che fare con la business logic).

Ma quando metto in vita la classe, c'è un solo oggetto che prende entrambe le responsabilità.

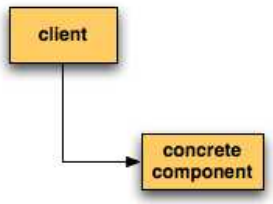
Quando invece uso il meccanismo della delega, ho 2 oggetti in vita e questo fa la differenze ai fini del testing; nel caso della delega la complessità a livello del testing è più alta di quella che vediamo nel codice.

Si può dunque avere la necessità di testare una funzione polimorfa per effetto di varietà nella configurazione delle deleghe o nella forma concreta che implementa le interfacce. Vediamo un esempio del secondo caso: il Pattern Decorator.

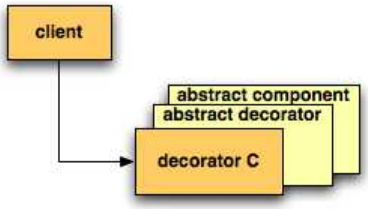
Il Pattern Decorator aggiunge dinamicamente responsabilità ad un oggetto in modo da estenderne le funzionalità (è un evoluzione del Pattern Proxy verso una struttura ripetitiva).



Il client invoca una funzionalità ad un oggetto abstract_component; nel caso più semplice quest'oggetto è un concrete_component:

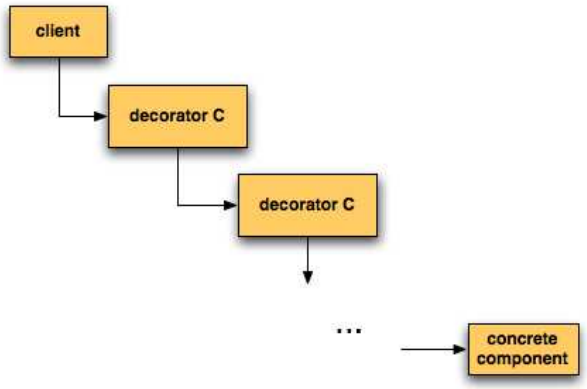


Nel caso più complicato è invece una classe derivata da abstract_decorator:



Nel caso in cui sia un abstract_decorator, avrà un puntatore ad un abstract_component, il quale a sua volta può essere implementato come un concrete_component, ma anche come una classe derivata da abstract_decorator (decorator_C), e così via. Nel caso

“peggiore” si può avere una lista, nella quale l’ultimo puntatore (se non è NULL), dovrà puntare ad un oggetto della classe concrete_component.



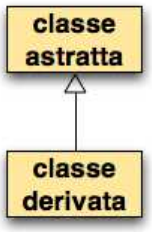
Questo pattern serve per esempio per decorare una funzione grafica, come una finestra, a cui posso eventualmente aggiungere una cornice, più una griglia, etc. Per esempio immaginiamo di voler comporre un certificato; la classe concrete_component si occuperà di realizzare la base del certificato, con scritto “Università di Firenze, Facoltà di Ingegneria”, i decorator mi aggiungono le diverse informazioni di cui ho bisogno, come per esempio i dati di uno studente o gli esami sostenuti; poi il certificato potrebbe essere in carta semplice o carta da bollo, etc. Ognuno dei decorator si prende una di queste responsabilità (per esempio uno si occuperà di stampare la lista degli esami.
Ogni oggetto decorator prima si occupa del suo proprio comportamento e poi fa una *upcall* sulla superclasse, la quale fa a sua volta un forwarding sull’abstract_component, chiamandone il metodo che implementa la funzione principale; questo metodo è virtuale, e allora verrà eseguito quello della classe derivata che ci sta sotto.
L’aspetto interessante di questo pattern è la possibilità di cambiare dinamicamente la configurazione. D’altro canto però il testing su qualcosa del genere diventa molto complicato.
Anche la gestione della memoria può essere un problema.

5.3.1 Class Dependency Graph

Per testare configurazioni polimorfiche ci possiamo avvalere dell’aiuto di un oggetto chiamato Class Dependency Graph. In questo grafo i vertici sono le classi e gli archi sono le dipendenze che ci sono tra di loro.
Queste dipendenze possono essere di diverso tipo:

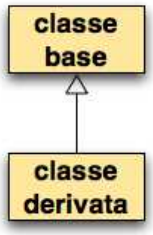
- dipendenze di uso: gli oggetti che istanziano la classe client delegano su oggetti che istanziano la classe server.

- dipendenze di implementazione: una classe derivata implementa i metodi astratti di una classe base.



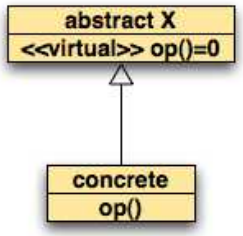
È la classe astratta che dipende dalla derivata.

- dipendenze di ereditarietà: la classe derivata eredità i metodi della classe base.



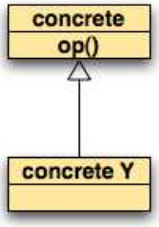
In questo caso è la classe derivata che dipende dalla classe da cui eredita.

dipendenza di implementazione:



La classe base dipende dalla classe derivata.

