

Politecnico di Milano

V FACOLTA' DI INGEGNERIA
CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA



Testing Object Oriented Software

Andrea Magni



Obiettivi della presentazione

- ▶ Affrontare gli aspetti critici dell'attività di **testing** nel caso in cui il software venga sviluppato con una metodologia Object Oriented.
- ▶ Mostrare alcuni **approcci operativi** per il testing: come generare alcuni casi di test automaticamente, come procedere nella scrittura dei test partendo dal codice.
- ▶ Esporre alcune considerazioni sul testing OO e sulle sue diverse possibili collocazioni con attenzione alle implicazioni metodologiche.



Un mondo ad oggetti ma...

- ▶ La metodologia di sviluppo Object Oriented nel corso degli anni '90 ha acquisito una grande popolarità. A ruota è seguito un forte sviluppo delle metodologie di specifica e design.
- ▶ Vantaggi:
 - ♦ Architetture modulari, manutenzione meno problematica, facilità di riuso del codice, buona documentazione...
 - ♦ Cosa non ha tratto particolari benefici? Il problema della **correttezza!** (Software error-free non esiste: ci sarà sempre il **bisogno** di fare testing!)



Object Oriented Testing

- ▶ Sotto alcuni punti di vista, i sistemi OO non sono per niente diversi dai sistemi tradizionali. (i.e. system level, procedure level)
- ▶ La complessità è cresciuta nei livelli intermedi del testing e questo aumento è dovuto alle peculiarità dei linguaggi OO.

Segue una veloce panoramica...



Object Oriented Software (I)

► **Gli oggetti sono dotati di stato**

- ♦ il comportamento di un metodo potrebbe non dipendere esclusivamente dai suoi parametri espliciti
- ♦ Non si può più fare l'assunzione che l'output di un metodo dipenda solo dal suo input (a differenza che nel testing di software “tradizionale”)

► **Information hiding e incapsulazione**

- ♦ La presenza della distinzione parte pubblica / parte privata (valida per attributi e metodi) porta a delle difficoltà nella realizzazione degli oracoli e nell'individuazione dei casi da testare.
L'incapsulazione può rendere difficile capire cos'è che va testato e/o renderlo inaccessibile.



Object Oriented Software (2)

► Ereditarietà

- ♦ Le classi figlie si trovano ad ereditare variabili e metodi che possono essere anche ridefiniti. La definizione di una classe può essere sparsa su una gerarchia anche molto “alta”.
- ♦ Problemi su come ottimizzare il test (cosa testare, come trovare casi di test che si possano riutilizzare sulla gerarchia) e necessità di un testing incrementale.

► Binding dinamico e polimorfismo

- ♦ Causano una crescita esponenziale del numero di possibili situazioni da testare (parametri, valori di ritorno). E' necessario cercare di coprire almeno le combinazioni significative (possono essere molte).



Object Oriented Software (3)

► Astrazione

- ♦ Esistono situazioni in cui c'è la necessità di testare una classe astratta anche se non sono disponibili tutte le sue possibili sottoclassi. Servono tecniche per selezionare un insieme adeguato di istanze per il testing.

► Eccezioni

- ♦ Sicuramente un elemento centrale della programmazione OO. Complicano il flusso di esecuzione e si propagano attraversando più gestori posizionati in punti diversi del codice.



Object Oriented Software (4)

► Concorrenza

- ♦ Più o meno gli stessi problemi degli approcci “tradizionali”: deadlocks, race conditions, comportamento non deterministico. In genere si ha una certa difficoltà nel riprodurre correttamente i test case.

► Altre aree problematiche

- ♦ Generics
- ♦ Conversions
- ♦ Shadow invocations
- ♦ (Testing levels)



Sistemi automatici di generazione dei test (I)

- ▶ Un'importante caratteristica del software OO è la dipendenza del comportamento dei metodi dallo **stato** di un oggetto.
- ▶ Lo stato di un oggetto in un certo istante è dovuto ad una serie di **chiamate ai metodi** che l'oggetto ha sperimentato fino a quel momento.
- ▶ Comportamenti scorretti possono essere rivelati dall'esecuzione di specifiche sequenze di messaggi! Uno dei compiti da assolvere è quello di individuare queste **sequenze “rivelatrici”**.



