



Defect testing

- Testing programs to establish the presence of system defects



Objectives

- To understand **testing techniques** that are geared to discover program faults
- To introduce guidelines for **interface testing**
- To understand specific approaches to **object-oriented testing**
- To understand the principles of **CASE tool support** for testing





Topics covered

- Defect testing
- Integration testing
- Object-oriented testing
- Testing workbenches

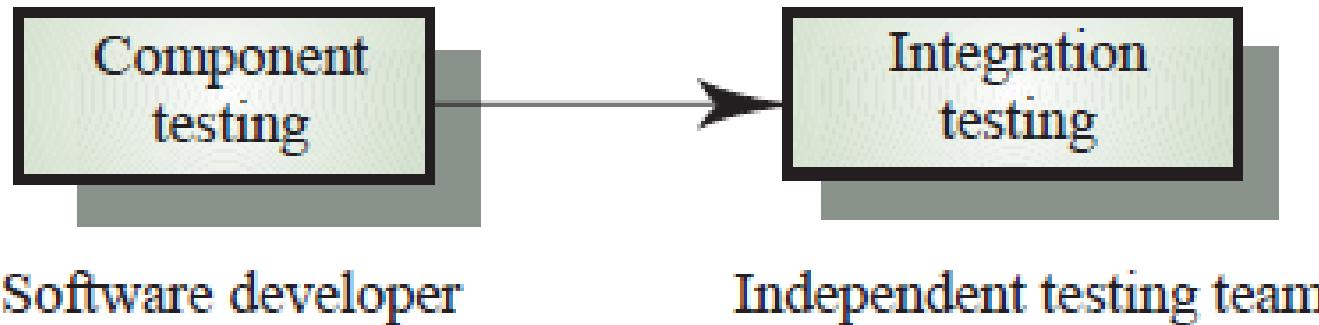


The testing process

- Component testing
 - Testing of individual program components
 - Usually the responsibility of the component developer (except sometimes for critical systems)
 - Tests are derived from the developer's experience
- Integration testing
 - Testing of groups of components integrated to create a system or sub-system
 - The responsibility of an independent testing team
 - Tests are based on a system specification



Testing phases



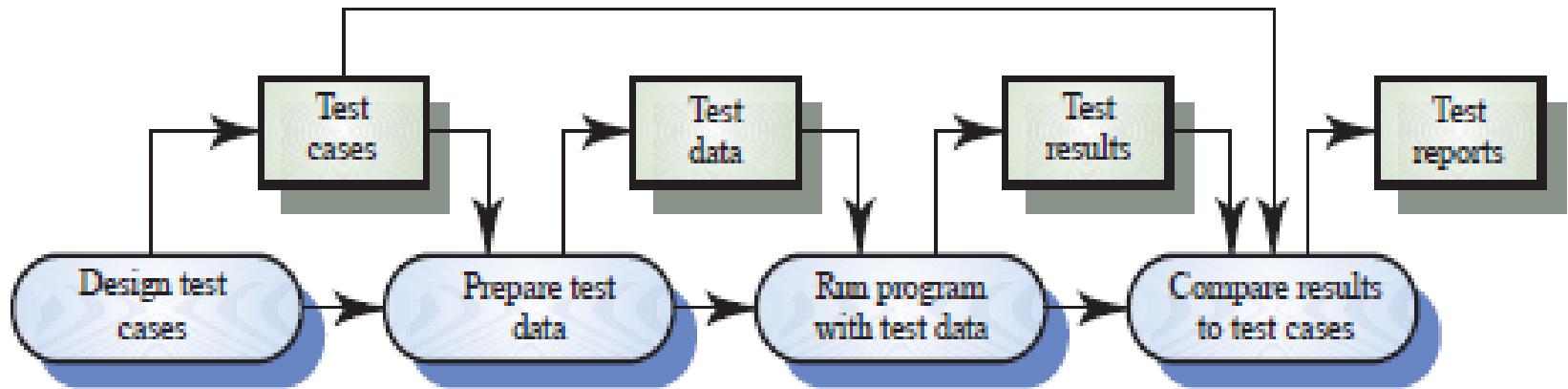


Defect testing

- 결함 테스팅(defect testing)의 목표는 프로그램에 있는 결함들을 발견하는 것임.
- 성공적인 결함 테스트는 프로그램이 비정상적인(anomalous) 방식으로 동작하도록 하는 것.
- 테스트는 결함의 부재(absence)가 아니라 결함의 존재를 보이는 것이다.