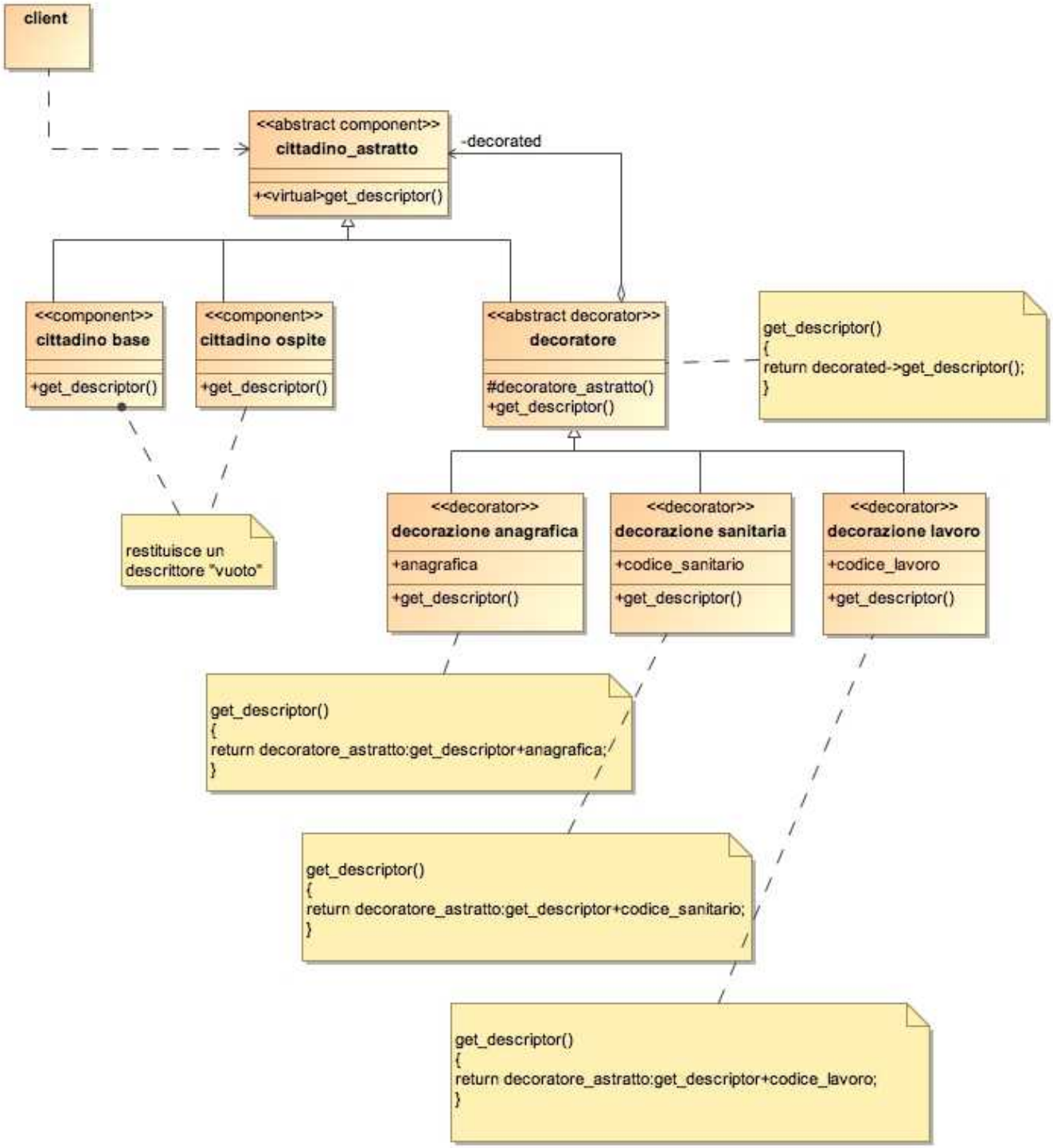
dipendenza di ereditarietà:



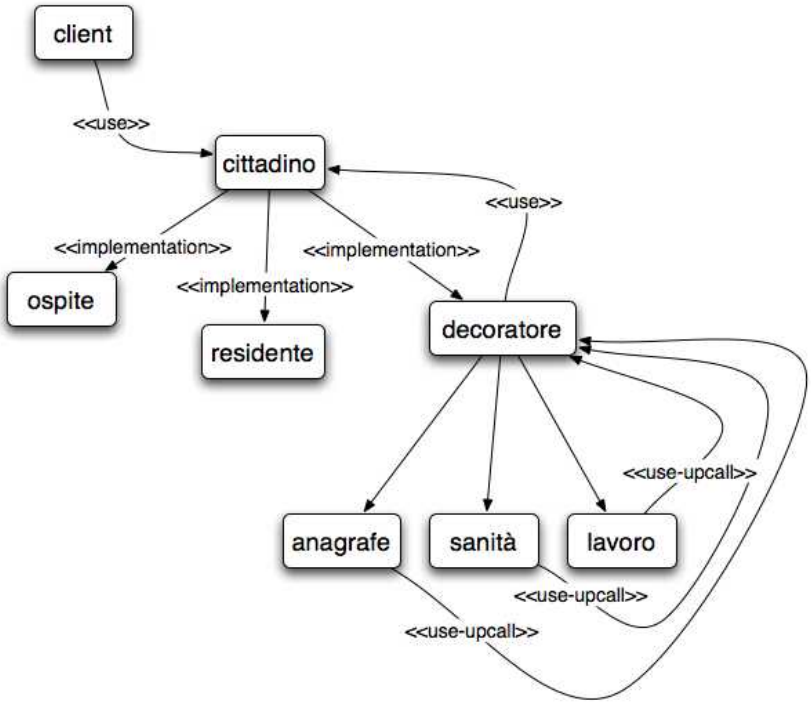
La classe derivata dipende dalla classe base.

- dipendenze di upcall esplicita: un metodo overridden. È un caso speciale perché non si applica dynamic binding.

Esempio: consideriamo un implementazione del pattern *decorator* per la creazione di certificati per i cittadini.