Sistemi automatici di generazione dei test (2)

- ▶ Sulla singola classe (isolata):
 1. **Data-flow analysis** applicata ai metodi contenuti nel componente da testare
 2. **Symbolic execution** per derivare una specifica formale per ogni metodo
 3. **Automated deduction** per derivare sequenze di invocazioni di metodi sfruttando le informazioni ottenute ai passi 1 e 2.
 4. Si **eseguono** tutte le sequenze individuate
- ▶ Integrazione con altre classi (ne parliamo più tardi)

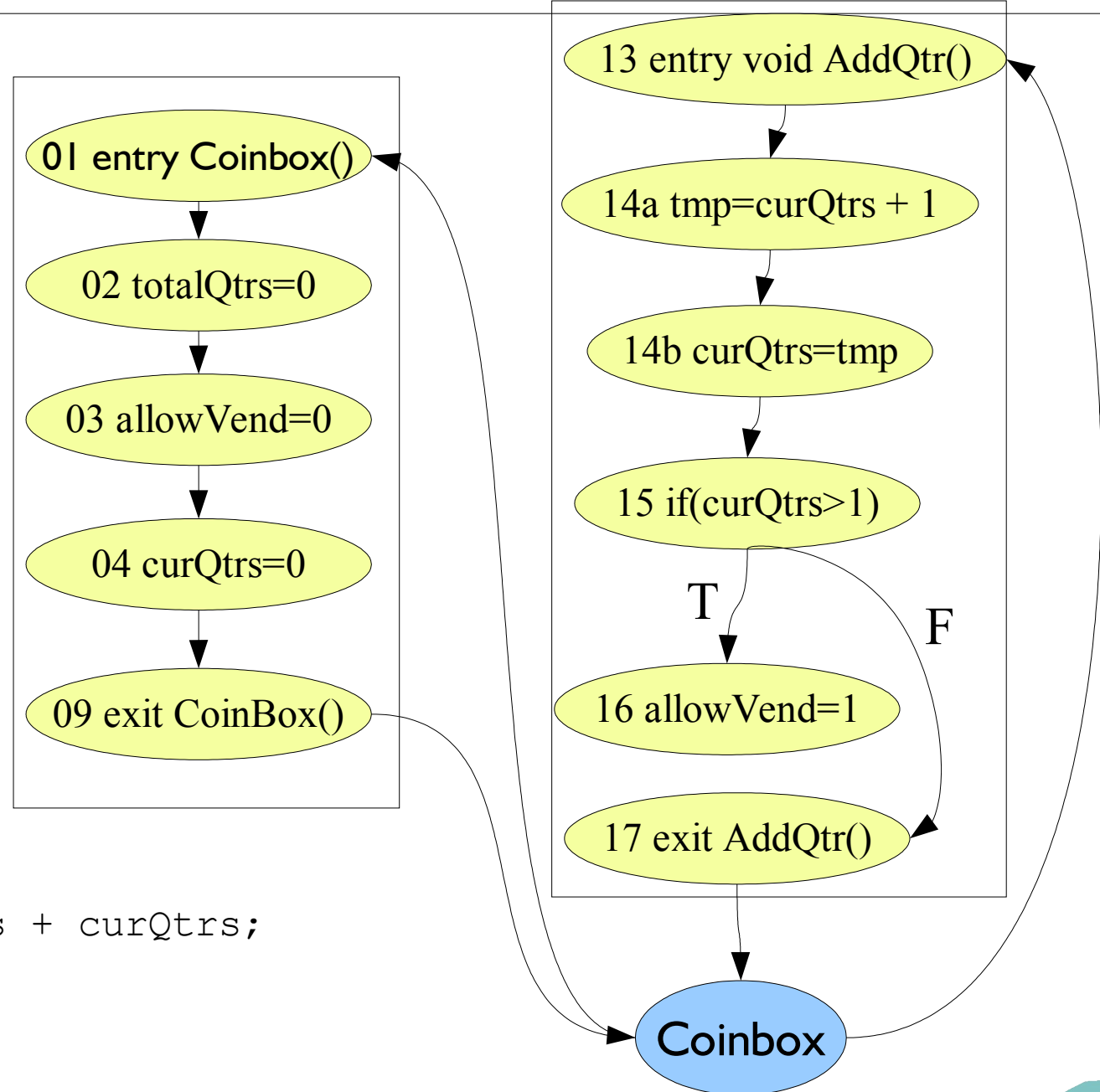


I) Vending Machine Example: Data Flow (I)

```

class CoinBox {
    unsigned totalQtrs;
    unsigned curQtrs;
    unsigned allowVend;
public:
    CoinBox () {
        totalQtrs = 0;
        allowVend = 0;
        curQtrs = 0;
    }
    void returnQtrs () {
        curQtrs = 0;
    }
    void addQtr () {
        curQtrs = curQtrs + 1;
        if (curQtrs > 1)
            allowVend = 1;
    }
    void vend () {
        if (allowedVend) {
            totalQtrs = totalQtrs + curQtrs;
            curQtrs = 0;
            allowVend = 0;
        }
    }
}