The defect testing process



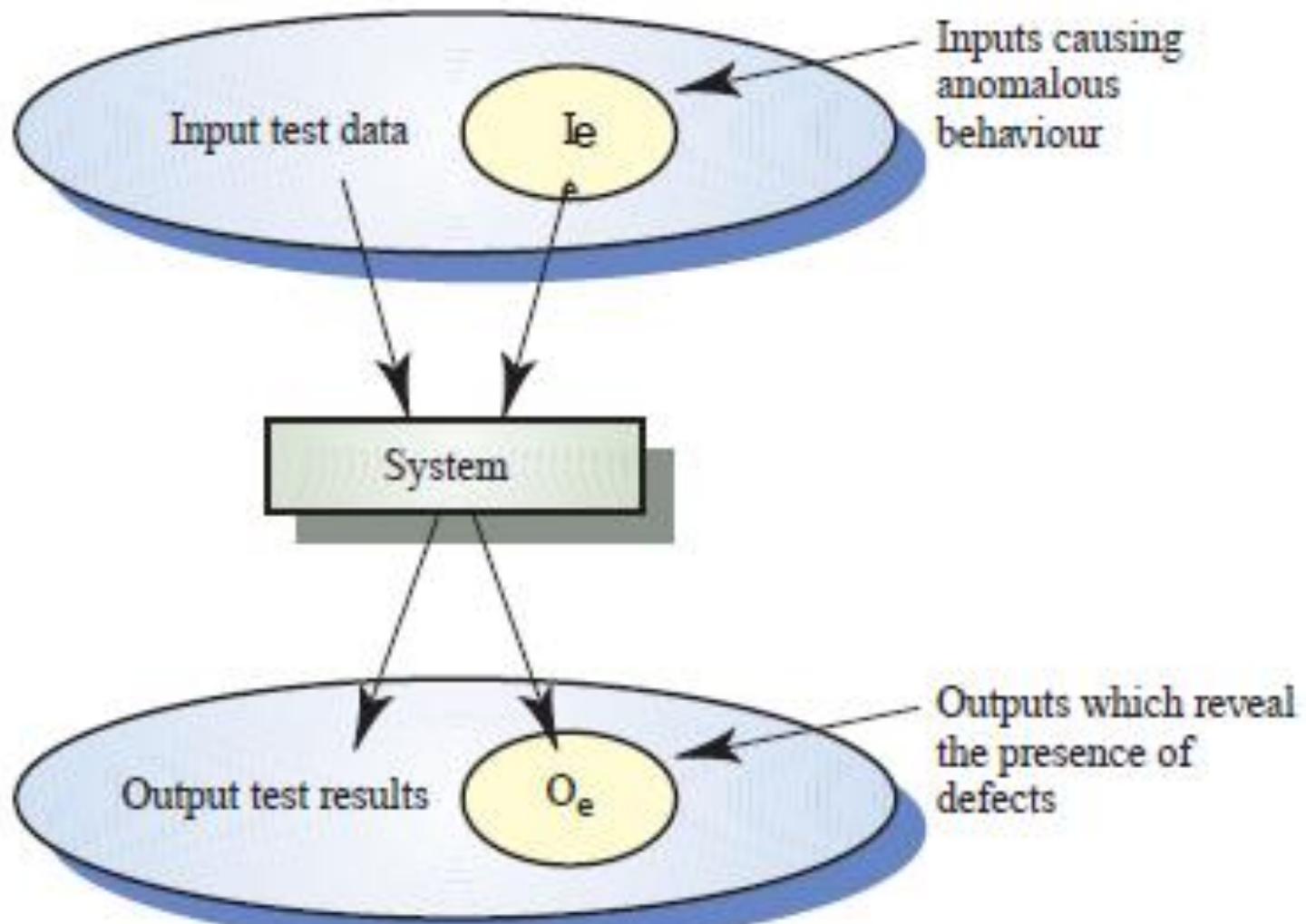
Black-box testing

- 프로그램을 'black-box'로 생각하고 테스팅을 수행하는 방법.
- The program test cases are based on the system specification
- Test planning can begin early in the software process