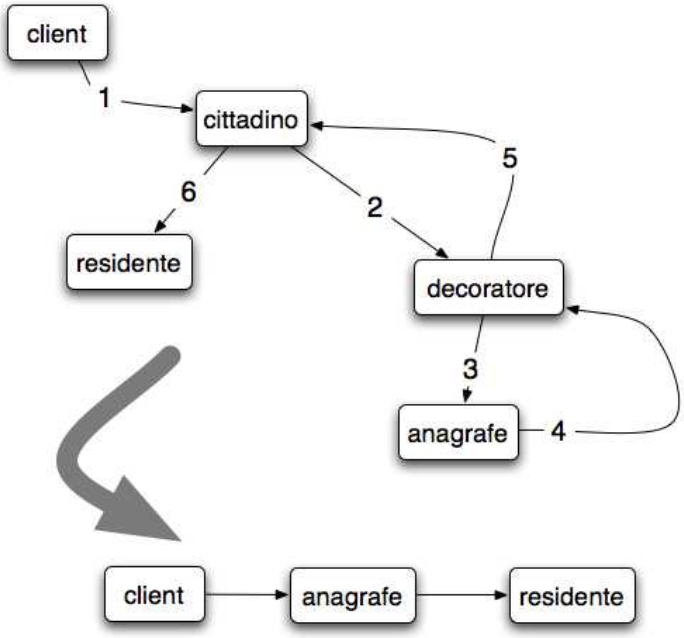

client
<<abstract component>>
cittadino_astratto
+<virtual>get_descriptor()
-decorated
<<component>>
cittadino base
+get_descriptor()
<<component>>
cittadino ospite
+get_descriptor()
<<abstract decorator>>
decoratore
#decoratore_astratto()
+get_descriptor()
get_descriptor()
{
return decorated->get_descriptor();
}
restituisce un
descrittore "vuoto"
<<decorator>>
decorazione anagrafica
+anagrafica
+get_descriptor()
<<decorator>>
decorazione sanitaria
+codice_sanitario
+get_descriptor()
<<decorator>>
decorazione lavoro
+codice_lavoro
+get_descriptor()
get_descriptor()
{
return decoratore_astratto:get_descriptor+anagrafica;
}
get_descriptor()
{
return decoratore_astratto:get_descriptor+codice_sanitario;
}
get_descriptor()
{
return decoratore_astratto:get_descriptor+codice_lavoro;
}

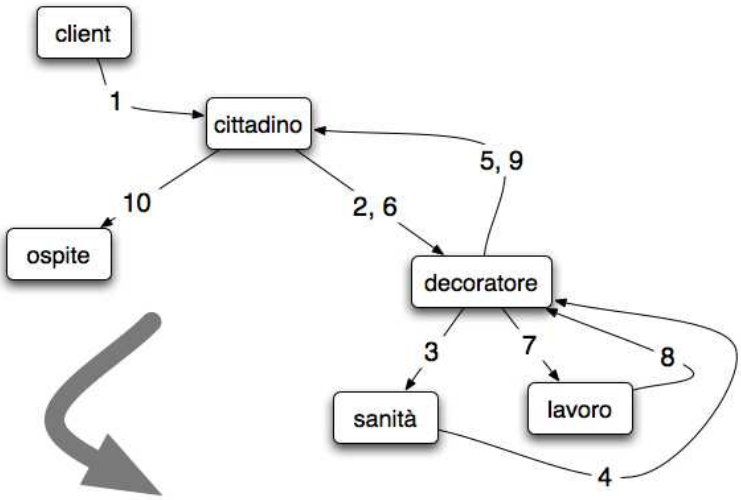
Costruiamo il Class Dependency Graph:



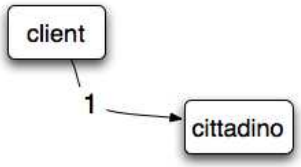
Questo grafico si può generare automaticamente dall'analisi del codice.

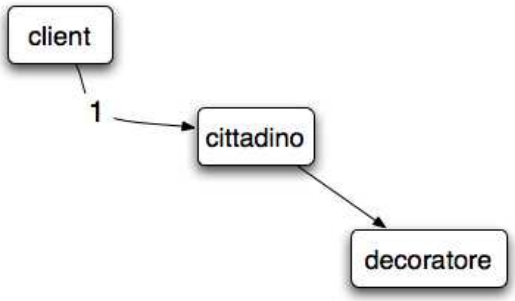
Posso definire diverse coperture sul Class Dependency Graph; ogni copertura sul grafo mi definisce una diversa configurazione degli oggetti.
Esempi di path:





Nota: non tutti i path sono validi (fattibili) per via dei vincoli definiti dalla politica di dynamic binding del linguaggio; un oggetto parzialmente astratto non può essere istanziato, il che implica che le gerarchie di ereditarietà devono essere discese fino a raggiungere una classe completamente concreta; e in questa prospettiva non sono istanziabili oggetti con costruttore protected o private.
Nel nostro esempio un path non può finire su un decoratore.





Realizzando una copertura *all nodes* garantisco di aver esercitato il sistema in configurazioni tali da aver generato almeno un'istanza di ogni classe e averla esercitata su un metodo.

All edges garantisce di aver esercitato tutte le relazioni di delega in tutti i versi.

Il criterio *all edges* conduce ad una suite di test diversa rispetto a quella individuata con *all nodes* soltanto se si hanno diversi archi uscenti da uno stesso nodo (per il polimorfismo), o diversi archi entranti in un solo nodo (che significa concorrenza).

Cosideriamo ora un esempio di problema tipico, la *fragile base class*. Questo problema si verifica quando un cambiamento in un dettaglio implementativo di una superclasse porta alla “rottura” di una sua sottoclasse, da cui il nome di classe base “fragile”.

Consideriamo una classe base con due metodi pubblici op1() e op2() e supponiamo che op2() usi op1(); il metodo op1() è dichiarato virtuale. Mettiamo ora che una sottoclasse di questa classe base ridefinisca il metodo op1(): la sottoclasse userà il metodo nuovo mentre la classe base potrà continuare a usare la vecchia versione del metodo. Il difetto di questo meccanismo è che, siccome op2() dipende da op1(), se op1() è dichiarato virtuale, quando il client chiama op2(), viene chiamata la versione di op1() della sottoclasse; questo metodo non è detto che faccia tutto quello che faceva il metodo della classe base; op1() e op2() sono accoppiati, quindi può darsi che op2() si rompa. L’*antipattern* è in questo caso avere dentro la classe dei metodi pubblici che dipendono l’uno dall’altro. Notiamo che se non dichiaro il metodo come virtuale il problema non si pone, ma se lo faccio tolgo la possibilità di riaprire la classe.
Vediamo un esempio: ho una classe Set che rappresenta un insieme di oggetti. La classe Set ha, tra li altri due metodi pubblici:

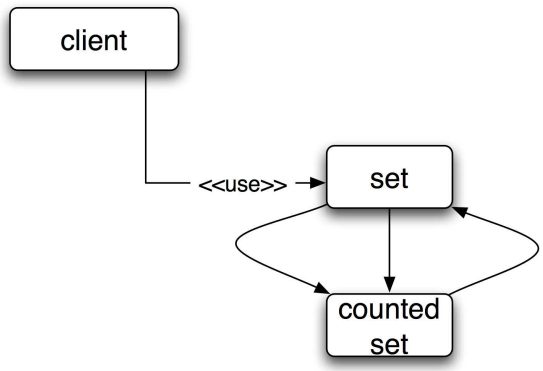
- addElement()
- include(): questo metodo include un altro set all’interno di questo

Una sottoclasse, CountedSet, che ha un campo (privato) card (la cardinalità dell’insieme) sovrascrive questi due metodi:

- addElement(): fa un upcall e poi incrementa cord
- include(): chiede al set che deve essere includo la sua cardinalità e incrementa card di conseguenza, poi chiama il metodo include della classe base; il problema appare in questo punto, perché il metodo include() della classe base chiama il metodo addElement(), viene chiamato il metodo della classe derivata e la cardinalità viene aumentata 2 volte.

Il problema deriva dal fatto che il metodo addElement() doveva essere privato.

Se disegno il grafo di questo sistema, troverò un self loop:



5.4 Control Flow Testing in prospettiva funzionale

Per poter usare i concetti del control flow testing in prospettiva funzionale[1], possiamo usare il diagramma dei casi d'uso, che specifica i requisiti funzionali del sistema.
Nello diagrammi di casi d'uso, per ogni caso d'uso del sistema, viene descritto il cosiddetto *flow of events*, ovvero il funzionamento del sistema, a livello d'interfaccia (si può fare a diversi livelli di astrazione) quando viene esercitata la funzionalità rappresentata da quel determinato caso d'uso; viene definito il flusso "normale", *basic flow*, e le eccezioni che possono venir generate, *alternate flows*.
A partire dai basic flows e gli alternate flows è possibile costruire un control flow graph. Uno scenario rappresenta una possibile esecuzione del caso d'uso; diversi scenari esercitano diversi flow. In pratica si realizzano diverse coperture del control flow graph (si possono usare i criteri *all nodes*, *all edges*).
Notiamo che il diagramma dei casi d'uso non rappresenta le dipendenze tra i dati (e quindi non si possono usare i criteri del data flow).
Devo poi identificare gli input per i vari flow (sensitization); quello che si rappresenta non è il valore dell'input, ma la classe: esistono delle classi di equivalenza. Per esempio "username e password sintatticamente valide" o, nel caso di inserimento di valori che devono stare dentro a un range predefinito, classi di equivalenza possono essere "sopra il massimo", "sotto il minimo", "uguale agli estremi", "all'interno dell'intervallo".
L'oracolo di solito è una persona che va a interpretare le specifiche.

[1] Se si vuole utilizzare una tecnica strutturale per testare in prospettiva funzionale una possibilità è anche rappresentare il sistema con un diagramma UML, generare automaticamente il codice corrispondente e usare la tecnica strutturale sul codice. Ma in questo modo non lavoriamo propriamente sulla specifica.

6. Appendice: Il caso Ariane5

ARIANE 5 - Flight 501 Failure
Report by the Inquiry Board
The Chairman of the Board Prof. J. L. LIONS
Paris, 19 July 1996

6.1 FOREWORD

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Engineers from the Ariane 5 project teams of CNES and Industry immediately started to investigate the failure. Over the following days, the Director General of ESA and the Chairman of CNES set up an independent Inquiry Board and nominated the following members :

- Prof. Jacques-Louis Lions (Chairman) Academie des Sciences (France)
- Dr. Lennart L beck (Vice-Chairman) Swedish Space Corporation (Sweden)
- Mr. Jean-Luc Fauquembergue Delegation Generale pour l'Armement (France)
- Mr. Gilles Kahn Institut National de Recherche en Informatique et en Automatique (INRIA), (France)
- Prof. Dr. Ing. Wolfgang Kubbat Technical University of Darmstadt (Germany)
- Dr. Ing. Stefan Levedag Daimler Benz Aerospace (Germany)
- Dr. Ing. Leonardo Mazzini Alenia Spazio (Italy)
- Mr. Didier Merle Thomson CSF (France)
- Dr. Colin O'Halloran Defence Evaluation and Research Agency (DERA), (U.K.)

The terms of reference assigned to the Board requested it

- to determine the causes of the launch failure,
- to investigate whether the qualification tests and acceptance tests were appropriate in relation to the problem encountered,
- to recommend corrective action to remove the causes of the anomaly and other possible weaknesses of the systems found to be at fault.

The Board started its work on 13 June 1996. It was assisted by a Technical Advisory Committee composed of :

- Dr Mauro Balduccini (BPD)
- Mr Yvan Choquer (Matra Marconi Space)
- Mr Remy Hergott (CNES)
- Mr Bernard Humbert (Aerospatiale)
- Mr Eric Lefort (ESA)

In accordance with its terms of reference, the Board concentrated its investigations on the causes of the failure, the systems supposed to be responsible, any failures of similar nature in similar systems, and events that could be linked to the accident. Consequently, the recommendations made by the Board are limited to the areas examined. The report contains the analysis of the failure, the Board's conclusions and its recommendations for corrective measures, most of which should be undertaken before the next flight of Ariane 5. There is in addition a report for restricted circulation in which the Board's findings are documented in greater technical detail. Although it consulted the telemetry data recorded during the flight, the Board has not undertaken an evaluation of those data. Nor has it made a complete review of the whole launcher and all its systems.

This report is the result of a collective effort by the Commission, assisted by the members of the Technical Advisory Committee.

We have all worked hard to present a very precise explanation of the reasons for the failure and to make a contribution towards the improvement of Ariane 5 software. This improvement is necessary to ensure the success of the programme.

The Board's findings are based on thorough and open presentations from the Ariane 5 project teams, and on documentation which has demonstrated the high quality of the Ariane 5 programme as regards engineering work in general and completeness and traceability of documents.