```





I) DataFlow (2) : DU pairs

#	variable	md (node#)	mu (node#)	notes
01	curQtrs	CoinBox (4)	addQtr (14a)	
02	curQtrs	CoinBox (4)	addQtr (15)	inf.
03	allowVend	CoinBox (3)	vend (19)	
04	totalQtrs	CoinBox (2)	vend (20a)	
05	curQtrs	returnQtrs (11)	addQtr (14a)	
06	curQtrs	returnQtrs (11)	addQtr (15)	inf.
07	curQtrs	returnQtrs (11)	vend (20a)	
08	curQtrs	addQtr (14b)	addQtr (14a)	
09	curQtrs	addQtr (14b)	addQtr (15)	
10	curQtrs	addQtr (14b)	vend (20a)	
11	allowVend	addQtr (16)	vend (19)	
12	totalQtrs	vend (20b)	vend (20a)	
13	curQtrs	vend (21)	addQtr (14a)	
14	curQtrs	vend (21)	addQtr (15)	inf.
15	curQtrs	vend (21)	vend (20a)	inf.
16	allowVend	vend (22)	vend (19)	
17	curQtrs	CoinBox (4)	vend (20a)	



2) Symbolic Execution

CoinBox

(true)
 $\text{def} = \{\text{totalQtrs}, \text{curQtrs}, \text{allowVend}\} \Rightarrow$
 $\text{totalQtrs}' = 0$
 $\text{curQtrs}' = 0$
 $\text{allowVend}' = 0$

addQtr

$(\text{curQtrs} > 0)$
 $\text{def} = \{\text{curQtrs}, \text{allowVend}\} \Rightarrow$
 $\text{curQtrs}' = \text{curQtrs} + 1$
 $\text{allowVend}' = 1$

$(\text{curQtrs} == 0)$
 $\text{def} = \{\text{curQtrs}\} \Rightarrow$
 $\text{curQtrs}' = 1$

CoinBox		
(true) $\text{def} = \{\text{totalQtrs}, \text{curQtrs}, \text{allowVend}\}$	\Rightarrow	$\text{totalQtrs}' = 0$ $\text{curQtrs}' = 0$ $\text{allowVend}' = 0$
addQtr		
$(\text{curQtrs} > 0)$ $\text{def} = \{\text{curQtrs}, \text{allowVend}\}$	\Rightarrow	$\text{curQtrs}' = \text{curQtrs} + 1$ $\text{allowVend}' = 1$
$(\text{curQtrs} == 0)$ $\text{def} = \{\text{curQtrs}\}$	\Rightarrow	$\text{curQtrs}' = 1$
vend		
$(\text{allowVend} \neq 0)$ $\text{def} = \{\text{totalQtrs}, \text{curQtrs}, \text{allowVend}\}$	\Rightarrow	$\text{totalQtrs}' = \text{totalQtrs} + \text{curQtrs}$ $\text{curQtrs}' = 0$ $\text{allowVend}' = 0$
$(\text{allowVend} == 0)$ $\text{def} = \{\}$	\Rightarrow	$\text{allowVend}' = 0$
returnQtrs		
(true) $\text{def} = \{\text{curQtrs}\}$	\Rightarrow	$\text{curQtrs}' = 0$

vend

$(\text{allowVend} <> 0)$
 $\text{def} = \{\text{totalQtrs}, \text{curQtrs}, \text{allowVend}\} \Rightarrow$
 $\text{totalQtrs}' = \text{totalQtrs} + \text{curQtrs}$
 $\text{curQtrs}' = 0$
 $\text{allowVend}' = 0$

$(\text{allowVend} == 0)$
 $\text{def} = \{\}$ \Rightarrow $\text{allowVend}' = 0$