Black-box testing



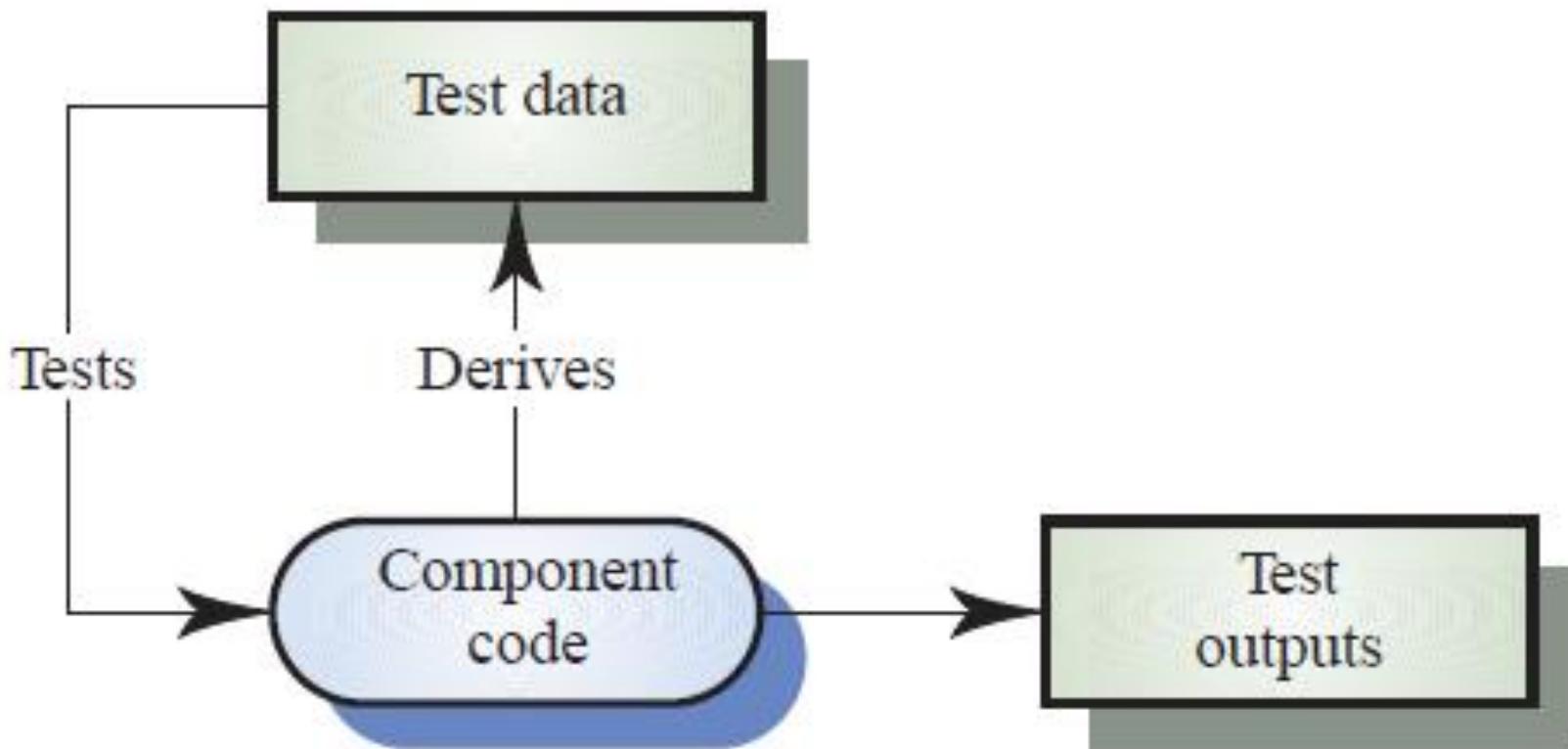
Structural testing

- Sometime called white-box testing
- 프로그램의 구조에 따라서 테스트 케이스들을 유도함. 부가적인 테스트 케이스들을 식별하기 위해 프로그램에 대한 지식이 사용됨.
- Objective is to exercise all program statements
(not all path combinations)





White-box testing



Path testing

- path testing: 프로그램에 있는 모든 경로가 적어도 한번은 실행되도록 테스트 케이스들의 집합을 구성하는 것.
- path testing의 시작점은 프로그램 판단(decision)들을 나타내는 노드와 제어 흐름을 나타내는 간선(arcs)들을 보여주는 프로그램 흐름 그래프임.
- Statements with conditions are therefore nodes in the flow graph





Program flow graphs

- 프로그램 제어 흐름을 보여줌. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node
- Used as a basis for computing the **cyclomatic complexity**
- Cyclomatic complexity = Number of edges - Number of nodes +2

