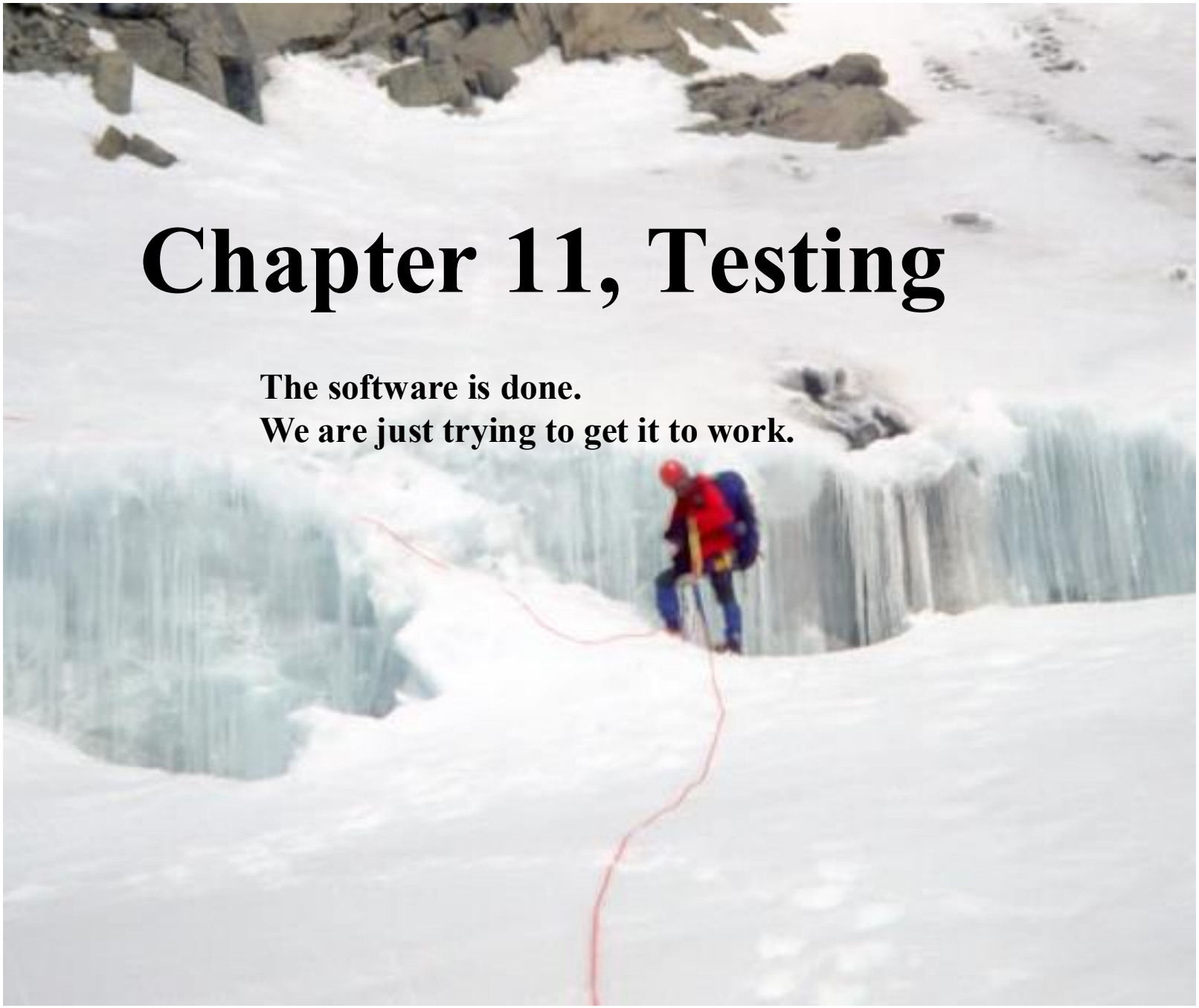
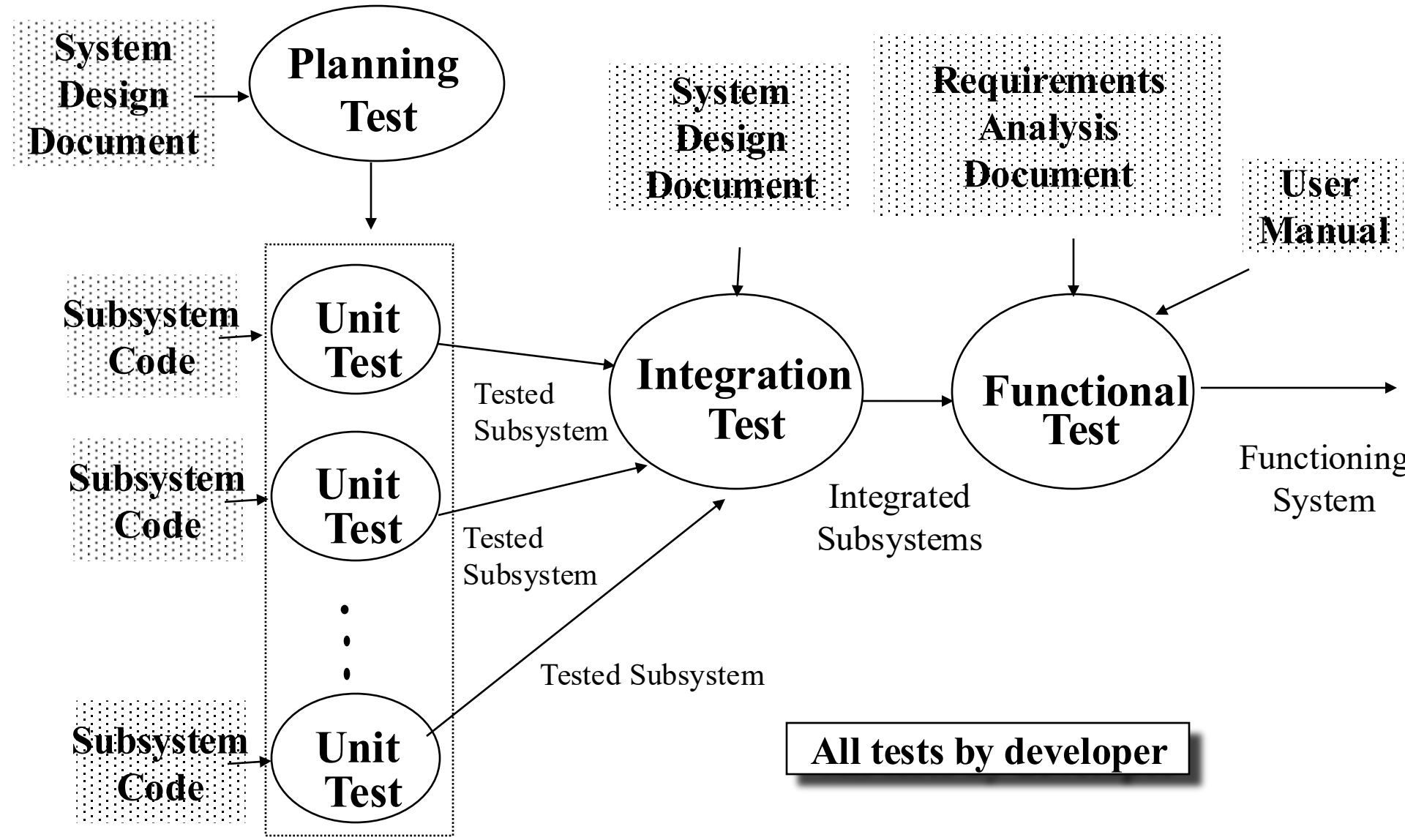


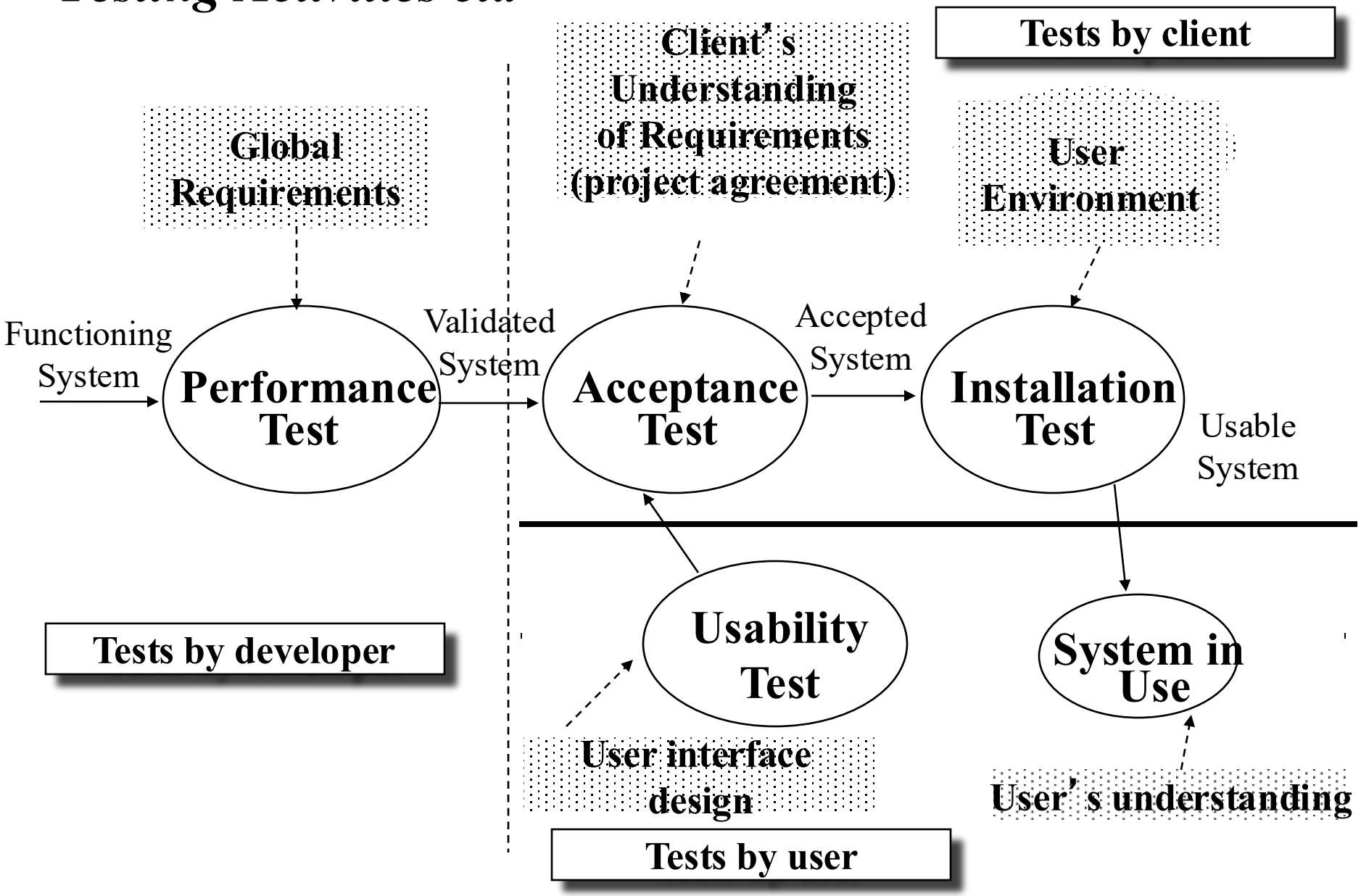
Chapter 11, Testing

The software is done.
We are just trying to get it to work.

Testing Activities



Testing Activities ctd



White-box Testing

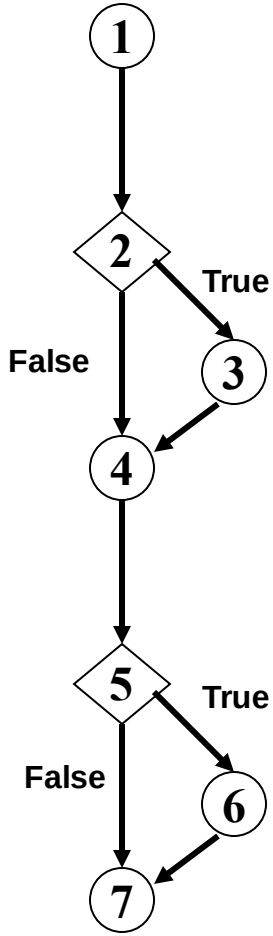
- Si focalizza sulla struttura interna della componente. Indipendentemente dall'input, ogni stato nel modello dinamico dell'oggetto e ogni integrazione tra gli oggetti viene testata.
  - **Completezza. Es.** Ogni statement nella componente è eseguito almeno una volta.
- 4 tipi di white-box testing
  - **Statement Testing**
  - **Loop Testing**
  - **Path Testing**
  - **Branch Testing**

White-box Testing

- Statement Testing (Algebraic Testing)
  - **Si testano i singoli statement**
- Loop Testing: definire in casi di test in modo da assicurasi che:
  - **Il loop che deve essere saltato completamente.**
  - **Loop da eseguire esattamente una volta**
  - **Loop da eseguire più di una volta**
- Path testing:
  - **Assicurarsi che tutti i path nel programma siano eseguiti**
- Branch Testing (Conditional Testing)
  - **assicurarsi che ogni possibile uscita da una condizione sia testata almeno una volta**

```
if ( i =TRUE) printf("YES\n");else printf("NO\n");
Test cases: 1) i = TRUE; 2) i = FALSE
```

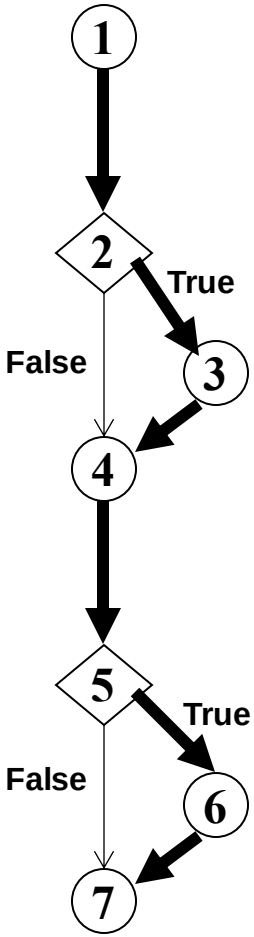
Program Control Graph of computeFine()



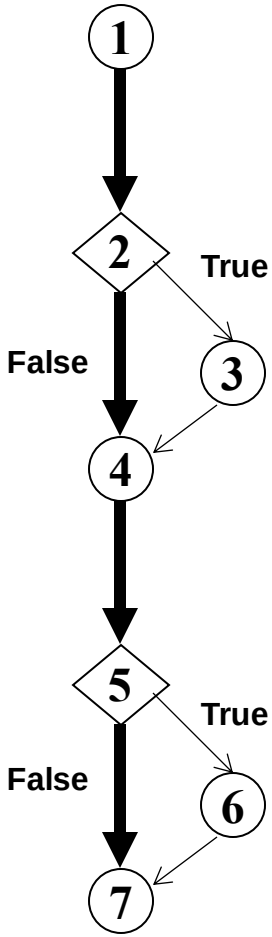
© 2010 John Wiley & Sons Ltd.

