



# Software Testing Technique

IT대학 컴퓨터학부 이우진  
(woojin@knu.ac.kr)

# Topics covered

---



- ✧ Introduction of Software Safety
- ✧ Software Testing
  - Development Testing (Unit → Component → System)
  - Release Testing
  - User Testing
- ✧ Test Case Generation (black-box, white-box)
- ✧ Test-driven development (TDD)
- ✧ Embedded Software Testing

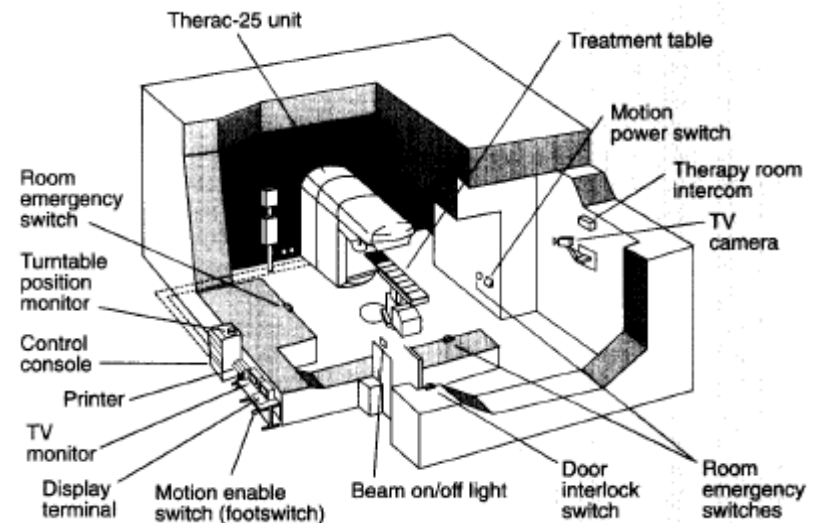
# Safety Issue in SW

- Functional Safety

- Functional safety is the part of the overall safety of a system or piece of equipment, including the safe managements of likely operator errors, hardware failures and environmental changes
- The objective of Functional Safety is freedom from unacceptable risk of physical injury or of damage to the health of people

- Therac-25 [Lev93]

- A computerized radiation therapy machine
- Six known accidents involved massive overdoses between June 1985 and Jan. 1987



# Therac-25의 사고 원인

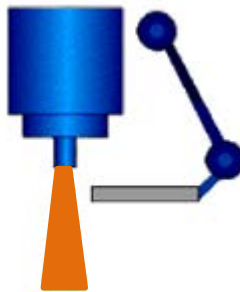


## ✧ Therac-25의 두 가지 동작모드

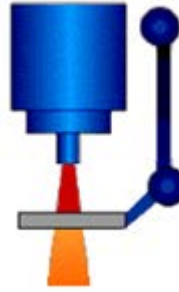
- X-ray 모드 : 텅스텐 스크린 막을 통과한 약한 방사능으로 얇은 곳에 있는 종양 치료에 사용
- Electron 모드 : 피부 깊숙이 있는 종양 치료에 사용

## ✧ 사고 시나리오

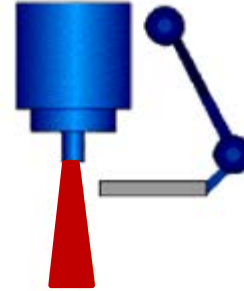
- X-ray 모드에서 방사능 수치 설정
- 오퍼레이터가 모드가 잘못된 것을 알고 급하게 **Electron** 모드로 변경 후에 치료 시작
- X-ray 모드에서 설정된 강력한 방사능이 환자에게 그대로 노출



(a) Electron 모드



(b) X-ray 모드



(c) 문제 상황

# Therac-25의 의료사고의 시사점



## ✧ 프로그래머 한 명이 제어 소프트웨어 개발

- 한 명의 프로그래머가 제어 소프트웨어 작성
- 개발 후에 어떠한 소프트웨어 검증도 거치지 않음

## ✧ 최초 사고 후 19개월 동안 그대로 사용

- 최초 사고 후에 여러 차례 시스템을 검증하였으나 사고원인을 찾지 못함
- 19개월 동안 6건의 사고 후에서야 사고 시나리오 재현 성공
- 임베디드 소프트웨어의 경우, 오류 시나리오를 찾기 어려움