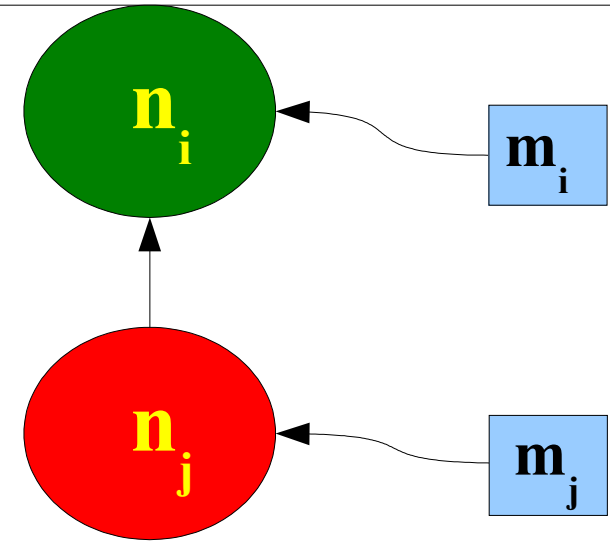
3) Sequence generation for UT (I)

- ▶ Si usano le specifiche ottenute con l'esecuzione simbolica per derivare sequenze di metodi cercando di coprire le coppie DU identificate al primo passo.
- ▶ Si cercano sequenze possibili esaminando le precondizioni necessarie per eseguire m_u . Si costruisce un **albero di possibili chiamate**: se il nodo i è associato al metodo m_i , i suoi figli sono metodi le cui postcondizioni non sono in contraddizione con le precondizioni di m_i .



3) Sequence generation for UT (2)

- ▶ Le postcondizioni di m_j non devono contraddire la path condition di n_i .



- ▶ La path condition di n_j è costruita dalla path condition di n_i , rimuovendo le clausole soddisfatte dalle postcondizioni di m_j e aggiungendo le precondizioni di m_j .



3) Sequence generation for UT (3)

- ▶ Condizioni di arresto (nella creazione dell'albero):
 - a) non ci sono nodi da aggiungere tali da realizzare una copertura possibile per la coppia DU in esame
 - b) raggiungo una certa profondità prefissata e mi fermo (+ notifica all'utente del framework)
 - c) trovo una copertura **possibile** per una coppia DU (3 condizioni):
 1. Uno dei nodi foglia è un costruttore
 2. Il percorso dalla radice alle foglie contiene un nodo n_d che corrisponde all'esecuzione dello statement s_d dal metodo m_d
 3. Il percorso è tale per cui la variabile v non è ri-definita fra n_d e il nodo radice



Integration Testing (I)

► Strategia:

- ♦ Trovare un ordine di integrazione che minimizzi le necessità di scaffolding (analisi delle dipendenze)
- ♦ Procedere alla integrazione accoppiata dei componenti
- ♦ Tradeoff: **#test cases** vs **completezza del test set**
 - Estremo 1: data una sequenza s di metodi del componente A che coinvolgano il componente B , ripetere la sequenza s per ogni stato di B (dovrò portare B in tutti gli stati possibili, usando i suoi metodi).
 - Estremo 2: provo ogni sequenza s di metodi di A con una sequenza di metodi di B .



Integration Testing (2)

- ▶ 1° approccio:
 - ◆ Considero i due componenti come se fossero raggruppati ma l'interfaccia è l'unione delle due.

- ▶ 2° approccio:
 - ◆ Considerare i due componenti come se fossero raggruppati, l'interfaccia a disposizione è quella di uno dei due (es. quella di A)
 - ◆ Semplicemente cerco delle sequenze di invocazioni per A.

- ▶ Il sistema è giudicato corretto se
 - Invocazioni indirette avvengono come previsto durante lo unit testing
 - I risultati delle sequenze di invocazione sono corretti



Come affrontare il testing? buon senso!

- ▶ Non affrontare tutti gli aspetti dell'object oriented testing contemporaneamente (difficile e costoso!)
- ▶ Una cosa alla volta e con tecniche differenti per ognuna. Porre attenzione alle interazioni significative e non a tutte le possibili combinazioni.
- ▶ Fattori in gioco per la scelta delle tecniche:
 - ◆ Applicazione sotto test, approccio di sviluppo seguito, garanzie di affidabilità del team di sviluppo, ambiente di sviluppo (+ linguaggio), tempi a disposizione, risorse a disposizione...