Distinct Paths of computeFine()

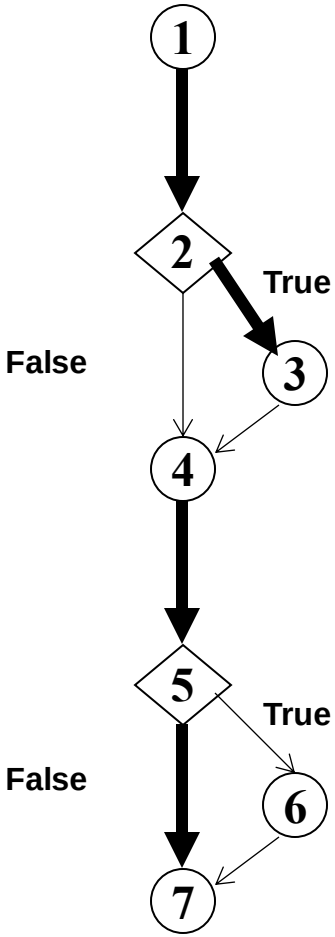
Path #1



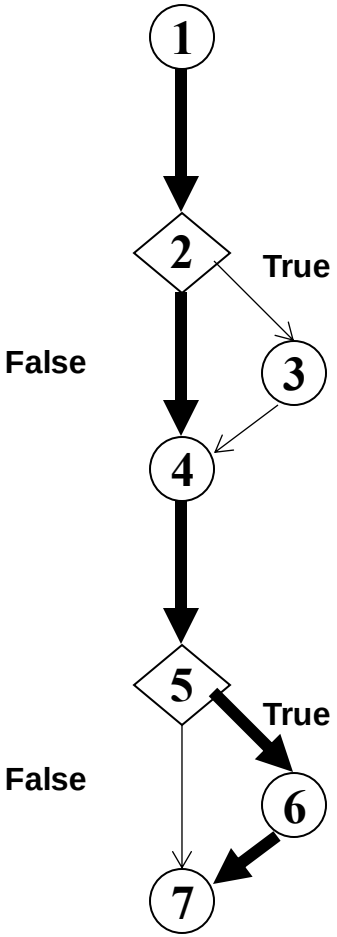
Path #2



Path #3



Path #4



© 2010 John Wiley & Sons Ltd.

Test Cases for Distinct Paths of computeFine()

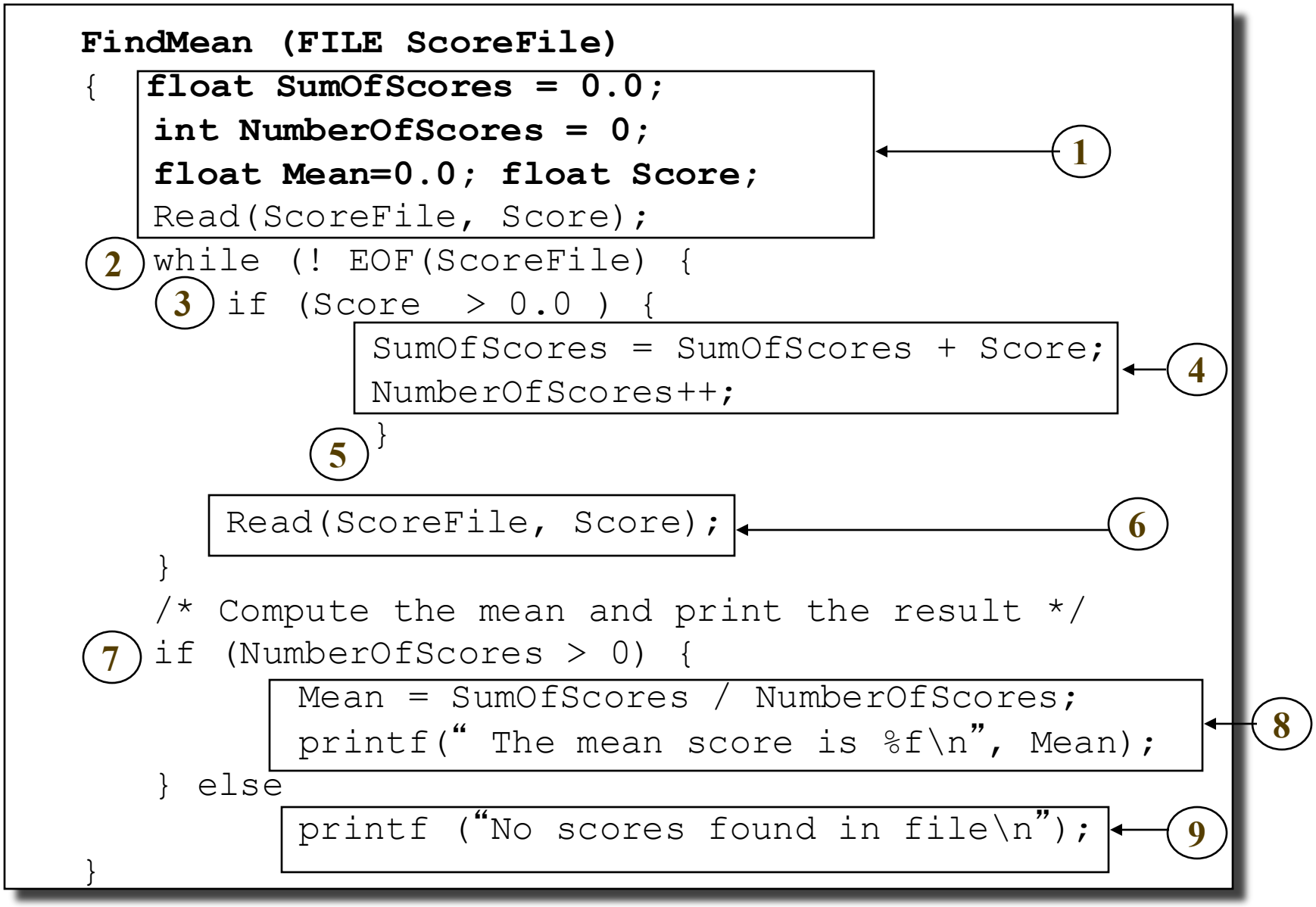
Test Case #	daysLate	printON	Path
1	1	TRUE	1-2-3-4-5-6 7
2	60	FALSE	1-2-4-5--7
3	1	FALSE	1-2-3-4-5-7
4	60	TRUE	1-2-4-5-6-7

© 2010 John Wiley & Sons Ltd.