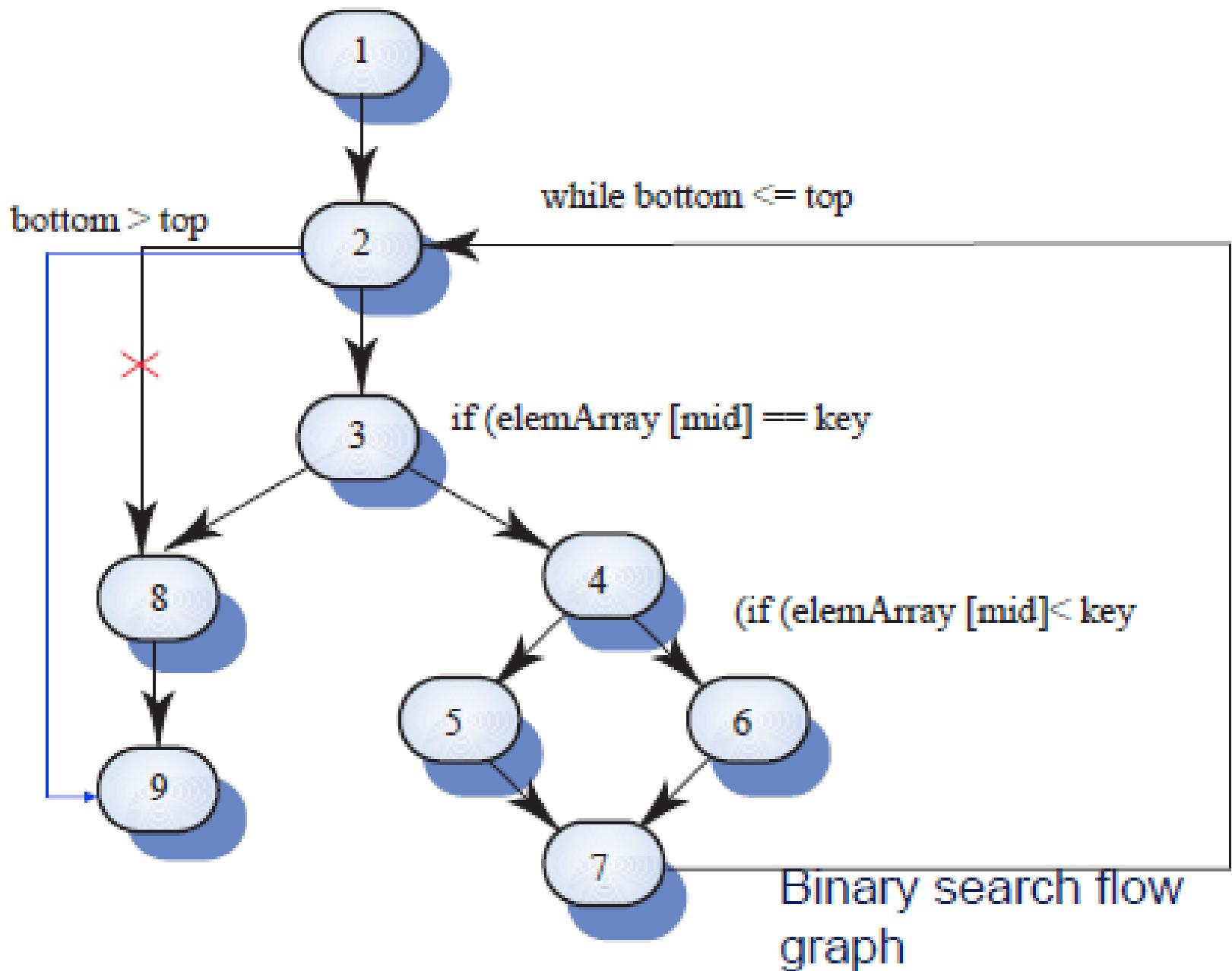




Cyclomatic complexity

- 모든 제어 문장들을 시험하기 위한 테스트들의 수는 cyclomatic complexity와 같다.
- Cyclomatic complexity는 프로그램에 있는 조건문의 수 + 1 이다.
- Useful if used with care. Does not imply adequacy of testing.
- Although all paths are executed, all combinations of paths are not executed







Independent paths

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9
- 테스트 케이스들은 이런 모든 경로들이 실행될 수 있도록 유도되어야 한다.
- 동적 프로그램 분석기가 경로들이 시험되었는지 검사하기 위해 사용될 수 있다.