- ▶ 임베디드 소프트웨어는 반드시 엄격한 안전성 검증을 거쳐야 함

# NSI 방사선 의료 사고



## ✧ National Cancer Institute, Panama City

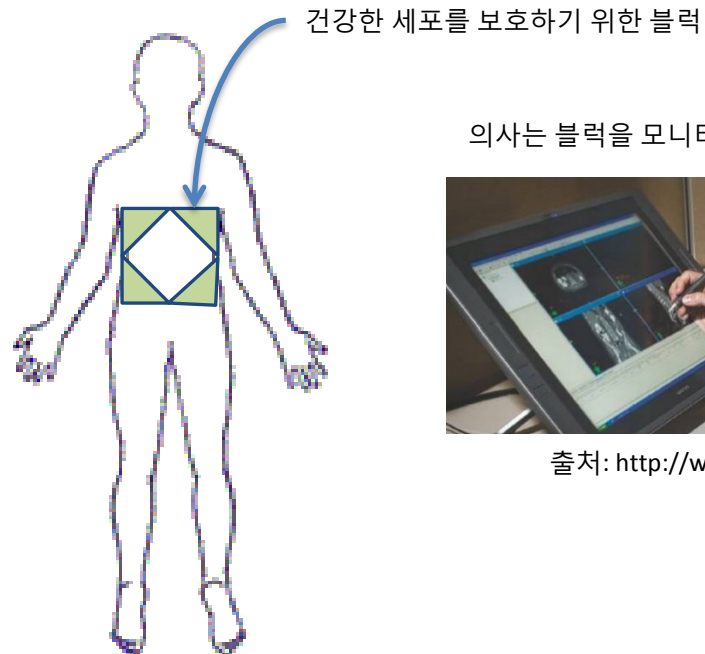
- Multidata Systems International, U.S. firm

## ✧ 2000년 28명의 환자가 방사선에 과다 노출

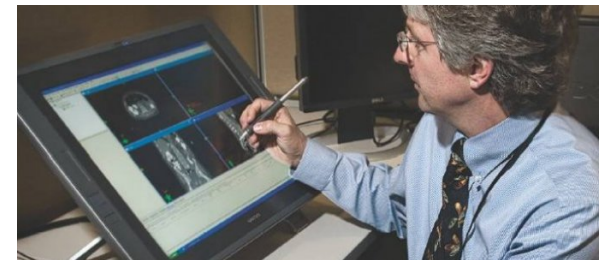
- 23명의 환자가 사망



출처: <http://www.hyperbaricoxygenchamber.com/>

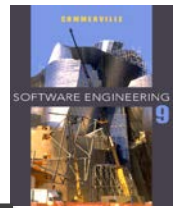


의사는 블럭을 모니터에 그려서 컴퓨터에 입력



출처: <http://www.wacom.com.sg>

# NSI 사고의 원인



출처: Investigation of an Accidental Exposure of Radiotherapy Patients in Panama, International Atomic Energy Agency, 2001