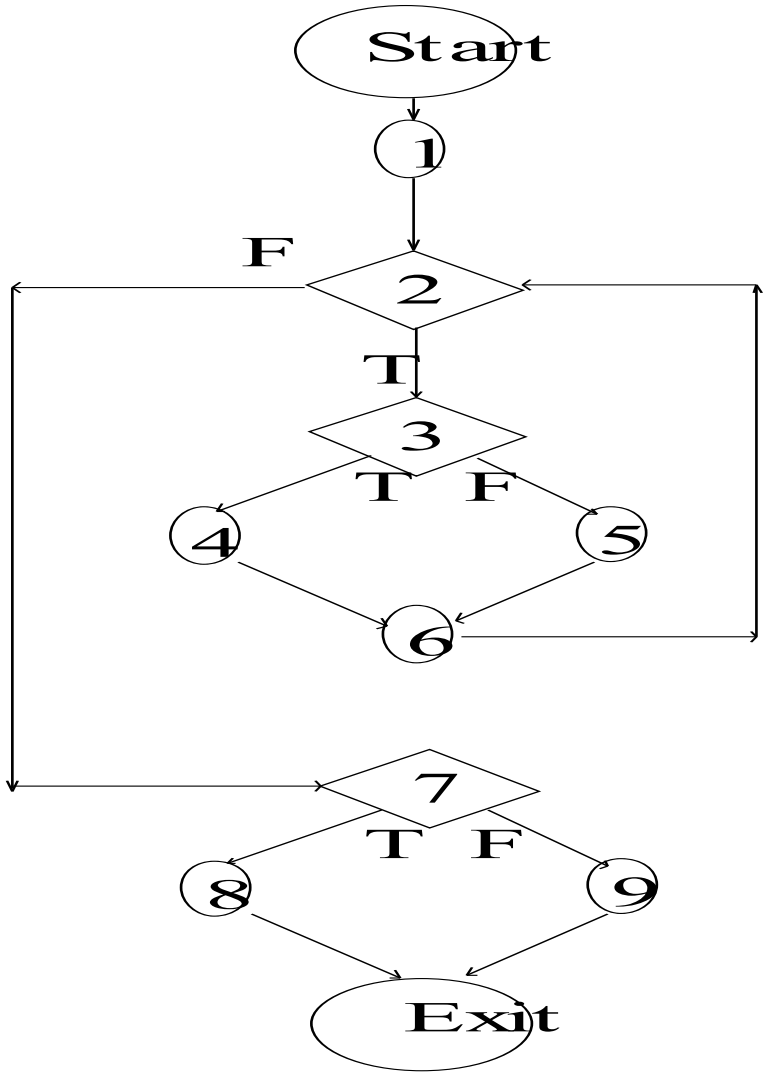
White-box Testing Example

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
                if ( Score > 0.0 ) {
                 SumOfScores = SumOfScores + Score;
                NumberOfScores++;
               }
 Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
              Mean = SumOfScores/NumberOfScores;
              printf("The mean score is %f \n",  Mean);
 } else
              printf("No scores found in file\n");
}
```

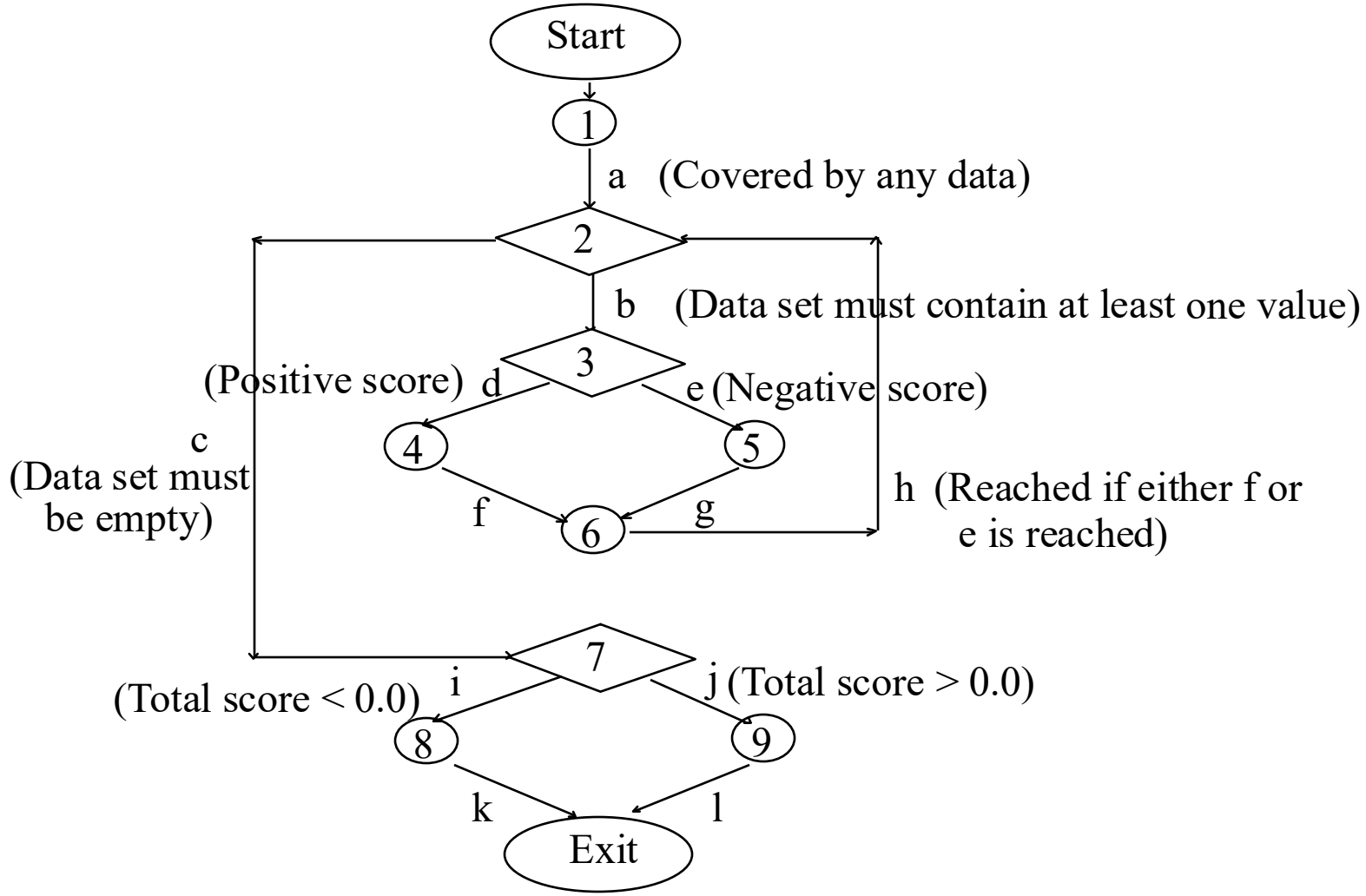
White-box Testing Example: Determining the Paths



Constructing the Logic Flow Graph



Finding the Test Cases



Test Cases

- Test case 1 : ? (To execute loop exactly once)
- Test case 2 : ? (To skip loop body)
- Test case 3: ?,? (to execute loop more than once)

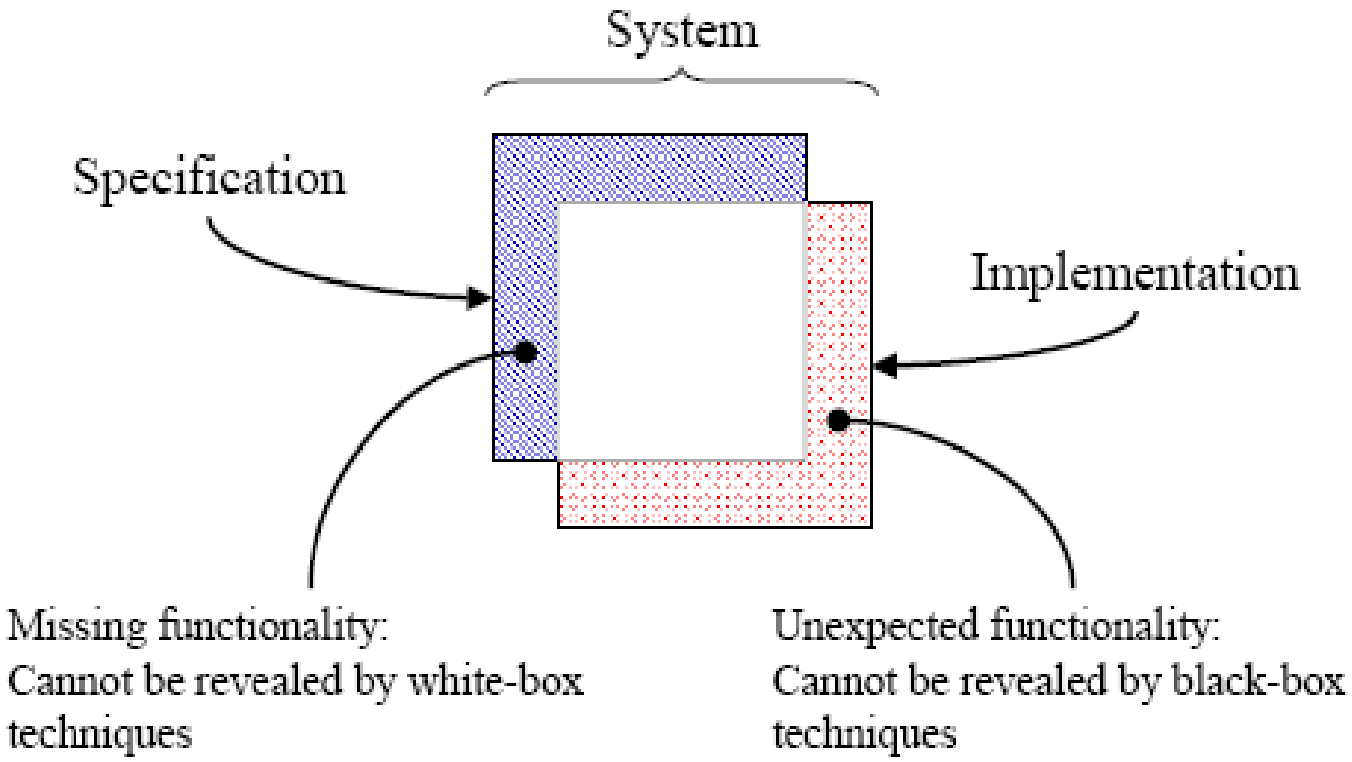
- These 3 test cases cover all control flow paths

- Complessità ciclomatica di McCabe: #di cammini linearmente indipendenti
- Vedere esempio Libro

Comparison of White & Black-box Testing

- White-box Testing:
  - **Potentially infinite number of paths have to be tested**
  - **White-box testing often tests what is done, instead of what should be done**
  - **Cannot detect missing use cases**
- Black-box Testing:
  - **Potential combinatorical explosion of test cases (valid & invalid data)**
  - **Often not clear whether the selected test cases uncover a particular error**
  - **Does not discover extraneous use cases ("features")**
- Both types of testing are needed
- White-box testing and black box testing are the extreme ends of a testing continuum.
- Any choice of test case lies in between and depends on the following:
  - **Number of possible logical paths**
  - **Nature of input data**
  - **Amount of computation**
  - **Complexity of algorithms and data structures**

Black vs. White Box Testing



The 4 Testing Steps

1. Select what has to be measured
   - **Completeness of requirements**
   - **Code tested for reliability**
   - **Design tested for cohesion**
2. Decide how the testing is done
   - **Code inspection**
   - **Black-box, white box,**
   - **Select integration testing strategy (big bang, bottom up, top down, sandwich)**

3. Develop test cases
   - **A test case is a set of test data or situations that will be used to exercise the unit (code, module, system) being tested or about the attribute being measured**
4. Create the test oracle
   - **An oracle contains of the predicted results for a set of test cases**
   - **The test oracle has to be written down before the actual testing takes place**

Guidance for Test Case Selection

- Use *analysis knowledge* about functional requirements (black-box):
  - **Use cases**
  - **Expected input data**
  - **Invalid input data**
- Use *design knowledge* about system structure, algorithms, data structures (white-box):
  - **Control structures**
    - **Test branches, loops, ...**
  - **Data structures**
    - **Test records fields, arrays, ...**

- Use *implementation knowledge* about algorithms:
  - **Force division by zero**

1. Create unit tests as soon as object design is completed:
- **Black-box test: Test the use cases & functional model**
- **White-box test: Test the dynamic model**
- **Data-structure test: Test the object model**

2. Develop the test cases
- **Goal: Find the minimal number of test cases to cover as many paths as possible**

3. Cross-check the test cases to eliminate duplicates
- **Don't waste your time!**

4. Desk check your source code
- **Reduces testing time**

5. Create a test harness
- **Test drivers and test stubs are needed for integration testing**