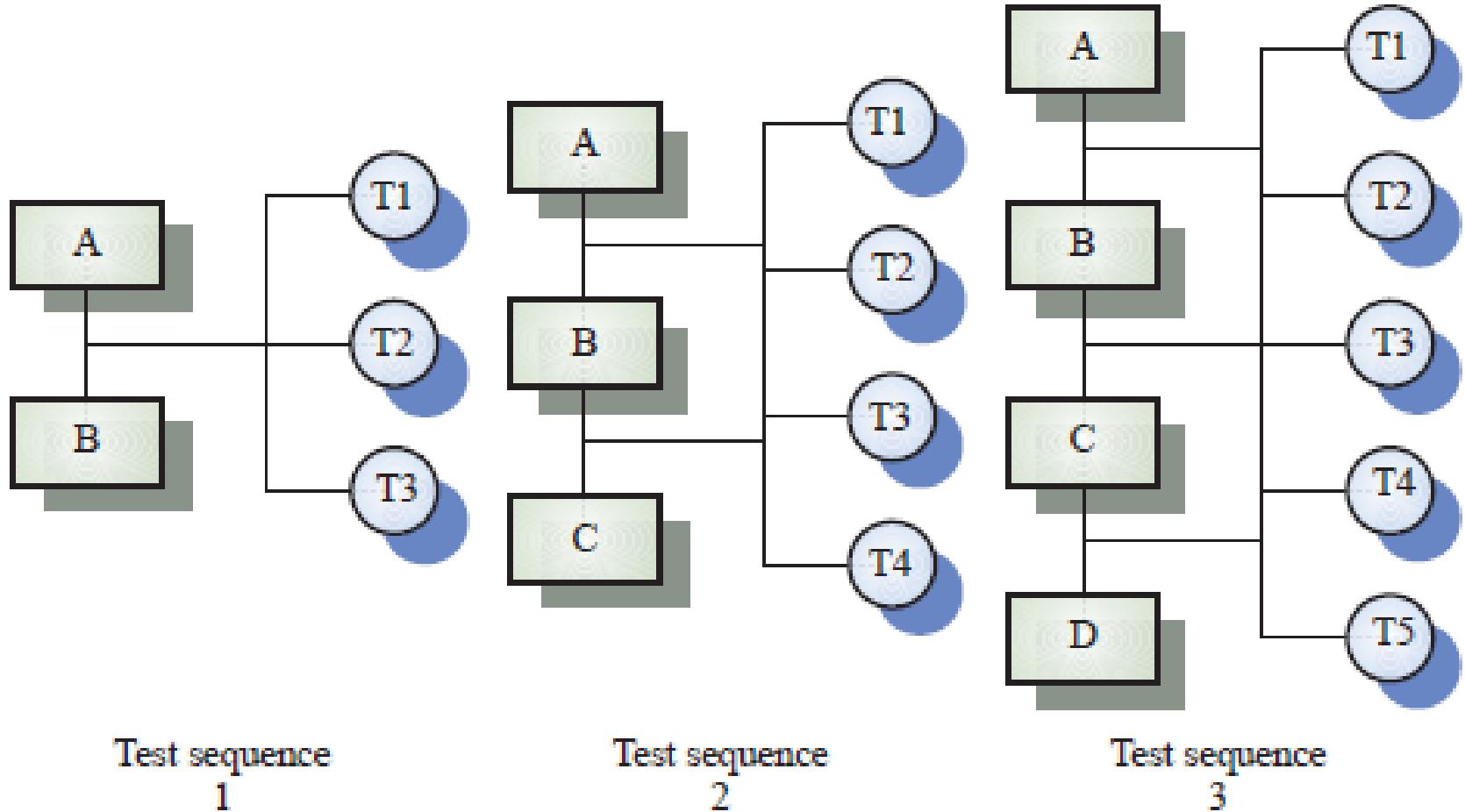


Integration testing

- Component들이 통합되어 구성된 전체 시스템 혹은 서브시스템들을 시험하는 것.
- 통합 테스팅은 테스트들이 명세서로부터 유도된 **black-box** 테스팅이어야 함
- 주요한 어려움은 에러들을 지역화(localization)하는 것임.
- 점진적인 통합 테스팅이 이 문제를 줄여줌.