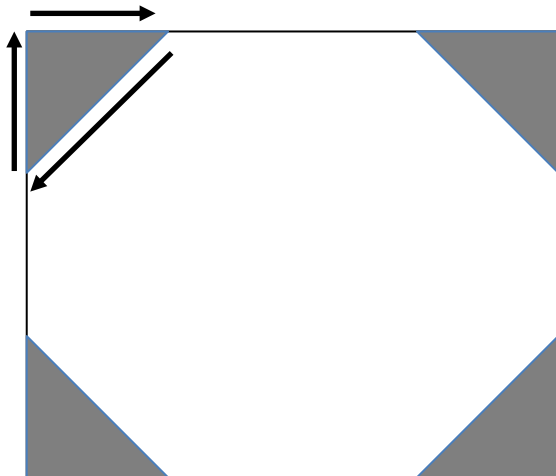
## ✧ 방사선량 계산 소프트웨어의 오류

- 예상치 못한 파라미터 값이 입력

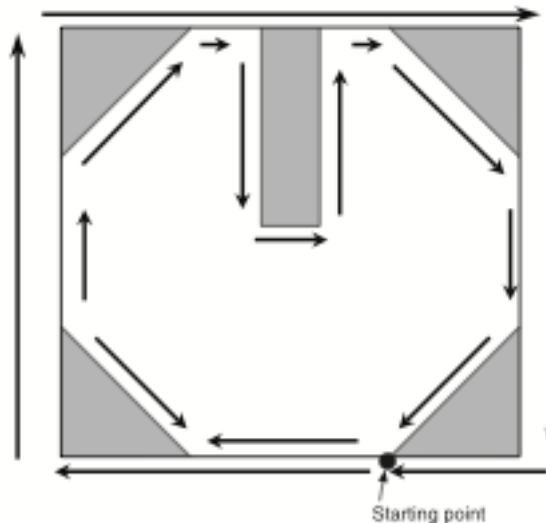
## ✧ 블록을 그리는 방식이 문제

- 네 개의 블록이 허용
- 다섯 개의 블록을 사용하길 원하였음
- 두 개의 루프를 이용하여 블록들을 도넛 형태로 그림
- 두 개의 루프를 이용할 때 그리는 방향에 따라 다른 행위를 보임

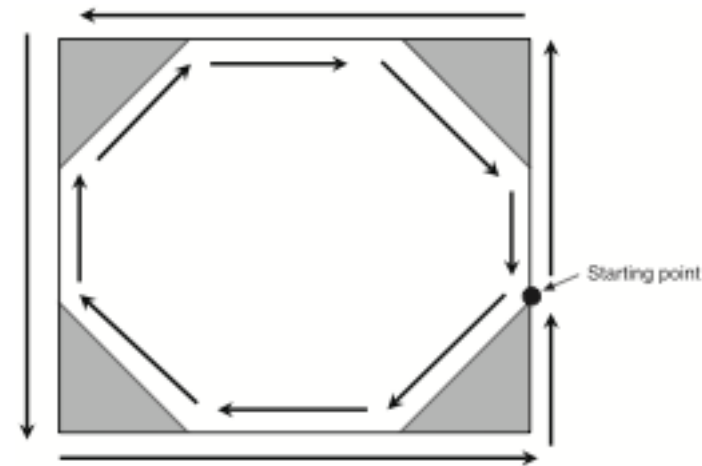
정상적인 방법



루프를 이용한 다섯 개의 블록을 사용



두 개의 루프를 다른 방향으로 그리면  
오류 발생



# Mars Climate Orbiter 사고

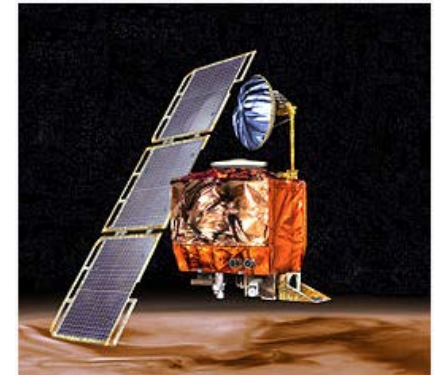
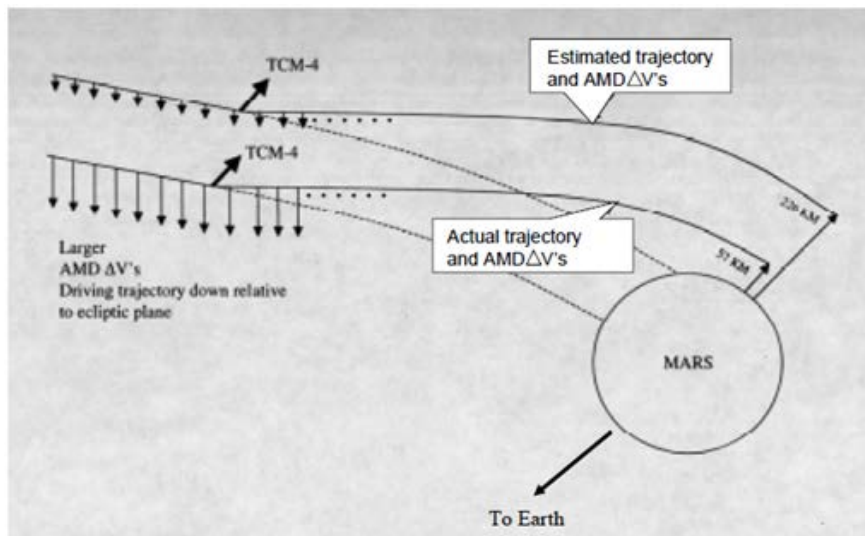