6. Describe the test oracle
- **Often the result of the first successfully executed test**

7. Execute the test cases
- **Don't forget regression testing**
- **Re-execute test cases every time a change is made.**

8. Compare the results of the test with the test oracle
- **Automate as much as possible**

Integration testing

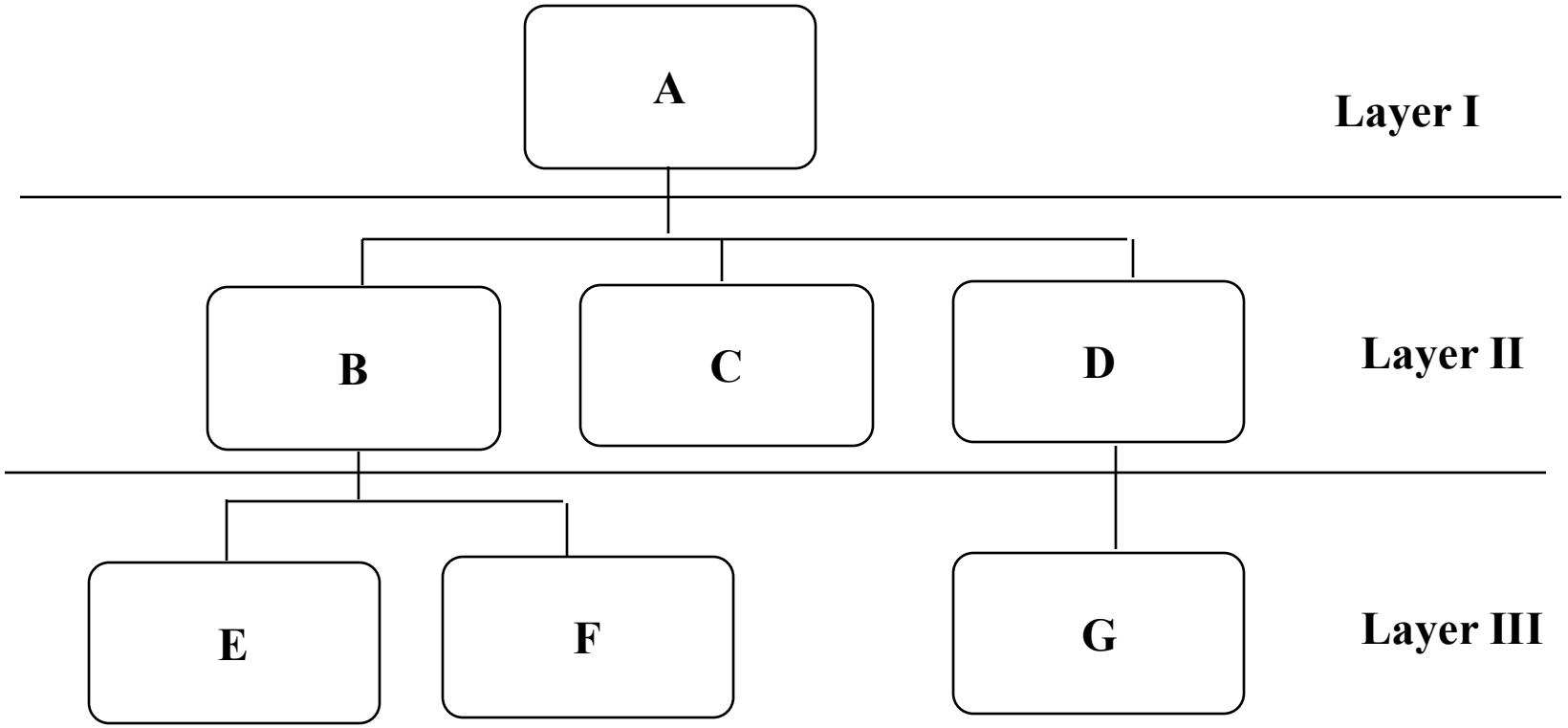
- Quando i bug in ogni componente sono stati rilevati e riparati, le componenti sono pronte per essere integrate in sottosistemi più grandi.
- **Integration testing** rileva bug che non sono stati determinati durante lo unit testing, focalizzandosi su un insieme di componenti che sono integrate
- Due o più componenti sono integrate e analizzate, e quando dei bug sono rilevati nuove componenti possono essere aggiunte per riparare i bug
- Sviluppare test stub e test driver per un test di integrazione sistematico è time-consuming
- L'ordine in cui le componenti sono integrate può influenzare lo sforzo richiesto per l'integrazione
  - **Vedremo diverse strategie di testing per ordinare le componenti da testare**

Component-Based Testing Strategy

- L'ordine in cui i sottosistemi sono selezioanti per il testing e l'integrazione determina la strategia di testing
  - **Big bang integration (Nonincremental)**
  - **Bottom up integration**
  - **Top down integration**
  - **Sandwich testing**
  - **Variations of the above**

- Ognuna di queste strategie è stata originariamente concepita per una decomposizione gerarchica del sistema
  - **ogni componente appartiene ad una gerarchia di layer, ordinati in base all'associazione "Call"**

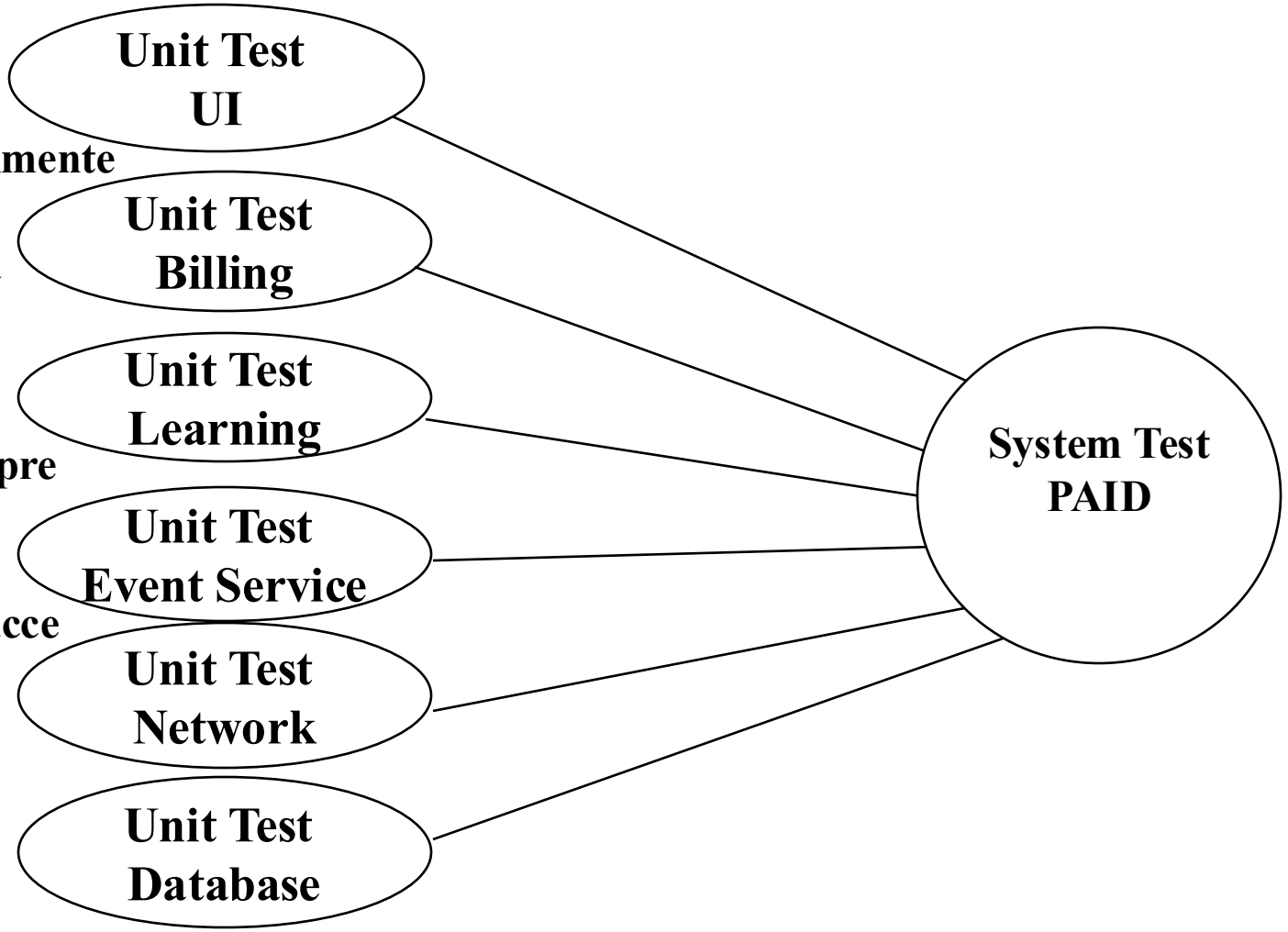
Example: Three Layer Call Hierarchy



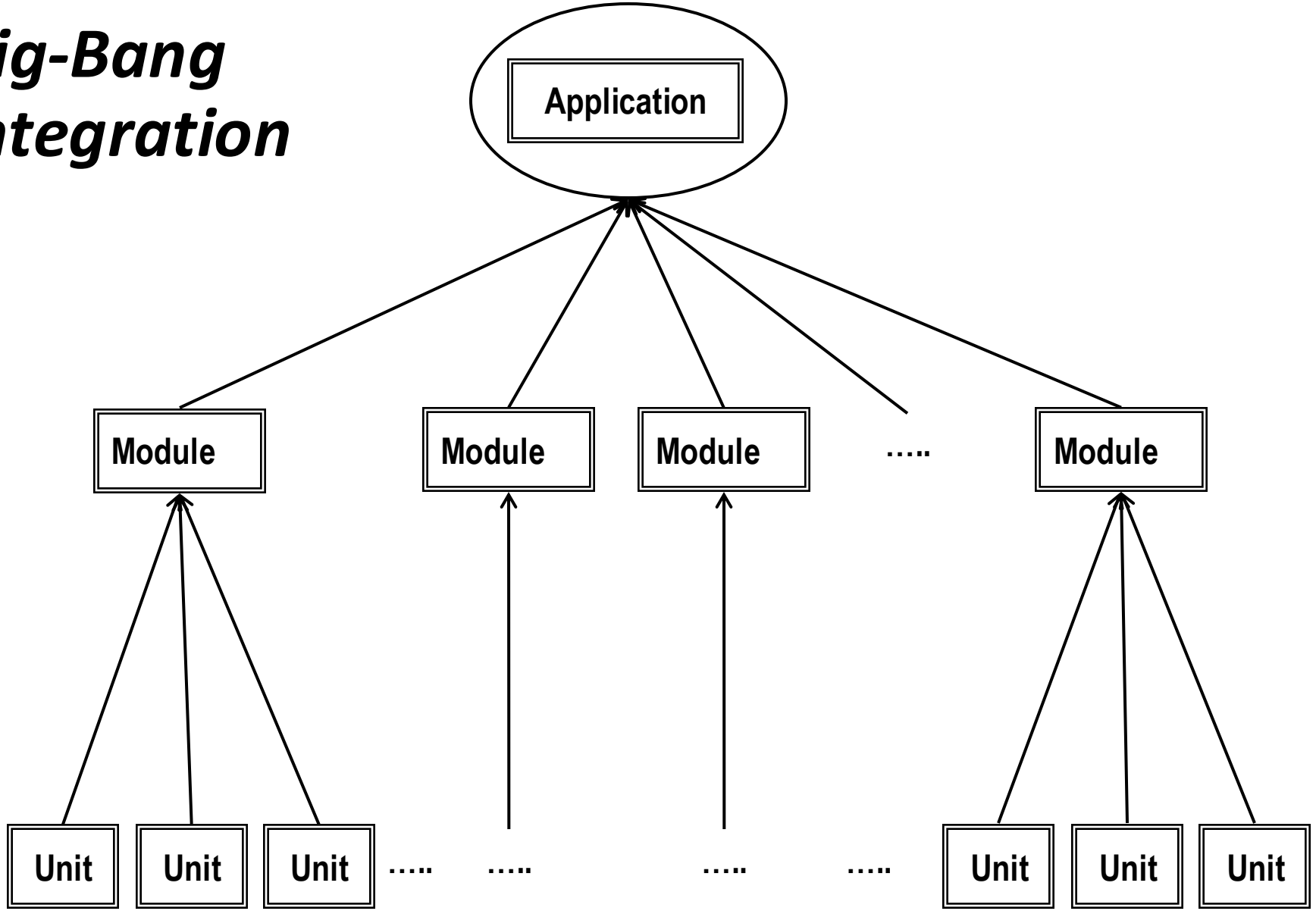
Integration Testing: Big-Bang Approach

Le componenti sono prima testate individualmente e poi testate insieme come un singolo sistema

Sebbene sia semplice, è costoso: se un test scopre un fallimento è impossibile distinguere i fallimenti nelle interfacce dai fallimenti all'interno delle componenti

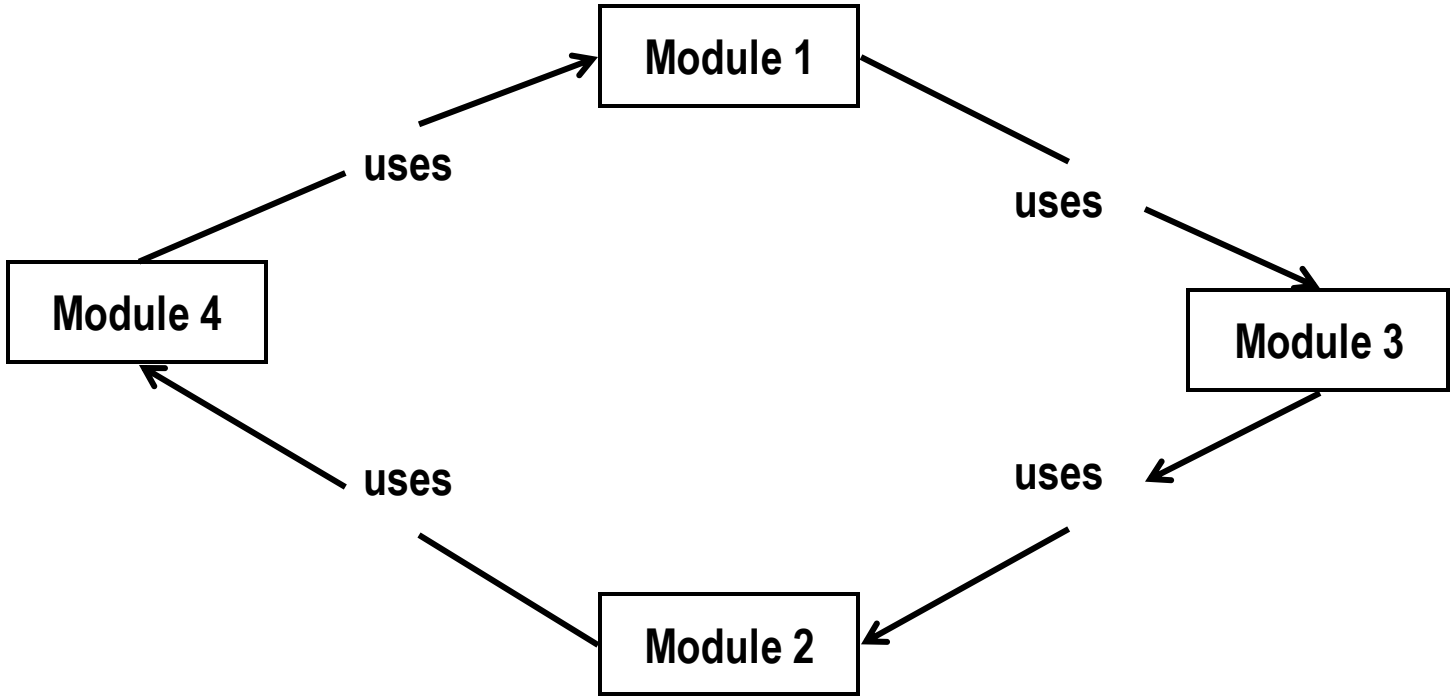


Big-Bang Integration



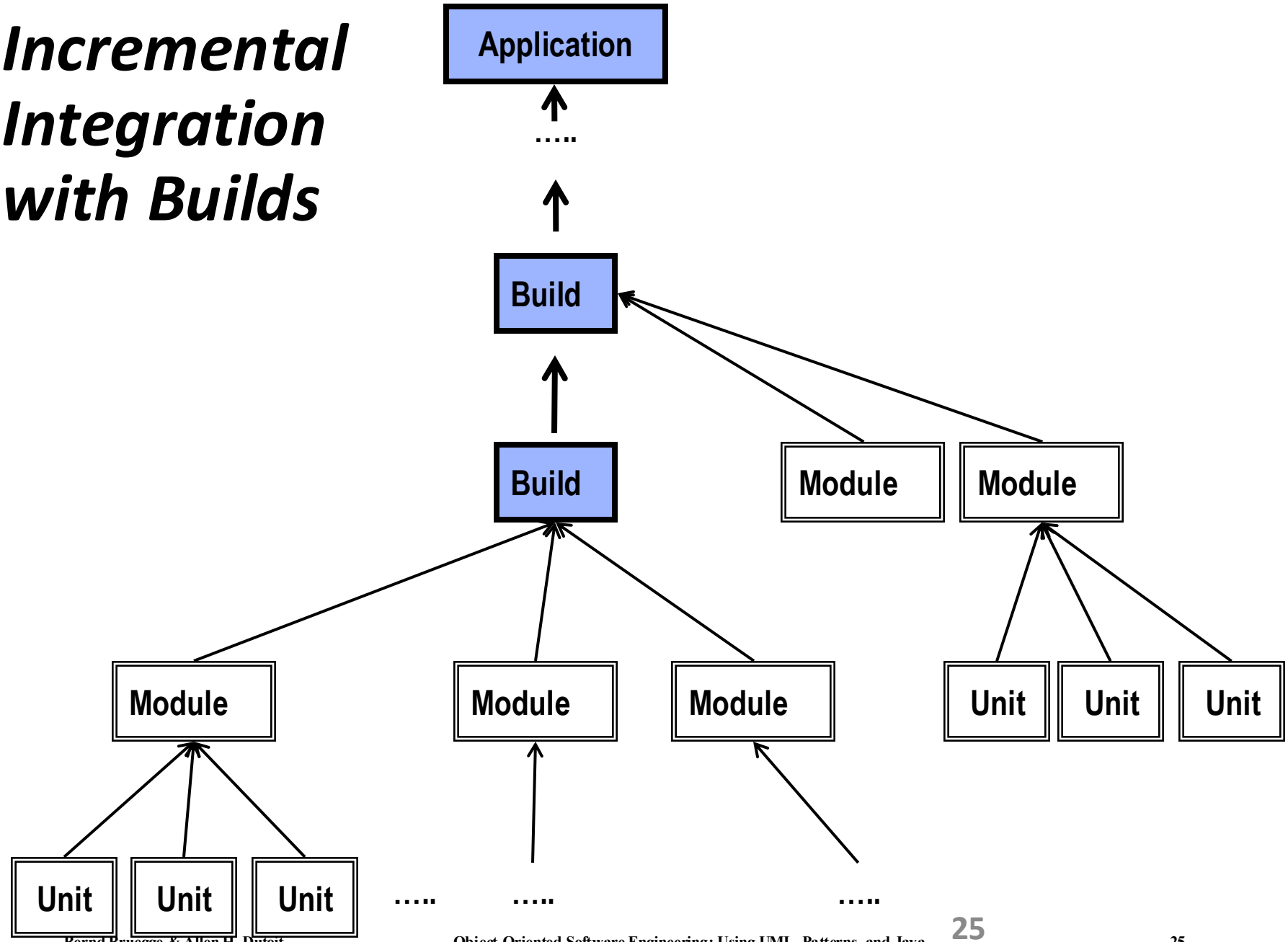
© 2010 John Wiley & Sons Ltd.

Module Self-Dependency ➔ Big Bang Integration



© 2010 John Wiley & Sons Ltd.

Incremental Integration with Builds

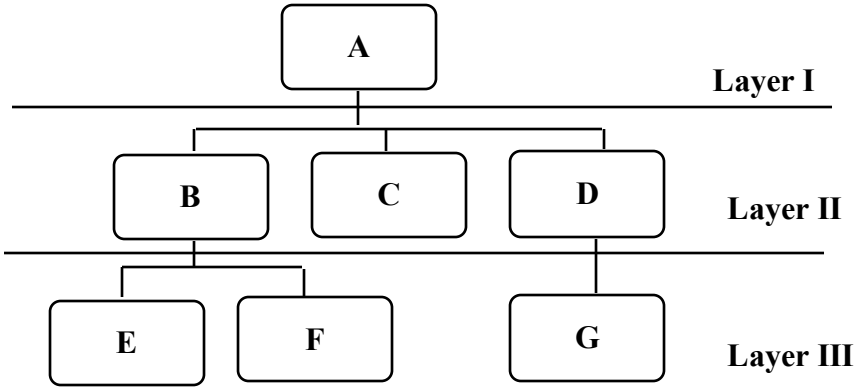


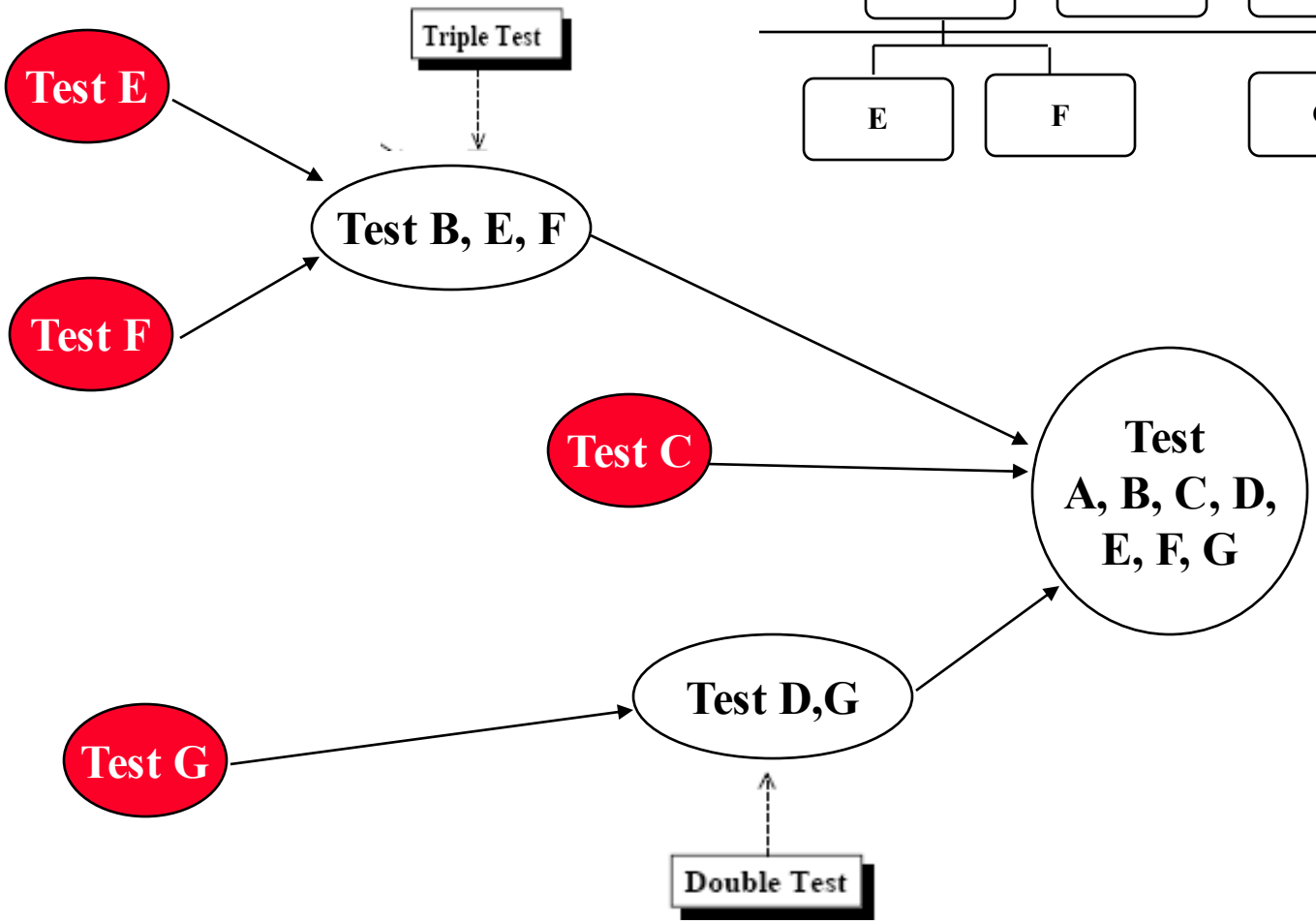
© 2010 John Wiley & Sons Ltd.

Bottom-up Testing Strategy

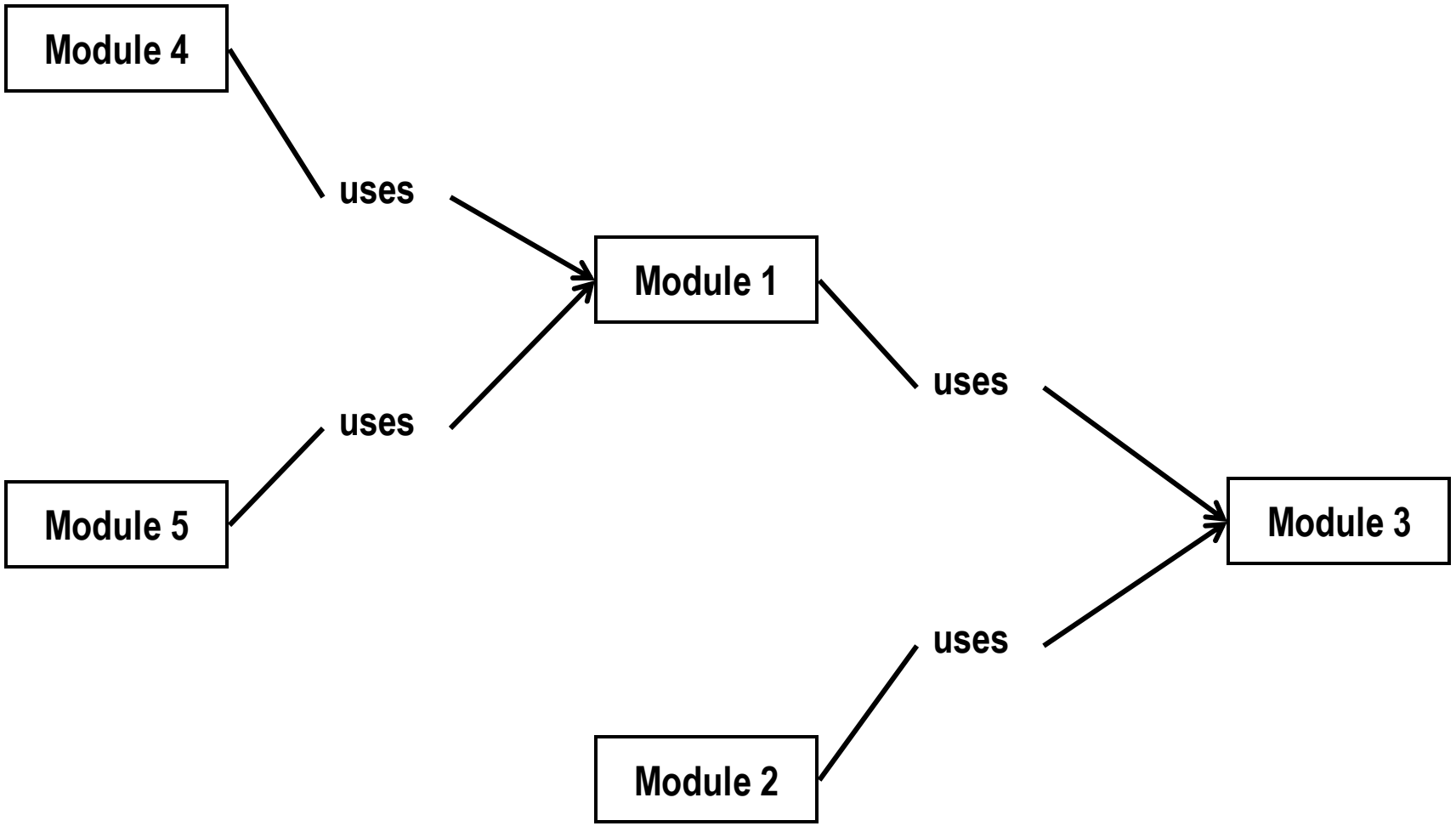
- I sottosistemi nel layer più in basso della gerarchia sono testati individualmente
- Poi i prossimi sottosistemi che sono testati sono quelli che "chiamano" i sottosistemi testati in precedenza
- Si ripete questo passo finchè tutti i sottosistemi sono testati
- I **test driver** sono usati per simulare le componenti dei layer più "in alto" che non sono stati ancora integrati

Bottom-up Integration



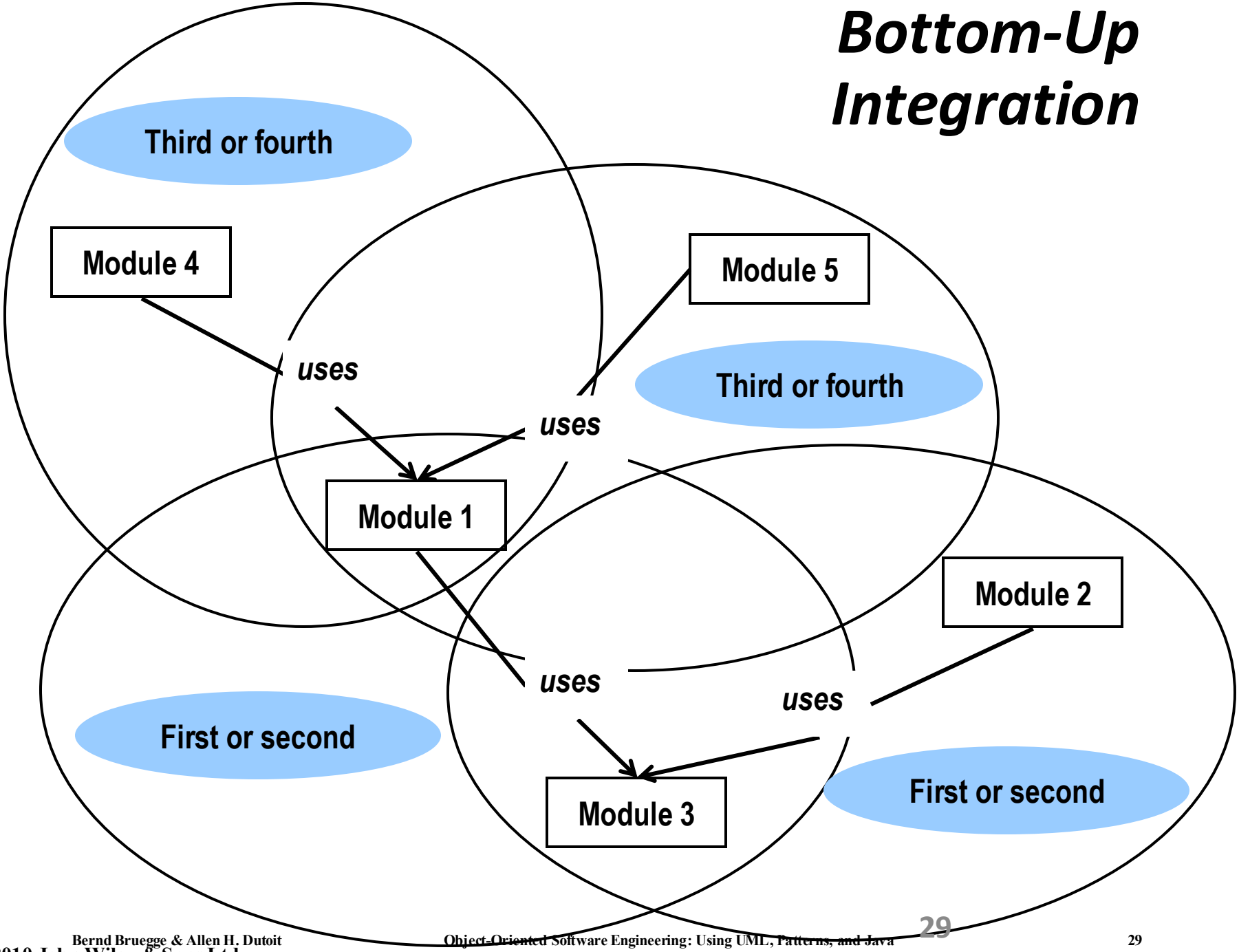


Module Dependencies



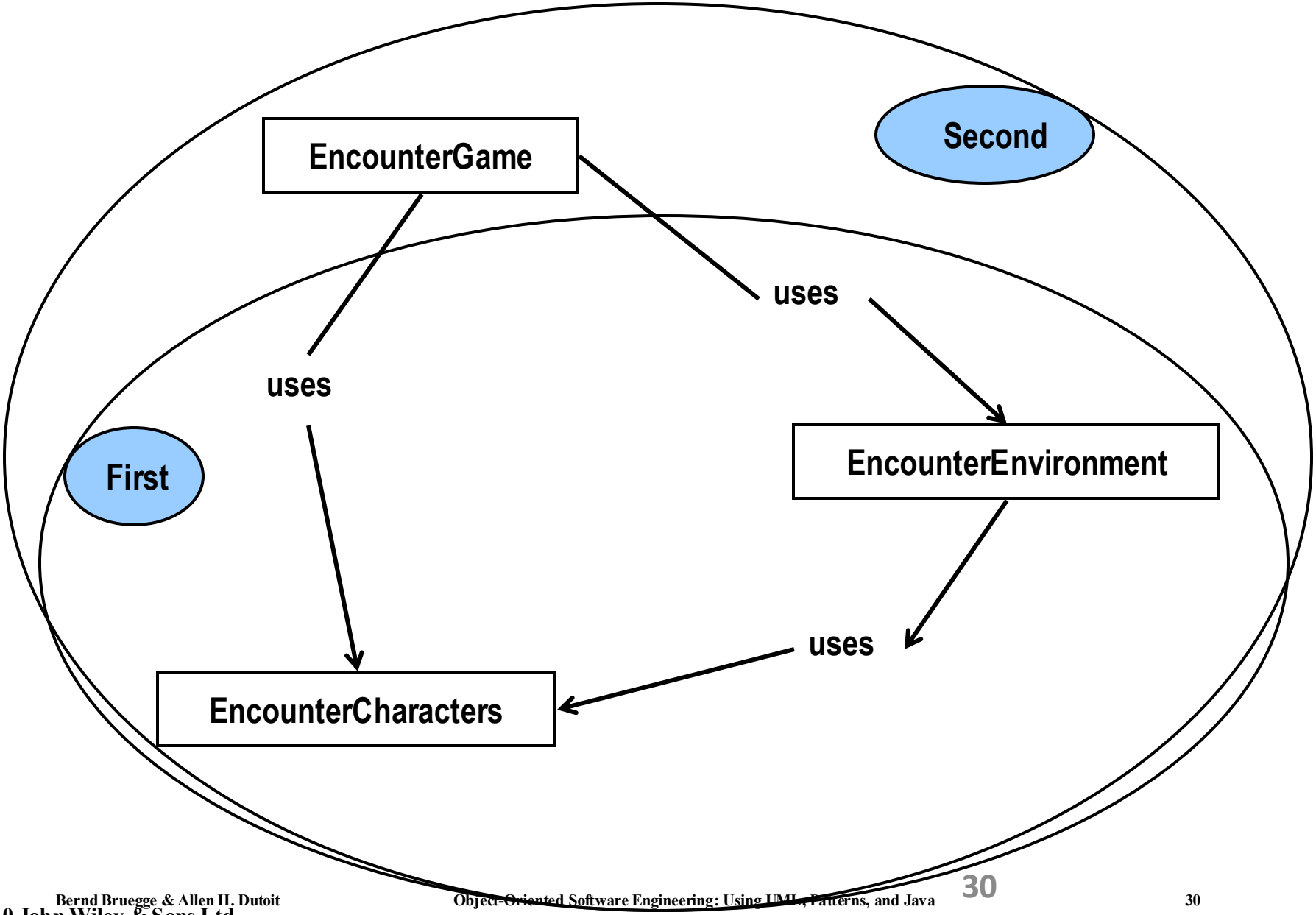
© 2010 John Wiley & Sons Ltd.

Bottom-Up Integration



© 2010 John Wiley & Sons Ltd.

Bottom-Up Integration Testing in Encounter



© 2010 John Wiley & Sons Ltd.

Pros and Cons of bottom up integration testing

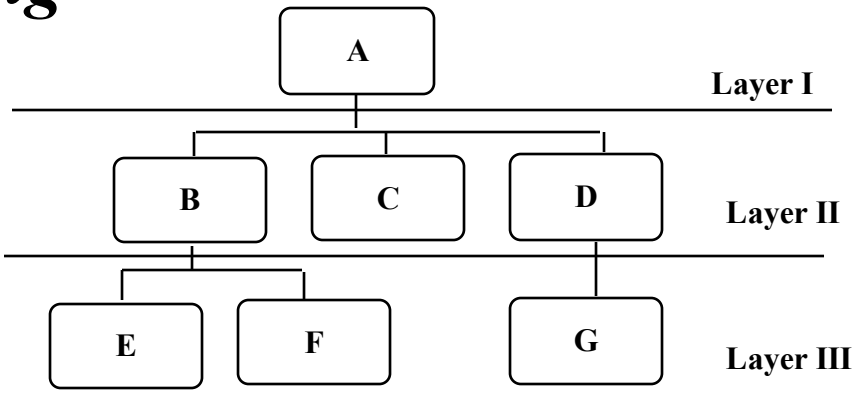
- Non buono per sistemi decomposti funzionalmente:
  - **Testa i sottosistemi più importanti alla fine**

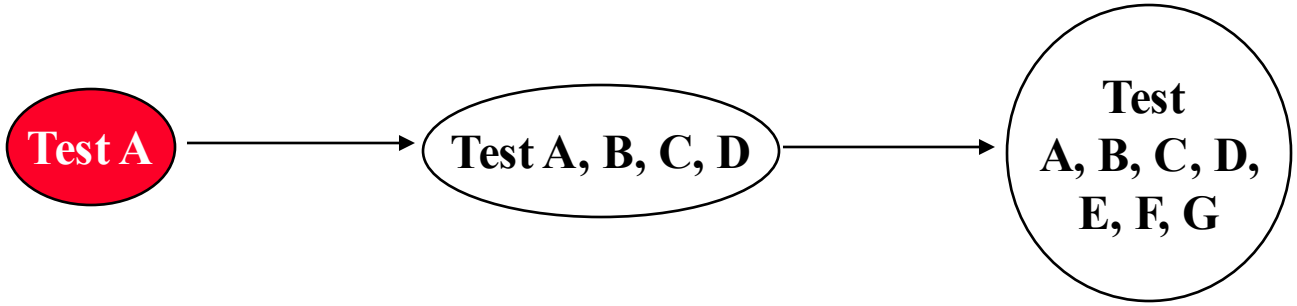
- Utile per integrare i seguenti sistemi
  - **Sistemi Object-oriented**
  - **Sistemi real-time**
  - **Sistemi con rigide richieste sulle performance**

Top-down Testing Strategy

- Testa prima i layer al top o i sottosistemi di controllo