Chairman of the Board

6.2 THE FAILURE

6.2.1 GENERAL DESCRIPTION

On the basis of the documentation made available and the information presented to the Board, the following has been observed:

The weather at the launch site at Kourou on the morning of 4 June 1996 was acceptable for a launch that day, and presented no obstacle to the transfer of the launcher to the launch pad. In particular, there was no risk of lightning since the strength of the electric field measured at the launch site was negligible. The only uncertainty concerned fulfilment of the visibility criteria.

The countdown, which also comprises the filling of the core stage, went smoothly until H0-7 minutes when the launch was put on hold since the visibility criteria were not met at the opening of the launch window (08h35 local time). Visibility conditions improved as forecast and the launch was initiated at H0 = 09h 33mn 59s local time (=12h 33mn 59s UT). Ignition of the Vulcain engine and the two solid boosters was nominal, as was lift-off. The vehicle performed a nominal flight

until approximately H0 + 37 seconds. Shortly after that time, it suddenly veered off its flight path, broke up, and exploded. A preliminary investigation of flight data showed:

- nominal behaviour of the launcher up to H0 + 36 seconds;
- failure of the back-up Inertial Reference System followed immediately by failure of the active Inertial Reference System;
- swivelling into the extreme position of the nozzles of the two solid boosters and, slightly later, of the Vulcain engine, causing the launcher to veer abruptly;
- self-destruction of the launcher correctly triggered by rupture of the links between the solid boosters and the core stage.

The origin of the failure was thus rapidly narrowed down to the flight control system and more particularly to the Inertial Reference Systems, which obviously ceased to function almost simultaneously at around H0 + 36.7 seconds.

6.2.2 INFORMATION AVAILABLE

The information available on the launch includes:

- telemetry data received on the ground until H0 + 42 seconds
- trajectory data from radar stations
- optical observations (IR camera, films) - inspection of recovered material.

The whole of the telemetry data received in Kourou was transferred to CNES/Toulouse where the data were converted into parameter over time plots. CNES provided a copy of the data to Aerospatiale, which carried out analyses concentrating mainly on the data concerning the electrical system.

6.2.3 RECOVERY OF MATERIAL

The self-destruction of the launcher occurred near to the launch pad, at an altitude of approximately 4000 m. Therefore, all the launcher debris fell back onto the ground, scattered over an area of approximately 12 km2 east of the launch pad. Recovery of material proved difficult, however, since this area is nearly all mangrove swamp or savanna.

Nevertheless, it was possible to retrieve from the debris the two Inertial Reference Systems. Of particular interest was the one which had worked in active mode and stopped functioning last, and for which, therefore, certain information was not available in the telemetry data (provision for transmission to ground of this information was confined to whichever of the two units might fail first). The results of the examination of this unit were very helpful to the analysis of the failure sequence.

6.2.4 UNRELATED ANOMALIES OBSERVED

Post-flight analysis of telemetry has shown a number of anomalies which have been reported to the Board. They are mostly of minor significance and such as to be expected on a demonstration flight.

One anomaly which was brought to the particular attention of the Board was the gradual development, starting at Ho + 22 seconds, of variations in the hydraulic pressure of the actuators of the main engine nozzle. These variations had a frequency of approximately 10 Hz.

There are some preliminary explanations as to the cause of these variations, which are now under investigation.

After consideration, the Board has formed the opinion that this anomaly, while significant, has no bearing on the failure of Ariane 501.

6.3 ANALYSIS OF THE FAILURE

6.3.1 CHAIN OF TECHNICAL EVENTS

In general terms, the Flight Control System of the Ariane 5 is of a standard design. The attitude of the launcher and its movements in space are measured by an Inertial Reference System (SRI). It has its own internal computer, in which angles and velocities are calculated on the basis of information from a "strap-down" inertial platform, with laser gyros and accelerometers. The data from the SRI are transmitted through the databus to the On-Board Computer (OBC), which executes the flight program and controls the nozzles of the solid boosters and the Vulcain cryogenic engine, via servovalves and hydraulic actuators.

In order to improve reliability there is considerable redundancy at equipment level. There are two SRIs operating in parallel, with identical hardware and software. One SRI is active and one is in "hot" stand-by, and if the OBC detects that the active SRI has failed it immediately switches to the other one, provided that this unit is functioning properly. Likewise there are two OBCs, and a number of other units in the Flight Control System are also duplicated.

The design of the Ariane 5 SRI is practically the same as that of an SRI which is presently used on Ariane 4, particularly as regards the software.

Based on the extensive documentation and data on the Ariane 501 failure made available to the Board, the following chain of events, their inter-relations and causes have been established, starting with the destruction of the launcher and tracing back in time towards the primary cause.

- The launcher started to disintegrate at about H0 + 39 seconds because of high aerodynamic loads due to an angle of attack of more than 20 degrees that led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher.

- This angle of attack was caused by full nozzle deflections of the solid boosters and the Vulcain main engine.
- These nozzle deflections were commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2). Part of these data at that time did not contain proper flight data, but showed a diagnostic bit pattern of the computer of the SRI 2, which was interpreted as flight data.
- The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception.
- The OBC could not switch to the back-up SRI 1 because that unit had already ceased to function during the previous data cycle (72 milliseconds period) for the same reason as SRI 2.
- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.
- The error occurred in a part of the software that only performs alignment of the strap-down inertial platform. This software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose.
- The alignment function is operative for 50 seconds after starting of the Flight Mode of the SRIs which occurs at H0 - 3 seconds for Ariane 5. Consequently, when lift-off occurs, the function continues for approx. 40 seconds of flight. This time sequence is based on a requirement of Ariane 4 and is not required for Ariane 5.
- The Operand Error occurred due to an unexpected high value of an internal alignment function result called BH, Horizontal Bias, related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time.
- The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values.

The SRI internal events that led to the failure have been reproduced by simulation calculations. Furthermore, both SRIs were recovered during the Board's investigation and the failure context was precisely determined from memory readouts. In addition, the Board has examined the software code which was shown to be consistent with the failure scenario. The results of these examinations are documented in the Technical Report.

Therefore, it is established beyond reasonable doubt that the chain of events set out above reflects the technical causes of the failure of Ariane 501.