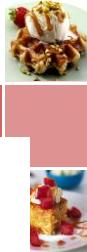


Incremental integration testing

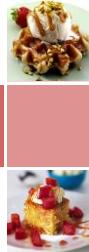
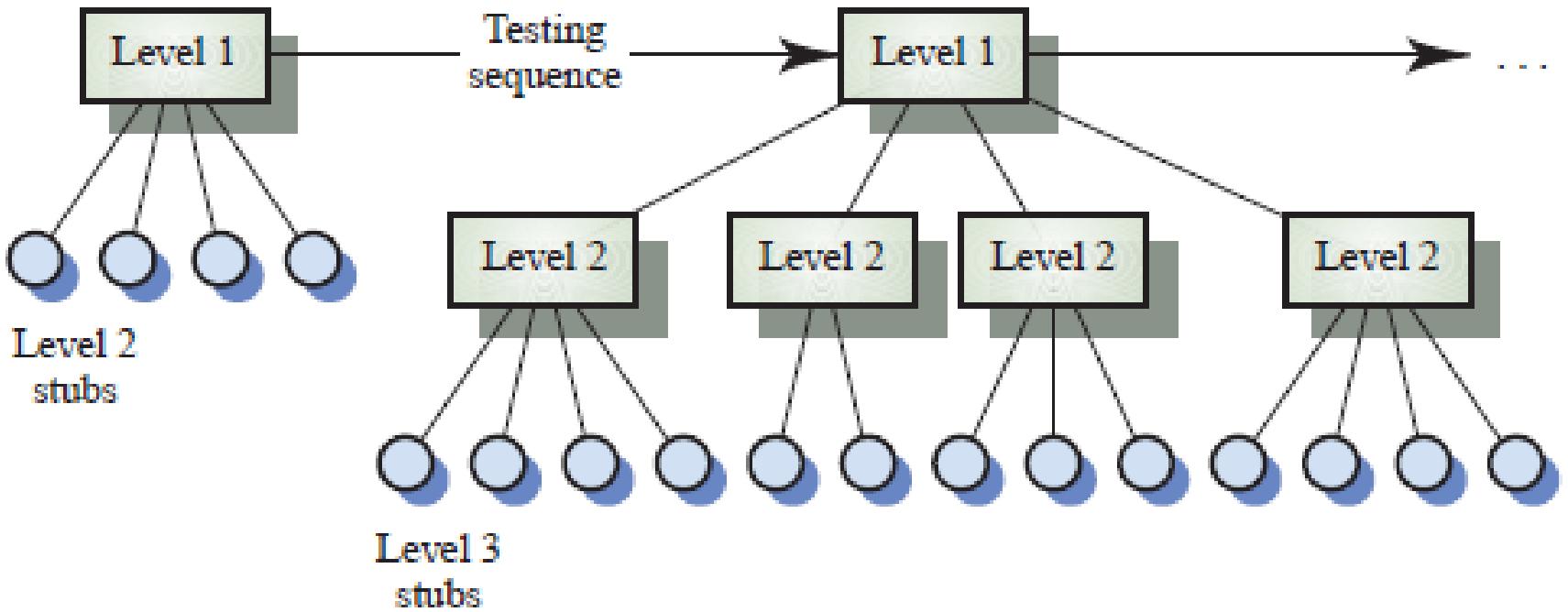