- Poi combina tutti i sottosistemi che sono chiamati dai sottosistemi testati e quindi testa la collezione risultante di sottosistemi
- Itera questa attività fino a quando tutti i sistemi non sono integrati e testati
- I **test stub** sono usati per simulare le componenti dei layer al bottom che non sono state ancora integrate
  - **A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data.**

Top-down Integration Testing



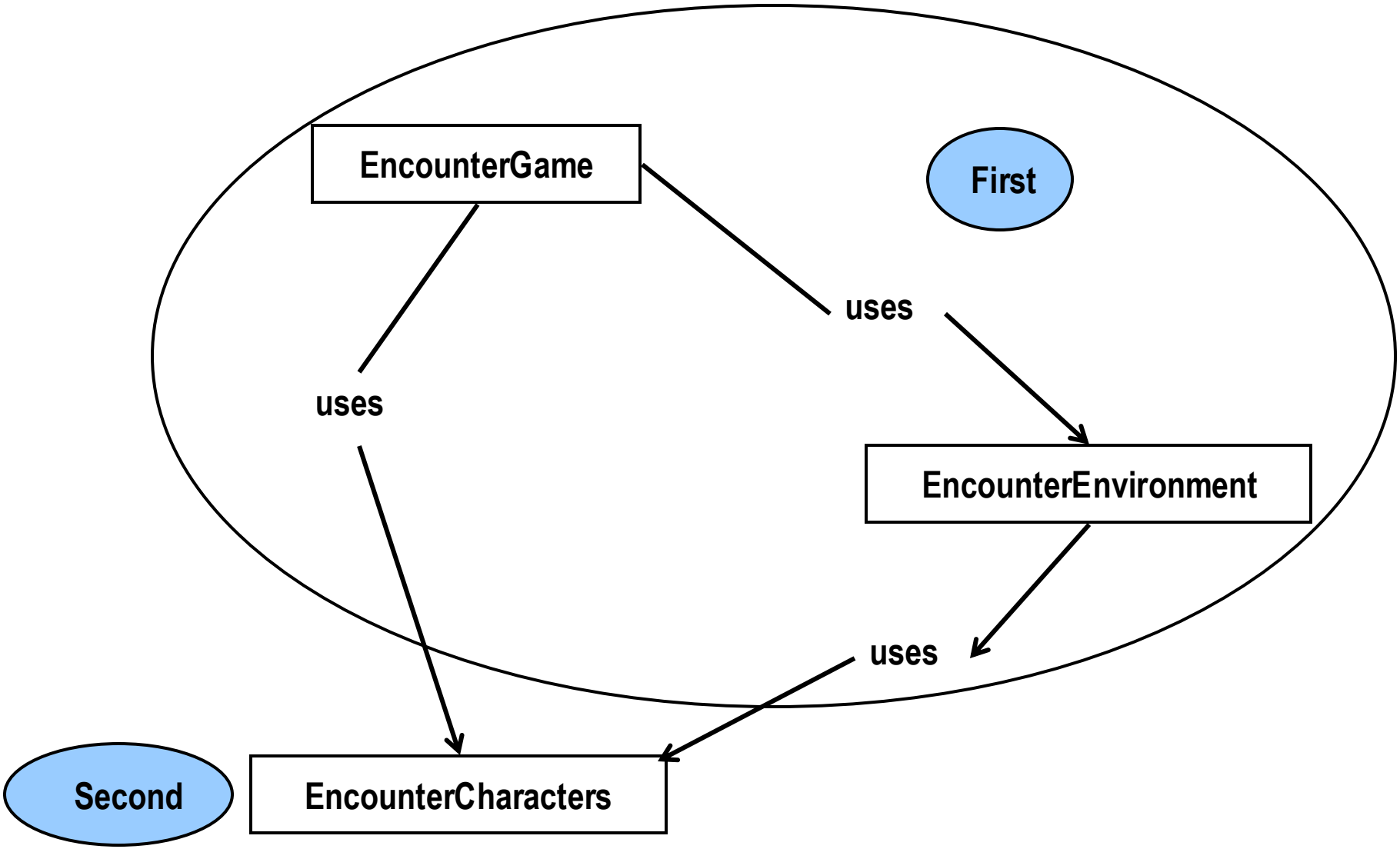


Layer I

Layer I + II

All Layers

Top-Down Integration Testing in Encounter



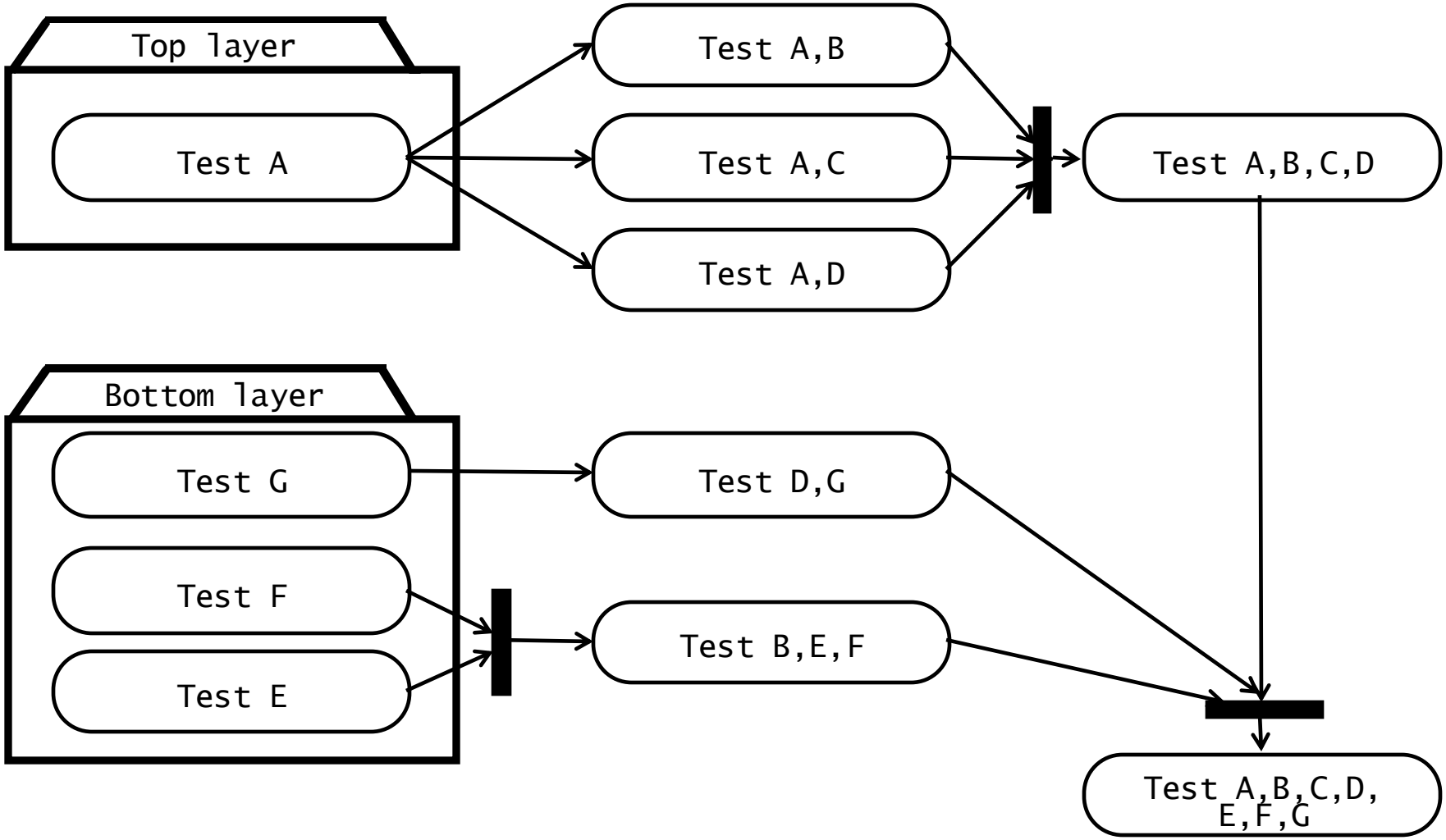
© 2010 John Wiley & Sons Ltd.

Pros and Cons of top-down integration testing

- Pros:
  - **I test cases possono essere definiti in termine delle funzionalità del sistema**
  - **Si possono riutilizzare nelle varie iterazioni**
- Cons:
  - **Scrivere gli stub può essere difficile: gli stub devono consentire tutte le possibili condizioni da testare**
  - **É possibile che un grande numero di stub sia richiesto, specialmente se il livello più in basso del sistema contiene molti metodi.**
- Una soluzione per evitare molti stub: ***Modified top-down testing strategy***
  - **Testa individualmente ogni layer della decomposizione prima di fare il merge dei layer**
  - **Svantaggio di questa modifica: Sono necessari sia test stub che test driver**

Sandwich Testing Strategy

- Combina l'uso di top-down strategy con bottom-up strategy
- *Il sistema è visto come se avesse 3 layers (va riformulato)*
  - **Un layer target nel mezzo**
  - **Un layer sopra il target**
  - **Un layer sotto il target**
  - **Il testing converge al layer target**
- Come selezionare il layer target se ci sono più di 3 layers?
  - **Heuristic: tenta di minimizzare il numero di stub e di driver**



Sandwich testing strategy

Pros and Cons of Sandwich Testing

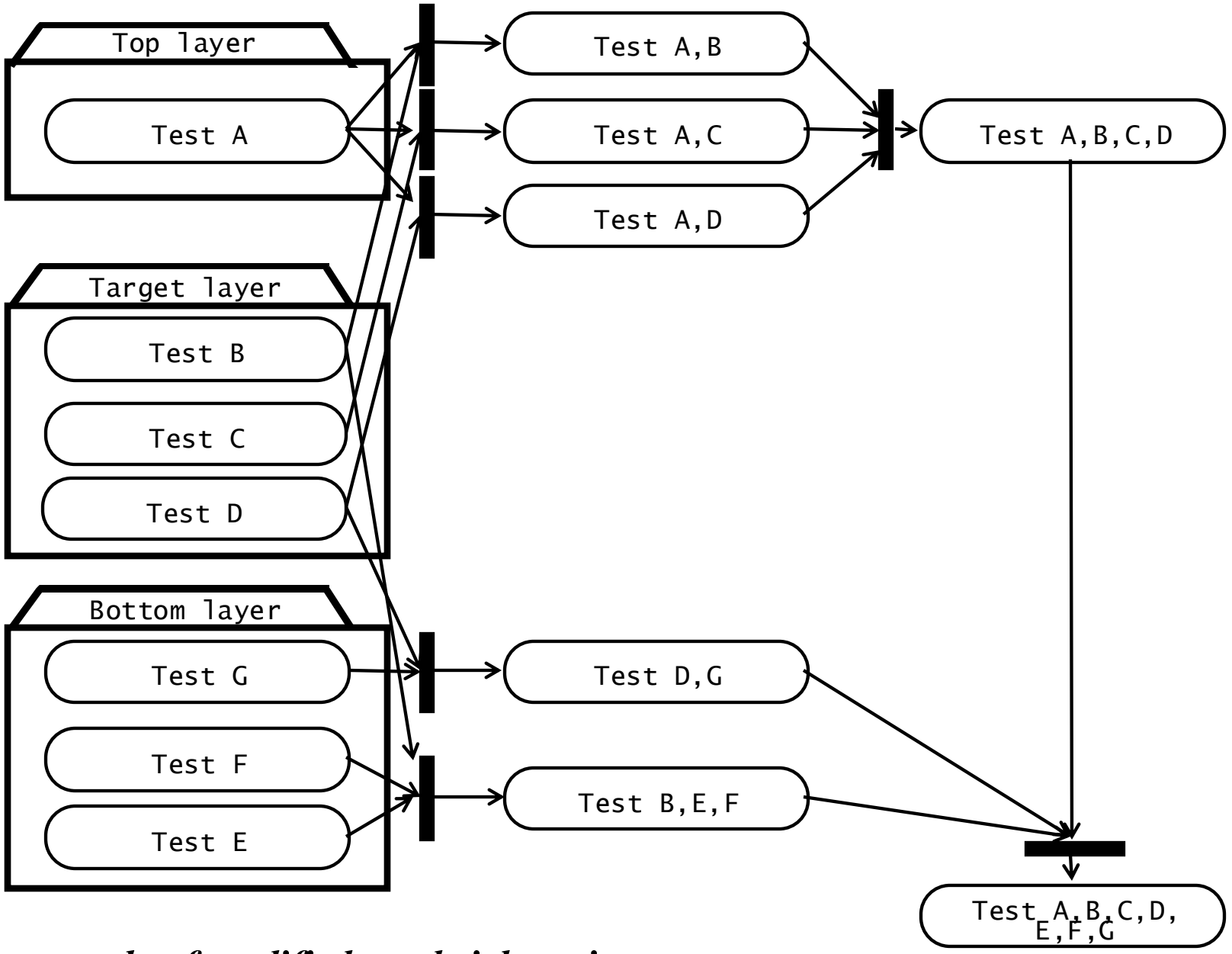
- PRO: I test dei layer al top e al bottom possono essere fatti in parallelo

- Problema: non testa i sottosistemi del target layer individualmente prima dell'integrazione
- Soluzione: **Modified sandwich testing strategy**

Modified Sandwich Testing Strategy

- Testa i 3 layer individualmente prima di combinarli in test incrementali con gli altri.

- I test individuali dei layer, consistono di un gruppo di 3 test (in parallelo):
  - **Test del layer al top (con stub per il layer target)**
  - **Test del layer nel mezzo (con driver and stub per i layer al top e al bottom rispettivamente)**
  - **Test del layer al bottom (con driver per il layer target)**
- I test dei layer combinati consistono in 2 test:
  - **Il layer al top accede al layer target. Può riusare i test del layer target dai test individuali, sostituendo i driver con le componenti del layer al top**
  - **Il layer al bottom è acceduto dal layer target. Può riusare i test del layer target dai test individuali, sostituendo gli stub con le componenti del layer al bottom**



An example of modified sandwich testing strategy.

Modified Sandwich Testing Strategy

- Test in parallel:
  - **Middle layer with drivers and stubs**
  - **Top layer with stubs**
  - **Bottom layer with drivers**
- Test in parallel:
  - **Top layer accessing middle layer (top layer replaces drivers)**
  - **Bottom accessed by middle layer (bottom layer replaces stubs)**

Steps in Integration Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Do *functional testing:* Define test cases that exercise all uses cases with the selected component
4. Do *structural testing:* Define test cases that exercise the selected component
5. Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify errors* in the (current) component configuration.