Un approccio sequenziale per lo sviluppatore

Più fasi, dalle singole classi alle loro interazioni.

► Intra-class testing (unit testing)

- Scopo: verificare la qualità di singole classi
- Focus: inheritance + state dependent behavior + exceptions
- Introduzione stub classes, oracoli.

► Inter-class testing

- Scopo: integrazione dei test per assicurare qualità generale
- Focus: hierarchy + polymorphism + execution related problems

► System and acceptance

- Con metodi tradizionali (derivare i test case dalle specifiche funzionali, non dipende dall'approccio di design)



Intra-class testing (unit testing)

1. Se **la classe è astratta**, comporre un insieme di istanze che coprano i casi significativi. Possono essere prese dall'applicazione (se disponibili) e/o create al solo scopo di essere testate.
2. Controllare la corretta invocazione dei **metodi ereditati** e di quelli ridefiniti (costruttori inclusi). Se vi sono ancestors già testati, determinare **quali** fra i metodi ereditati **devono essere testati nuovamente** e quali test case possono essere riutilizzati.
3. Progettare una serie di test case intra-class, basati sul **modello a macchina a stati del comportamento della classe** interessata.
4. Integrare i test case funzionali con quelli **strutturali**, aumentando il modello a macchina a stati con relazioni strutturali derivati dal codice della classe.
5. Progettare un insieme iniziale di test case per la **gestione delle eccezioni**, testando sistematicamente le eccezioni che devono essere sollevate dai metodi della classe e quelle che devono essere “trappate” e gestite.
6. Progettare un insieme iniziale di test case per le **chiamate polimorfiche** (considerando solo le chiamate locali).



Inter-class testing (integration testing)

1. Identificare i **“cluster” di classi** da testare con un metodo incrementale: prima le classi foglie (che non includono/usano altre classi) fino a salire alle classi radice.
2. Progettare un insieme di inter-class test cases di tipo **funzionale** per il cluster sotto test.
3. Integrare i test cases funzionali con quelli **strutturali**, basati sulla copertura strutturale dei test funzionali.
4. Integrare l'insieme dei test per la **gestione delle eccezioni** con test che provino la corretta gestione di eccezioni non locali.
5. Integrare l'insieme dei test per il polimorfismo con test che provino l'interazione inter-class delle **chiamate polimorfiche** (e del dynamic binding).



Integration Testing: come?

▶ Top-down

- Si parte da chi occupa il posto più in alto della gerarchia dettata dalla relazione d'uso. (no drivers, stub complessi)

▶ Bottom-up

- Duale della Top-down (no stubs, drivers complessi)

▶ Big-bang

- Tutti i moduli integrati contemporaneamente (no stubs, no drivers)

▶ Threads

- Si seguono i thread di esecuzione

▶ Critical modules

- Si segue un criterio di criticità (parti critiche prima!)



Considerazioni di stampo metodologico

- ▶ Il testing è una tecnica che viene usata in modo differente a seconda dello stampo metodologico che si adotta:
 - ♦ Testing per la verifica del codice e dei requisiti
 - ♦ Testing come strumento di sviluppo integrato

- ▶ In ogni caso è un fattore importante su più fronti:
 - ♦ Innalzamento qualità del software prodotto
 - ♦ Abbattimento costi di sviluppo.

- ▶ Pareri a riguardo
 - ♦ Qualche nome: Fowler, Beck, Cockburn, Hamill...
 - ♦ Alcuni articoli sono un po' datati ma rappresentano i fondamenti di questo ambito



The Quality Process (I)

- ▶ **Scopo: ottenere una adeguata **dependability****
 - ♦ Correttezza / Affidabilità / Sicurezza / Robustezza
 - ♦ Selezionare quali attività svolgere ed organizzarle
 - ♦ Valutare interazioni e trade-off con altri obiettivi importanti

- ▶ **Proprietà**
 - ♦ Completeness: scovare **ogni** classe importante di faults
 - ♦ Timeliness: individuazione **tempestiva** (high leverage)
 - ♦ Cost-effectiveness: selezionare le attività in base al loro **costo** e alla loro **efficacia**.
 - considerare il costo sull'intero ciclo di sviluppo e di vita del prodotto
 - è probabile che il fattore dominante sia il costo dovuto alla reiterazione di un'attività attraverso molti cicli di modifica



The Quality Process (2)

► Process Visibility

- ♦ A che punto siamo? In anticipo? In ritardo?
- ♦ E' importante avere un'indicazione forte della qualità del sistema **prima** che questo raggiunga la fase finale di testing
 - Pianificare le attività il **prima possibile**
 - Applicare tecniche di analisi su porzioni di codice prima del loro inserimento nel codice definitivo
 - Il numero di errori di design/codifica non è una misura dell'affidabilità del sistema testato. Il numero di errori trovati che non ricadono nelle previsioni di design è un indicatore preliminare di possibili problemi di qualità.