Approaches to integration testing

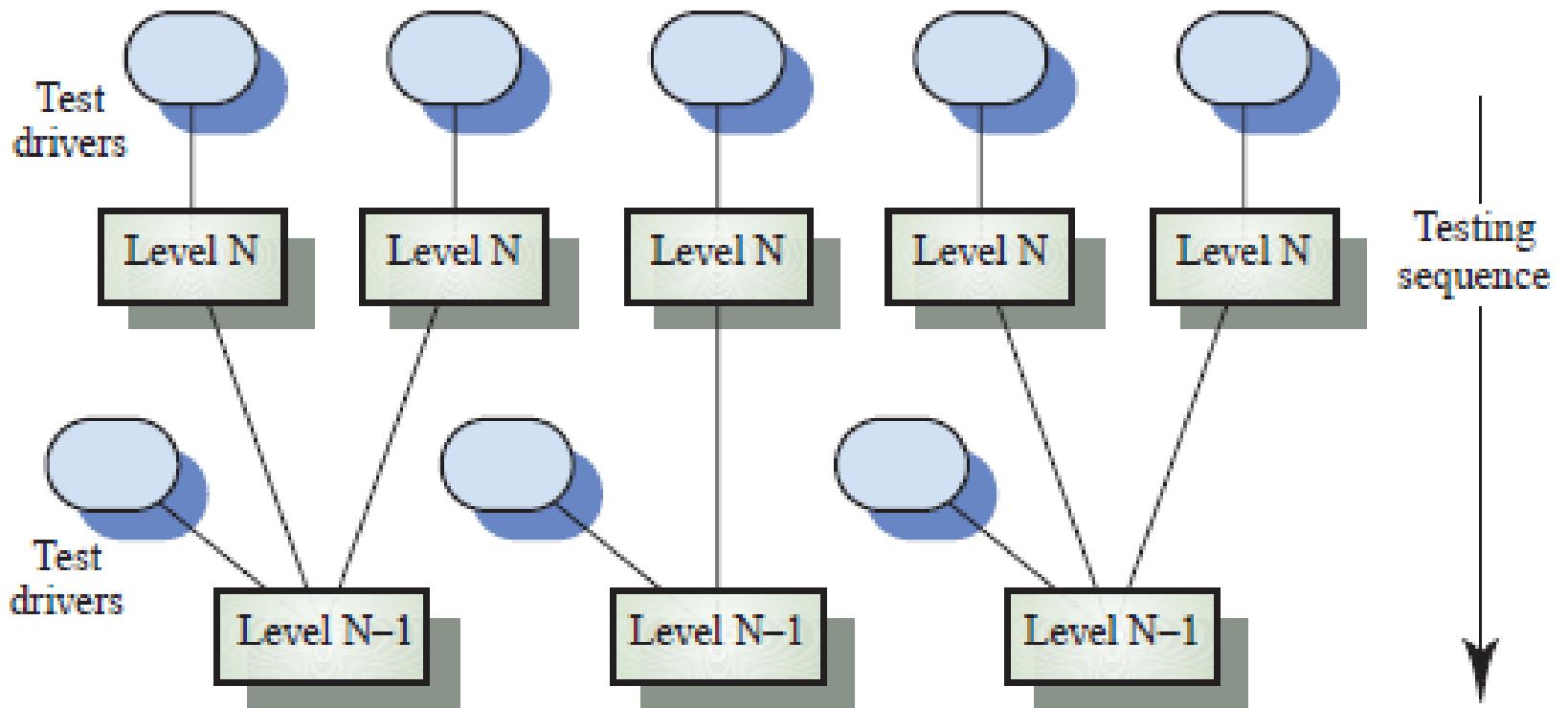
- Top-down testing
 - Start with high-level system and integrate from the top-down replacing individual components by **stubs** where appropriate
- Bottom-up testing
 - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies



Top-down testing



Bottom-up testing





Testing approaches

- 구조 확인(Architectural validation)
Top-down integration testing is better at discovering errors in the system architecture
- 시스템 증명(System demonstration)
 - Top-down integration testing allows a limited demonstration at an early stage in the development
- 테스트 구현(Test implementation)
 - Often easier with bottom-up integration testing
- 테스트 관측(Test observation)
 - Problems with both approaches. Extra code may be required to observe tests



Interface testing

- 모듈 또는 서브시스템들이 보다 큰 시스템을 생성하기 위해 통합될 때 수행됨.
- 목적: 인터페이스 여러 또는 인터페이스들에 대한 개별적인 가정으로 인한 결함을 검출하기 위해서.
- 객체들이 각자의 인터페이스들에 의해 정의되는 객체-지향 개발에서는 특히 중요함.





Interface testing guidelines

- 극단적인 값을 갖는 매개변수들을 가지고 프로시저들을 호출할 수 있도록 테스트를 설계하라.
- 항상 null 포인터를 가지고 포인터 매개변수들을 시험하여라.
- 컴포넌트들이 잘못 동작하도록 테스트를 설계하여라.
- 메시지 전달 시스템에서 스트레스(stress) 테스팅을 사용하여라.
- 공유 메모리 시스템에서는 컴포넌트들이 동작하는 순서가 변하도록 하여라.



Stress testing

- Exercises the system beyond its maximum design load. 시스템에 과부하를 주는 것은 종종 결함들이 드러나도록 한다.
- 시스템에 과부하를 주는 것은 장애시의 동작을 시험하는 것이다. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data
- 분산 시스템에 특별히 적절함.
분산 시스템은 네트워크에 과부하가 걸리면 급격한 성능 저하 현상을 보일 수 있기 때문에.