6.3.2 COMMENTS ON THE FAILURE SCENARIO

In the failure scenario, the primary technical causes are the Operand Error when converting the horizontal bias variable BH, and the lack of protection of this conversion which caused the SRI computer to stop.

It has been stated to the Board that not all the conversions were protected because a maximum workload target of 80% had been set for the SRI computer. To determine the vulnerability of unprotected code, an analysis was performed on every operation which could give rise to an exception, including an Operand Error. In particular, the conversion of floating point values to integers was analysed and operations involving seven variables were at risk of leading to an Operand Error. This led to protection being added to four of the variables, evidence of which appears in the Ada code. However, three of the variables were left unprotected. No reference to justification of this decision was found directly in the source code. Given the large amount of documentation associated with any industrial application, the assumption, although agreed, was essentially obscured, though not deliberately, from any external review.

The reason for the three remaining variables, including the one denoting horizontal bias, being unprotected was that further reasoning indicated that they were either physically limited or that there was a large margin of safety, a reasoning which in the case of the variable BH turned out to be faulty. It is important to note that the decision to protect certain variables but not others was taken jointly by project partners at several contractual levels.

There is no evidence that any trajectory data were used to analyse the behaviour of the unprotected variables, and it is even more important to note that it was jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specification.

Although the source of the Operand Error has been identified, this in itself did not cause the mission to fail. The specification of the exception-handling mechanism also contributed to the failure. In the event of any kind of exception, the system specification stated that: the failure should be indicated on the databus, the failure context should be stored in an EEPROM memory (which was recovered and read out for Ariane 501), and finally, the SRI processor should be shut down.

It was the decision to cease the processor operation which finally proved fatal. Restart is not feasible since attitude is too difficult to re-calculate after a processor shutdown; therefore the Inertial Reference System becomes useless. The reason behind this drastic action lies in the culture within the Ariane programme of only addressing random hardware failures. From this point of view exception - or error - handling mechanisms are designed for a random hardware failure which can quite rationally be handled by a backup system.

Although the failure was due to a systematic software design error, mechanisms can be introduced to mitigate this type of problem. For example the computers

within the SRIs could have continued to provide their best estimates of the required attitude information. There is reason for concern that a software exception should be allowed, or even required, to cause a processor to halt while handling mission-critical equipment. Indeed, the loss of a proper software function is hazardous because the same software runs in both SRI units. In the case of Ariane 501, this resulted in the switch-off of two still healthy critical units of equipment.

The original requirement accounting for the continued operation of the alignment software after lift-off was brought forward more than 10 years ago for the earlier models of Ariane, in order to cope with the rather unlikely event of a hold in the count-down e.g. between - 9 seconds, when flight mode starts in the SRI of Ariane 4, and - 5 seconds when certain events are initiated in the launcher which take several hours to reset. The period selected for this continued alignment operation, 50 seconds after the start of flight mode, was based on the time needed for the ground equipment to resume full control of the launcher in the event of a hold.

This special feature made it possible with the earlier versions of Ariane, to restart the count- down without waiting for normal alignment, which takes 45 minutes or more, so that a short launch window could still be used. In fact, this feature was used once, in 1989 on Flight 33.

The same requirement does not apply to Ariane 5, which has a different preparation sequence and it was maintained for commonality reasons, presumably based on the view that, unless proven necessary, it was not wise to make changes in software which worked well on Ariane 4.

Even in those cases where the requirement is found to be still valid, it is questionable for the alignment function to be operating after the launcher has lifted off. Alignment of mechanical and laser strap-down platforms involves complex mathematical filter functions to properly align the x-axis to the gravity axis and to find north direction from Earth rotation sensing. The assumption of preflight alignment is that the launcher is positioned at a known and fixed position. Therefore, the alignment function is totally disrupted when performed during flight, because the measured movements of the launcher are interpreted as sensor offsets and other coefficients characterising sensor behaviour.

Returning to the software error, the Board wishes to point out that software is an expression of a highly detailed design and does not fail in the same sense as a mechanical system. Furthermore software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess.

An underlying theme in the development of Ariane 5 is the bias towards the mitigation of random failure. The supplier of the SRI was only following the

specification given to it, which stipulated that in the event of any detected exception the processor was to be stopped. The exception which occurred was not due to random failure but a design error. The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. The Board has reason to believe that this view is also accepted in other areas of Ariane 5 software design. The Board is in favour of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.

This means that critical software - in the sense that failure of the software puts the mission at risk - must be identified at a very detailed level, that exceptional behaviour must be confined, and that a reasonable back-up policy must take software failures into account.

6.3.3 THE TESTING AND QUALIFICATION PROCEDURES

The Flight Control System qualification for Ariane 5 follows a standard procedure and is performed at the following levels :

- Equipment qualification
- Software qualification (On-Board Computer software)
- Stage integration
- System validation tests.

The logic applied is to check at each level what could not be achieved at the previous level, thus eventually providing complete test coverage of each sub-system and of the integrated system.

Testing at equipment level was in the case of the SRI conducted rigorously with regard to all environmental factors and in fact beyond what was expected for Ariane 5. However, no test was performed to verify that the SRI would behave correctly when being subjected to the count-down and flight time sequence and the trajectory of Ariane 5.

It should be noted that for reasons of physical law, it is not feasible to test the SRI as a "black box" in the flight environment, unless one makes a completely realistic flight test, but it is possible to do ground testing by injecting simulated accelerometric signals in accordance with predicted flight parameters, while also using a turntable to simulate launcher angular movements. Had such a test been performed by the supplier or as part of the acceptance test, the failure mechanism would have been exposed.

The main explanation for the absence of this test has already been mentioned above, i.e. the SRI specification (which is supposed to be a requirements

document for the SRI) does not contain the Ariane 5 trajectory data as a functional requirement.

The Board has also noted that the systems specification of the SRI does not indicate operational restrictions that emerge from the chosen implementation. Such a declaration of limitation, which should be mandatory for every mission-critical device, would have served to identify any non-compliance with the trajectory of Ariane 5.