System Testing

- Unit testing e Integration testing si focalizzano sulla ricerca di bug nelle componenti individuali e nelle intefacce tra le componenti.
- Il System testing assicura che il sistema completo è conforme ai requisiti funzionali e non funzionali.
- Attività:
  - **Functional Testing.**
  - **Pilot Testing**
  - **Performance Testing.**
  - **Acceptance Testing**
  - **Installation Testing**
- Impatto dei requisiti sul testing del sistema:
  - Più espliciti sono i requisiti, più facile sono da testare.
  - La qualità degli use case influenza il Functional Testing
  - La qualità della decomposizione dei sottosistemi influenza lo Structure Testing
  - La qualità dei requisiti non funzionali e dei vincoli influenza il Performance Testing

Functional Testing

Essenzialmente il black box testing

- Goal: testare le funzionalità del sistema
- I test case sono progettati dal documento dei requisiti e si focalizza sulle richieste e le funzioni chiave (gli use case)
- Il sistema è trattato come un black box.
- I test case per le unità possono essere riusati, ma devono essere scelti quelli rilevanti per l'utente finale e che hanno una buona probabilità di riscontrare un fallimento.

Functional Testing

- Function testing finds differences between the use case model and the observed system behavior
  - **Usability testing finds differences between the use case model and the user's expectation of the system**
- Test cases should excercise both common and exceptional use cases

Example

- ***PurchaseTicket*** use case describes the steps necessary for a *Passenger* to successfully purchase a ticket.
  - ***TimeOut*, *Cancel*, *OutOfOrder*, and *NoChange* use cases describe various exceptional conditions resulting from the state of the distributor or actions by the Passenger**

Figure 11-23: use case model

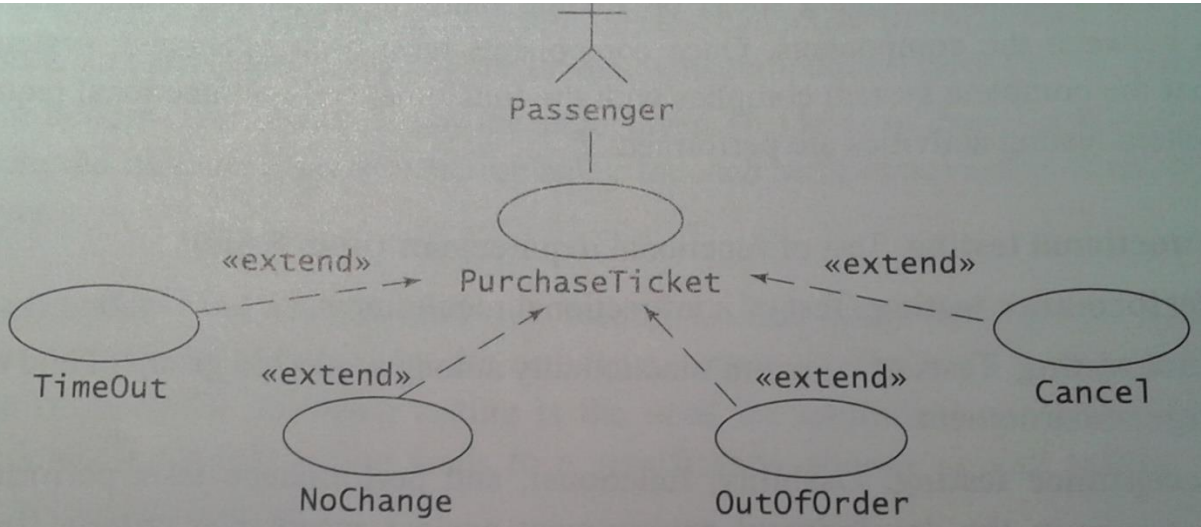


Figure 11-23 An example of use case model for a subway ticket distributor (UML use case diagram).

1. The `Passenger` may press multiple zone buttons before inserting money, in which case the `Distributor` should display the amount of the last zone.
2. The `Passenger` may select another zone button after beginning to insert money, in which case the `Distributor` should return all money inserted by the `Passenger`.
3. The `Passenger` may insert more money than needed, in which case the `Distributor` should return the correct change.

Figure 11-24: use case PurchaseTicket

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

Figure 11-24 An example of use case from the ticket distributor use case model `PurchaseTicket`

Features to be tested

- We notice that three features of the Distributor are likely to fail and should be tested:
  - **1) The *Passenger* may press multiple zone buttons before inserting money, in which case the *Distributor* should display the amount of the last zone**
  - **2) The *Passenger* may select another zone button after beginning to insert money, in which case the *Distributor* should return all money inserted by the *Passenger***
  - **3) The *Passenger* may insert more money than needed, in which case the *Distributor* should return the correct change**

Exercising the three features

- Flow of events describes both the inputs to the system (stimuli that the *Passenger* sends to the the *Distributor*) and desired outputs (correct responses from the *Distributor*)