Martin Fowler

- ▶ Articolo “Testing Methods: The Ugly Duckling”, giugno 1998, spiega le linee guida per fare testing in modo produttivo e evidenzia i vantaggi derivanti dallo sviluppo incrementale accoppiato allo unit e integration testing (“Self-Testing Code”, “test code before production code”, “enough testing code”).
- ▶ Articolo “Keeping Software Soft”, dicembre 1998: affronta i problemi derivanti dalla dinamicità del progetto e sviluppo software indicando nel **dynamic design** una via percorribile.
 - Precondizioni richieste: tests + oggetti + refactoring.
 - Risultati: no “fear of changing requirements” + “be responsive to your users without sacrificing your design” + future (!)



Martin Fowler

«Early on in my programming life I discovered that I spent much more time and effort removing bugs than I did writing code. If you run tests regularly, bugs tend to show up earlier. If it's not long since your last write of a test, you know which bit of code contains the bug.

After writing self-testing code for a while, developers realize that they are spending less time debugging and thus developing faster. The tests enable them to refactor more easily, thus keeping the system design simpler and allowing them to develop faster.»



Paul Hamill

(Sviluppatore con esperienza decennale (Java, C++). Esperto di XP e unit testing)

- ▶ Data l'importanza di eseguire regolarmente i test, sono stati realizzati dei frameworks per semplificare le operazioni necessarie (esecuzione e controllo degli esiti). Tutti derivano da quello ideato da Kent Beck per Smalltalk (SUnit) pubblicato nel 1999, il suo porting più famoso e diffuso è JUnit (opera di E. Gamma).
- ▶ “Unit Test Frameworks”, O'Reilly, 2004



Alistair Cockburn

- ▶ Nei primi anni '90 fu consulente per IBM, con il compito di definire e documentare una metodologia per lo sviluppo object-oriented. Prese visione del maggior numero possibile di progetti, considerando ciò che i vari teams indicavano come fattore importante per il successo conseguito (o fallimento subito). Le conclusioni furono sorprendenti.
- ▶ « These results have been consistent, from 1991 to 1999, from Hong Kong to Americas, to Norway and South Africa, in COBOL, Smalltalk, Java, VB, Sapiens and Synon. The shortest statement of the results are:
 - ♦ To the extent you can replace written documentation with face-to-face interactions, you can reduce reliance on written work products and improve the likelihood of delivering the system.
 - ♦ The more frequently you can deliver running, tested slices of the system, the more you can reduce reliance on written “promissary” notes and improve the likelihood of delivering the system.
- ▶ Written, reviewed requirements and design documents are “promises” for what will be built, serving as timed progress markers. There are times when creating them is good. However, a more accurate timed progress marker is running, tested code. It is more accurate because it is not a timed promise, it is a timed accomplishment. »



Riferimenti

► Materiale usato per la presentazione

- Mauro Pezzè, Michal Young, “Testing OO Software”, (materiale)
- Martin Fowler, “Mocks objects aren't stubs”, “Testing Methods: The Ugly Duckling”, “Keeping Software Soft” <http://www.martinfowler.com>
- Alessandro Orso, Sergio Silva, “Open Issues and Research Directions in Object-Oriented Testing”, Gennaio 1998
- Ugo Buy, Carlo Ghezzi, Alessandro Orso, Mauro Pezzè, Matteo Valsasna, “A Framework for Testing Object Oriented Components”, May 1999

► Spunti per approfondimenti:

- Cem Kaner, James Bach, Bret Pettichord, “Lessons Learned in Software Testing”, Wiley, 2001
- Alistair Cockburn, “Surviving Object-Oriented Projects”, Addison-Wesley Professional, 1997
- Paul Hamill, “Unit Test Frameworks”, O'Reilly, 2004