The other principal opportunity to detect the failure mechanism beforehand was during the numerous tests and simulations carried out at the Functional Simulation Facility ISF, which is at the site of the Industrial Architect. The scope of the ISF testing is to qualify :

- the guidance, navigation and control performance in the whole flight envelope,
- the sensors redundancy operation, - the dedicated functions of the stages,
- the flight software (On-Board Computer) compliance with all equipment of the Flight Control Electrical System.

A large number of closed-loop simulations of the complete flight simulating ground segment operation, telemetry flow and launcher dynamics were run in order to verify :

- the nominal trajectory
- trajectories degraded with respect to internal launcher parameters
- trajectories degraded with respect to atmospheric parameters
- equipment failures and the subsequent failure isolation and recovery

In these tests many equipment items were physically present and exercised but not the two SRIs, which were simulated by specifically developed software modules. Some open-loop tests, to verify compliance of the On-Board Computer and the SRI, were performed with the actual SRI. It is understood that these were just electrical integration tests and "low-level " (bus communication) compliance tests.

It is not mandatory, even if preferable, that all the parts of the subsystem are present in all the tests at a given level. Sometimes this is not physically possible or it is not possible to exercise them completely or in a representative way. In these cases it is logical to replace them with simulators but only after a careful check that the previous test levels have covered the scope completely.

This procedure is especially important for the final system test before the system is operationally used (the tests performed on the 501 launcher itself are not

addressed here since they are not specific to the Flight Control Electrical System qualification).

In order to understand the explanations given for the decision not to have the SRIs in the closed-loop simulation, it is necessary to describe the test configurations that might have been used.

Because it is not possible to simulate the large linear accelerations of the launcher in all three axes on a test bench (as discussed above), there are two ways to put the SRI in the loop:

A) To put it on a three-axis dynamic table (to stimulate the Ring Laser Gyros) and to substitute the analog output of the accelerometers (which can not be stimulated mechanically) by simulation via a dedicated test input connector and an electronic board designed for this purpose. This is similar to the method mentioned in connection with possible testing at equipment level.

B) To substitute both, the analog output of the accelerometers and the Ring Laser Gyros via a dedicated test input connector with signals produced by simulation.

The first approach is likely to provide an accurate simulation (within the limits of the three-axis dynamic table bandwidth) and is quite expensive; the second is cheaper and its performance depends essentially on the accuracy of the simulation. In both cases a large part of the electronics and the complete software are tested in the real operating environment.

When the project test philosophy was defined, the importance of having the SRIs in the loop was recognized and a decision was taken to select method B above. At a later stage of the programme (in 1992), this decision was changed. It was decided not to have the actual SRIs in the loop for the following reasons :

- The SRIs should be considered to be fully qualified at equipment level
- The precision of the navigation software in the On-Board Computer depends critically on the precision of the SRI measurements. In the ISF, this precision could not be achieved by the electronics creating the test signals.
- The simulation of failure modes is not possible with real equipment, but only with a model.
- The base period of the SRI is 1 millisecond whilst that of the simulation at the ISF is 6 milliseconds. This adds to the complexity of the interfacing electronics and may further reduce the precision of the simulation.

The opinion of the Board is that these arguments were technically valid, but since the purpose of a system simulation test is not only to verify the interfaces but also to verify the system as a whole for the particular application, there was a definite

risk in assuming that critical equipment such as the SRI had been validated by qualification on its own, or by previous use on Ariane 4.

While high accuracy of a simulation is desirable, in the ISF system tests it is clearly better to compromise on accuracy but achieve all other objectives, amongst them to prove the proper system integration of equipment such as the SRI. The precision of the guidance system can be effectively demonstrated by analysis and computer simulation.

Under this heading it should be noted finally that the overriding means of preventing failures are the reviews which are an integral part of the design and qualification process, and which are carried out at all levels and involve all major partners in the project (as well as external experts). In a programme of this size, literally thousands of problems and potential failures are successfully handled in the review process and it is obviously not easy to detect software design errors of the type which were the primary technical cause of the 501 failure. Nevertheless, it is evident that the limitations of the SRI software were not fully analysed in the reviews, and it was not realised that the test coverage was inadequate to expose such limitations. Nor were the possible implications of allowing the alignment software to operate during flight realised. In these respects, the review process was a contributory factor in the failure.

6.3.4 POSSIBLE OTHER WEAKNESSES OF SYSTEMS INVOLVED

In accordance with its termes of reference, the Board has examined possible other weaknesses, primarily in the Flight Control System. No weaknesses were found which were related to the failure, but in spite of the short time available, the Board has conducted an extensive review of the Flight Control System based on experience gained during the failure analysis.

The review has covered the following areas :

- The design of the electrical system,
- Embedded on-board software in subsystems other than the Inertial Reference System,
- The On-Board Computer and the flight program software.

In addition, the Board has made an analysis of methods applied in the development programme, in particular as regards software development methodology.

The results of these efforts have been documented in the Technical Report and it is the hope of the Board that they will contribute to further improvement of the Ariane 5 Flight Control System and its software.

6.4 CONCLUSIONS

6.4.1 FINDINGS

The Board reached the following findings:

a) During the launch preparation campaign and the count-down no events occurred which were related to the failure.

b) The meteorological conditions at the time of the launch were acceptable and did not play any part in the failure. No other external factors have been found to be of relevance.

c) Engine ignition and lift-off were essentially nominal and the environmental effects (noise and vibration) on the launcher and the payload were not found to be relevant to the failure. Propulsion performance was within specification.

d) 22 seconds after H0 (command for main cryogenic engine ignition), variations of 10 Hz frequency started to appear in the hydraulic pressure of the actuators which control the nozzle of the main engine. This phenomenon is significant and has not yet been fully explained, but after consideration it has not been found relevant to the failure.

e) At 36.7 seconds after H0 (approx. 30 seconds after lift-off) the computer within the back-up inertial reference system, which was working on stand-by for guidance and attitude control, became inoperative. This was caused by an internal variable related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer.

f) Approx. 0.05 seconds later the active inertial reference system, identical to the back-up system in hardware and software, failed for the same reason. Since the back-up inertial system was already inoperative, correct guidance and attitude information could no longer be obtained and loss of the mission was inevitable.