Flow of events	1. The `Passenger` presses in succession the zone buttons 2, 4, 1, and 2. 2. The `Distributor` should display in succession $1.25, $2.25, $0.75, and $1.25. 3. The `Passenger` inserts a $5 bill. 4. The `Distributor` returns three $1 bills and three quarters and issues a 2-zone ticket. 5. The `Passenger` repeats steps 1–4 using his second $5 bill. 6. The `Passenger` repeats steps 1–3 using four quarters and three dimes. The `Distributor` issues a 2-zone ticket and returns a nickel. 7. The `Passenger` selects zone 1 and inserts a dollar bill. The `Distributor` issues a 1-zone ticket and returns a quarter. 8. The `Passenger` selects zone 4 and inserts two $1 bills and a quarter. The `Distributor` issues a 4-zone ticket. 9. The `Passenger` selects zone 4. The `Distributor` displays $2.25. The `Passenger` inserts a $1 bill and a nickel, and selects zone 2. The `Distributor` returns the $1 bill and the nickel and displays $1.25.
Exit condition	The `Passenger` has three 2-zone tickets, one 1-zone ticket, and one 4-zone ticket.

Test cases are derived for all use cases, including use cases for exceptional behavior

Figure 11-25 An example of test case derived from the Purchase ticket use ...

Performance Testing

- Stress Testing
  - **Stress limits of system (maximum # of users, extended operation)**
- Volume testing
  - **Test what happens if large amounts of data are handled**
- Configuration testing
  - **Test the various software and hardware configurations**
- Compatibility test
  - **Test backward compatibility with existing systems**
- Security testing
  - **Try to violate security requirements**
- Timing testing
  - **Evaluate response times and time to perform a function**
- Environmental test
  - **Test tolerances for heat, humidity, motion, portability**
- Quality testing
  - **Test reliability, maintainability & availability of the system**
- Recovery testing
  - **Tests system's response to presence of errors or loss of data.**
- Human factors testing
  - **Tests user interface with user**

Test Cases for Performance Testing

- **Spingere il sistema integrato verso i suoi limiti!**
- **Goal: tentare di rompere il sistema**
- **Testare come il sistema si comporta quando è sovraccarico.**
  - **Possono essere identificati colli di bottiglia? (I primi candidati ad ad essere riprogettati nella prossima iterazione)**
- **Tenta ordini di esecuzioni non usuali**
  - **Call a receive() before send()**
- **Controlla le risposte del sistema a grandi volumi di dati**
  - **Se si è supposto che il sistema debba gestire 1000 item, provalo con 1001 item.**

Pilot testing

- Il sistema è installato e usato da un insieme di utenti selezionati
- Nessuna linea guida o scenario è fornito agli utenti
- Sistemi pilota sono utili quando un sistema è costruito senza un insieme di richieste specifiche, o senza un particolare cliente in mente
- **Alpha test**. È un test pilota con utenti che esercitano il sistema **nell'ambiente di sviluppo**
- **Beta test**. Il test di accettazione è realizzato da un numero limitato di utenti **nell'ambiente di utilizzo**
  - **Il nuovo paradigma ampiamente utilizzato con la distribuzione del software tramite Internet**
  - **Offre il software a chiunque che è interessato a testarlo**
  - **Gli utenti sono caricati del testing del software!**

Acceptance testing

- Tre modi in cui il cliente può valutare un sistema durante l'acceptance testing:
  - **Benchmark test. Il cliente prepara un insieme di test case che rappresentano le condizioni tipiche sotto cui il sistema dovrà operare**
  - **Competitor testing. Il nuovo sistema è testato contro un sistema esistente o un prodotto concorrente (in reengineering project)**
  - **Shadow testing. Una forma di testing a confronto, il nuovo sistema e il sistema legacy sono eseguiti in parallelo e i loro output sono confrontati**