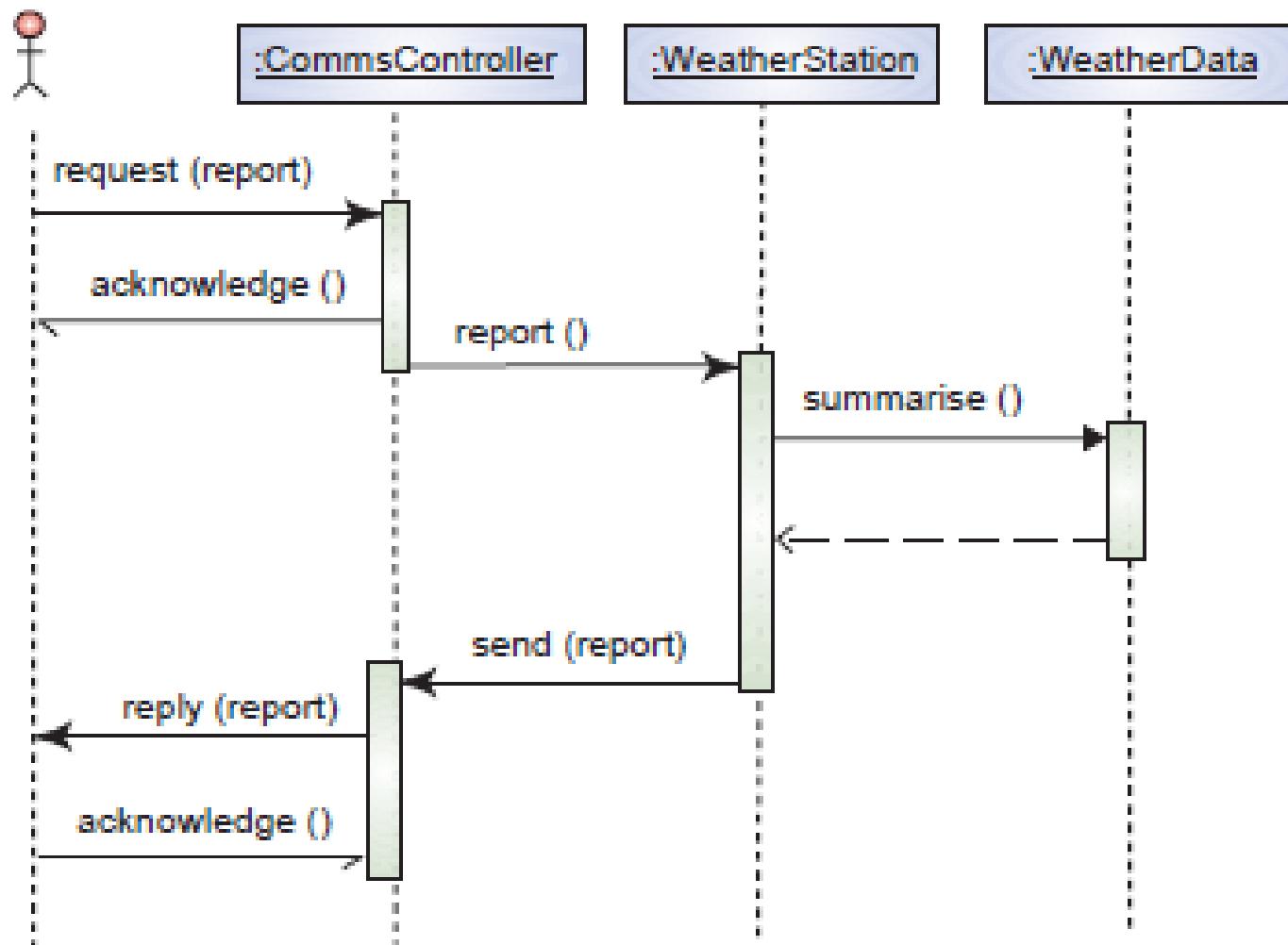


Scenario-based testing

- 유즈-케이스로부터 시나리오를 식별. 이것을 시나리오에 관련된 객체들을 보여주는 interaction 다이아그램으로 보충함.
- Consider the scenario in the weather station system where a report is generated



Collect weather data





Weather station testing

- Thread of methods executed
 - CommsController:request → WeatherStation:report → WeatherData:summarise
- Inputs and outputs
 - report 요청의 입력(관련된 acknowledge 포함) 그리고 report의 최종 출력
 - raw data를 생성하고 그것이 적절하게 summarise 되는가를 테스트함.
 - Use the same raw data to test the WeatherData object





Key points

- Test parts of a system which are **commonly used** rather than those which are **rarely executed**
- **Equivalence partitions** are sets of test cases where the program should behave in an equivalent way
- **Black-box testing** is based on the system specification
- **Structural testing** identifies test cases which cause all paths through the program to be executed

--- --- ---

Key points

- **Test coverage measures** ensure that all statements have been executed at least once.
- **Interface defects** arise because of specification misreading, misunderstanding, errors or invalid timing assumptions
- To test **object classes**, test all operations, attributes and states
- Integrate object-oriented systems around **clusters** of objects