g) As a result of its failure, the active inertial reference system transmitted essentially diagnostic information to the launcher's main computer, where it was interpreted as flight data and used for flight control calculations.

h) On the basis of those calculations the main computer commanded the booster nozzles, and somewhat later the main engine nozzle also, to make a large correction for an attitude deviation that had not occurred.

i) A rapid change of attitude occurred which caused the launcher to disintegrate at 39 seconds after H0 due to aerodynamic forces.

j) Destruction was automatically initiated upon disintegration, as designed, at an altitude of 4 km and a distance of 1 km from the launch pad.

k) The debris was spread over an area of 5 x 2.5 km2. Amongst the equipment recovered were the two inertial reference systems. They have been used for analysis.

l) The post-flight analysis of telemetry data has listed a number of additional anomalies which are being investigated but are not considered significant to the failure.

m) The inertial reference system of Ariane 5 is essentially common to a system which is presently flying on Ariane 4. The part of the software which caused the interruption in the inertial system computers is used before launch to align the inertial reference system and, in Ariane 4, also to enable a rapid realignment of the system in case of a late hold in the countdown. This realignment function, which does not serve any purpose on Ariane 5, was nevertheless retained for commonality reasons and allowed, as in Ariane 4, to operate for approx. 40 seconds after lift-off.

n) During design of the software of the inertial reference system used for Ariane 4 and Ariane 5, a decision was taken that it was not necessary to protect the inertial system computer from being made inoperative by an excessive value of the variable related to the horizontal velocity, a protection which was provided for several other variables of the alignment software. When taking this design decision, it was not analysed or fully understood which values this particular variable might assume when the alignment software was allowed to operate after lift-off.

o) In Ariane 4 flights using the same type of inertial reference system there has been no such failure because the trajectory during the first 40 seconds of flight is such that the particular variable related to horizontal velocity cannot reach, with an adequate operational margin, a value beyond the limit present in the software.

p) Ariane 5 has a high initial acceleration and a trajectory which leads to a build-up of horizontal velocity which is five times more rapid than for Ariane 4. The higher horizontal velocity of Ariane 5 generated, within the 40-second timeframe, the excessive value which caused the inertial system computers to cease operation.

q) The purpose of the review process, which involves all major partners in the Ariane 5 programme, is to validate design decisions and to obtain flight qualification. In this process, the limitations of the alignment software were not fully analysed and the possible implications of allowing it to continue to function during flight were not realised.

r) The specification of the inertial reference system and the tests performed at equipment level did not specifically include the Ariane 5 trajectory data. Consequently the realignment function was not tested under simulated Ariane 5 flight conditions, and the design error was not discovered.

s) It would have been technically feasible to include almost the entire inertial reference system in the overall system simulations which were performed. For a number of reasons it was decided to use the simulated output of the inertial reference system, not the system itself or its detailed simulation. Had the system been included, the failure could have been detected.

t) Post-flight simulations have been carried out on a computer with software of the inertial reference system and with a simulated environment, including the actual trajectory data from the Ariane 501 flight. These simulations have faithfully reproduced the chain of events leading to the failure of the inertial reference systems.

6.4.2 CAUSE OF THE FAILURE

The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift- off). This loss of information was due to specification and design errors in the software of the inertial reference system.

The extensive reviews and tests carried out during the Ariane 5 Development Programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure.

6.5 RECOMMENDATIONS

On the basis of its analyses and conclusions, the Board makes the following recommendations.

R1 Switch off the alignment function of the inertial reference system immediately after lift-off. More generally, no software function should run during flight unless it is needed.

R2 Prepare a test facility including as much real equipment as technically feasible, inject realistic input data, and perform complete, closed-loop, system testing. Complete simulations must take place before any mission. A high test coverage has to be obtained.

R3 Do not allow any sensor, such as the inertial reference system, to stop sending best effort data.

R4 Organize, for each item of equipment incorporating software, a specific software qualification review. The Industrial Architect shall take part in these reviews and report on complete system testing performed with the equipment. All restrictions on use of the equipment shall be made explicit for the Review Board. Make all critical software a Configuration Controlled Item (CCI).

R5 Review all flight software (including embedded software), and in particular :

- Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment. Check these assumptions against the restrictions on use of the equipment.
- Verify the range of values taken by any internal or communication variables in the software.
- Solutions to potential problems in the on-board computer software, paying particular attention to on-board computer switch over, shall be proposed by the project team and reviewed by a group of external experts, who shall report to the on-board computer Qualification Board.

R6 Wherever technically feasible, consider confining exceptions to tasks and devise backup capabilities.

R7 Provide more data to the telemetry upon failure of any component, so that recovering equipment will be less essential.

R8 Reconsider the definition of critical components, taking failures of software origin into account (particularly single point failures).

R9 Include external (to the project) participants when reviewing specifications, code and justification documents. Make sure that these reviews consider the substance of arguments, rather than check that verifications have been made.

R10 Include trajectory data in specifications and test requirements.

R11 Review the test coverage of existing equipment and extend it where it is deemed necessary.

R12 Give the justification documents the same attention as code. Improve the technique for keeping code and its justifications consistent.

R13 Set up a team that will prepare the procedure for qualifying software, propose stringent rules for confirming such qualification, and ascertain that specification, verification and testing of software are of a consistently high quality in the Ariane 5 programme. Including external RAMS experts is to be considered.

R14 A more transparent organisation of the cooperation among the partners in the Ariane 5 programme must be considered. Close engineering cooperation, with clear cut authority and responsibility, is needed to achieve system coherence, with simple and clear interfaces between partners.

- END –

7. Credits

I contenuti di questo testo riportano l'esperienza di insegnamento nel corso di Metodi di Verifica e Testing nel corso di Laurea Magistrale in Ingegneria Informatica tenuto presso la Facoltà di Ingegneria di Firenze a partire dall'anno accademico 2004/05.

Hanno collaborato alla sistematizzazione dei contenuti: Evelina Agostini, Mirco Soderi, Marco Lastrucci, … Bargiotti,