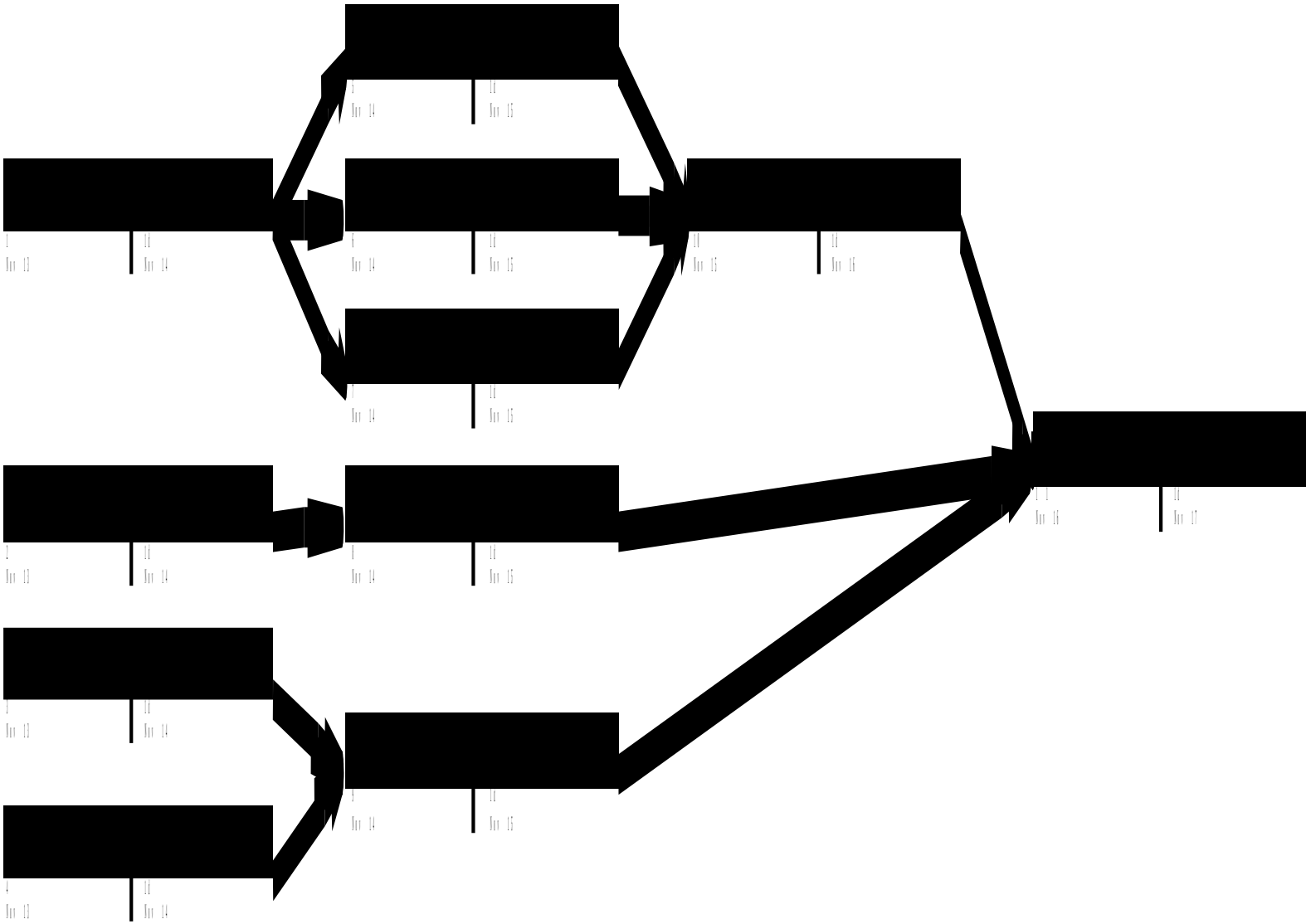
- Se il cliente è soddisfatto, il sistema è accettato, eventualmente con una lista di cambiamenti da effettuare

Installation testing

- Dopo che il sistema è accettato, esso è installato nell'ambiente di utilizzo.
- Il sistema installato deve soddisfare in pieno le richieste del cliente
- In molti casi il test di installazione ripete i test case eseguiti durante il functional testing e il performance testing nell'ambiente di utilizzo
- Quando il cliente è soddisfatto, il sistema viene formalmente rilasciato, ed è pronto per l'uso

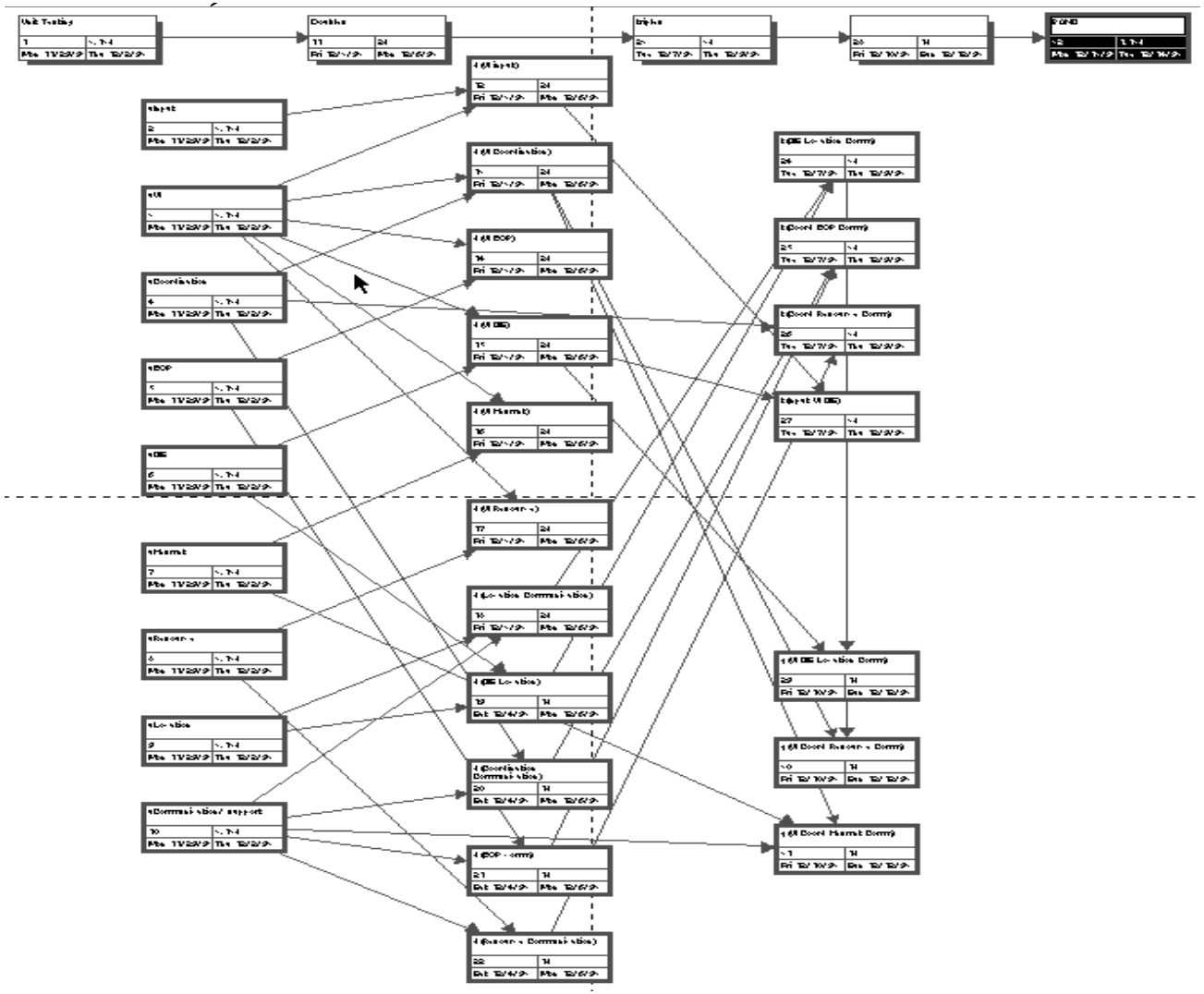
Managing testing

- Come gestire le testing activity per minimizzare le risorse necessarie
  - Alla fine, gli sviluppatori dovrebbero rilevare e quindi riparare un numero sufficiente di bug tale che il sistema soddisfi i requisiti funzionali e non funzionali entro un limite accettabile da parte del cliente
- Pianificazione delle testing activity (diagramma di PERT che mostra le dipendenze)
- Le attività sono documentate in 4 tipi di documenti:
  - il **Test Plan** che si focalizza sugli aspetti manageriali;
  - Ogni test è documentato attraverso un **Test Case Specification**;
  - Ogni esecuzione di un test case è documentata attraverso un **Test Incident Report**;
  - Il **Test Report Summary** elenca tutti i fallimenti rilevati durante i test che devono essere investigati
- Ruoli assegnati durante il testing



Example of a PERT chart for a schedule of the sandwich tests

Esempio di un diagramma di PERT (Scheduling Sandwich Tests)



Unit Tests **Double Tests** **Triple Tests** **SystemTests**

Documenting testing: Planning

- ***Test Plan***: focuses on the managerial aspects of testing.
  - **It documents the scope, the approach, resources, schedule of testing activities.**
  - **The requirements and the components to be tested are identified in this document**
- ***Test Case Specification***: documents each test
  - **This document contains the inputs, drivers, stubs, and expected outputs of the tests, as well as the tasks to be performed**

Test plan

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

Test Case Specification

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

Documenting testing: Execution Documents

- ***Test Execution Report:*** documents each execution
- Includes:
  - ***Test item transmittal report*: accompanying document for a tested software item**
    - **Contains at least the information to identify the software item**
  - ***Test log*: The actual results of the tests and differences from expected output are recorded**
  - ***Test Incident Report*: describes all incidents discovered during the tests that need to be investigated (including the failures)**
    - **From this document the developers analyze and prioritize each failure and plan for changes in the system and in the models.**
    - **These changes can trigger new test cases and new test executions**
  - ***Test Summary Report*: a summary of the testing activity.**
    - **Lists the incidents that have been solved and the incidents that have not been solved**
    - **Evaluates the adopted testing techniques, by describing their limitations**

Testing has its own Life Cycle

Establish the test objectives

Design the test cases

Write the test cases

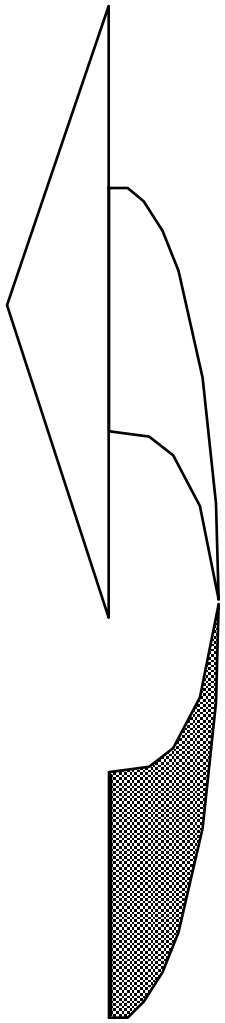
Test the test cases

Execute the tests

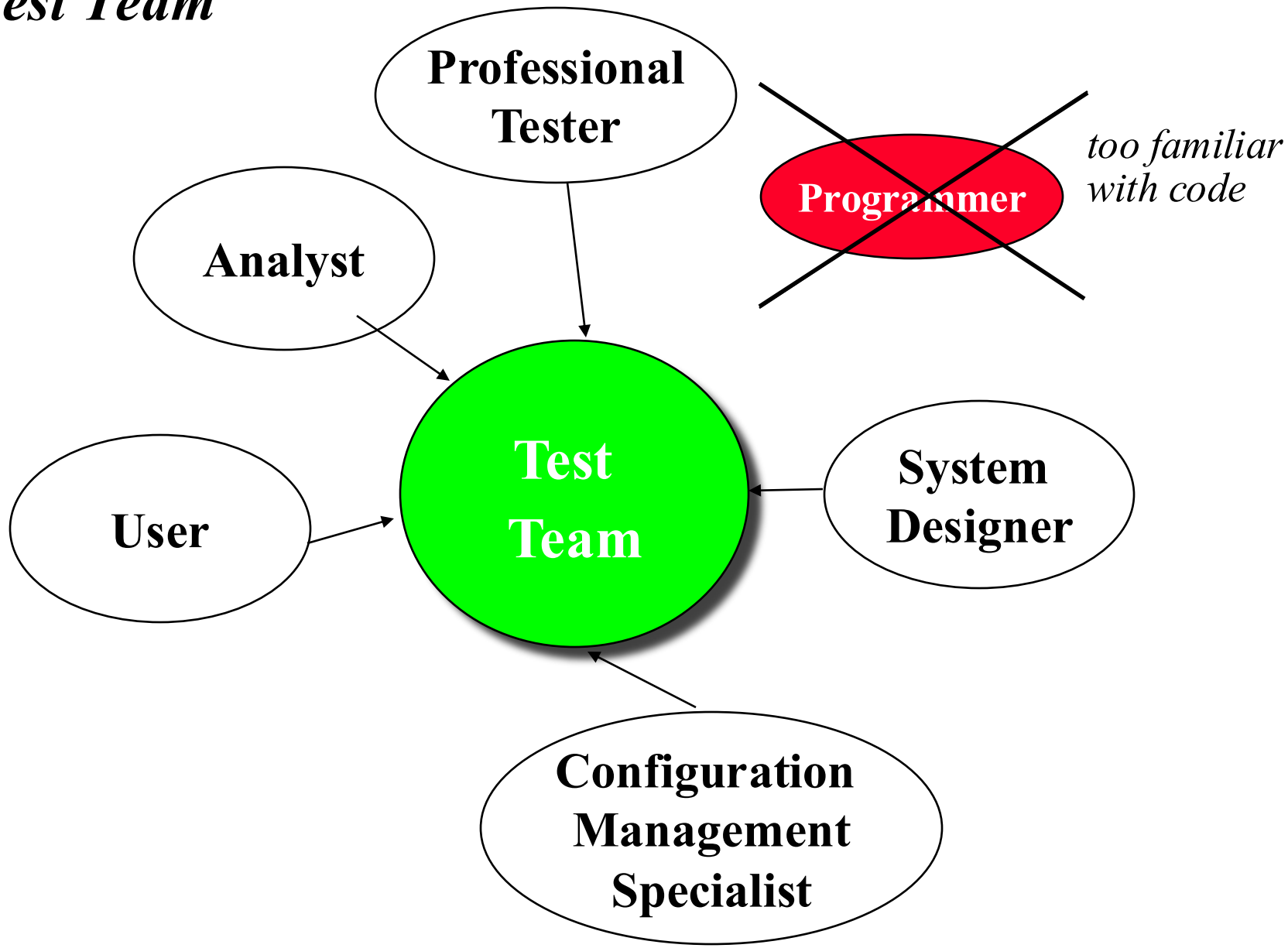
Evaluate the test results

Change the system

Do regression testing



Test Team


Professional Tester
Programmer
too familiar with code
Analyst
Test Team
User
System Designer
Configuration Management Specialist