## ✧ NASA's Mars Climate Orbiter

- 338 kg Robotic Space Probe (launched on 1998. Dec. 11)
- After 416 M miles in 41 Weeks, it is lost (1999. 09)

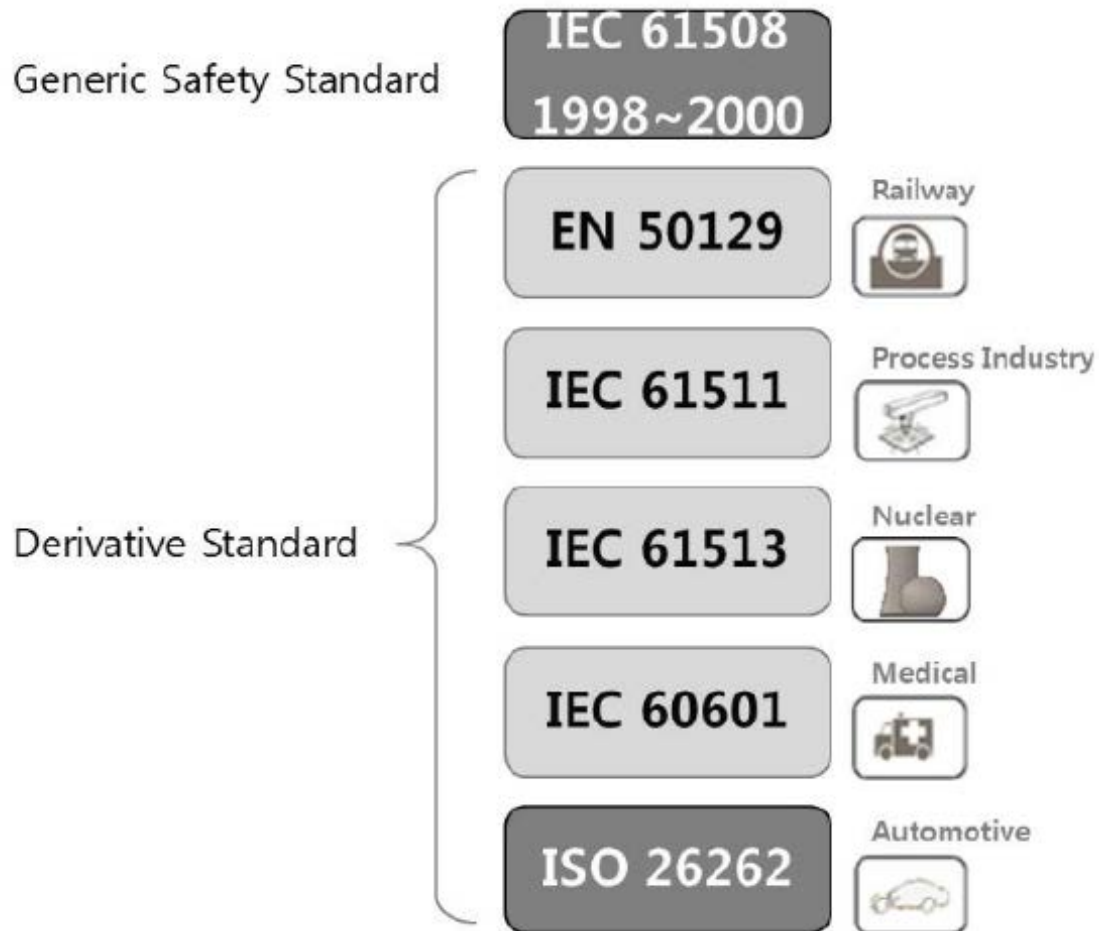
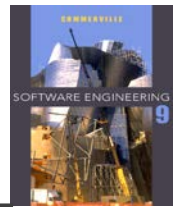
## ✧ 사고 원인

- NASA와 Lockheed사와의 communication
  - Newton Metric Unit → Pound Unit

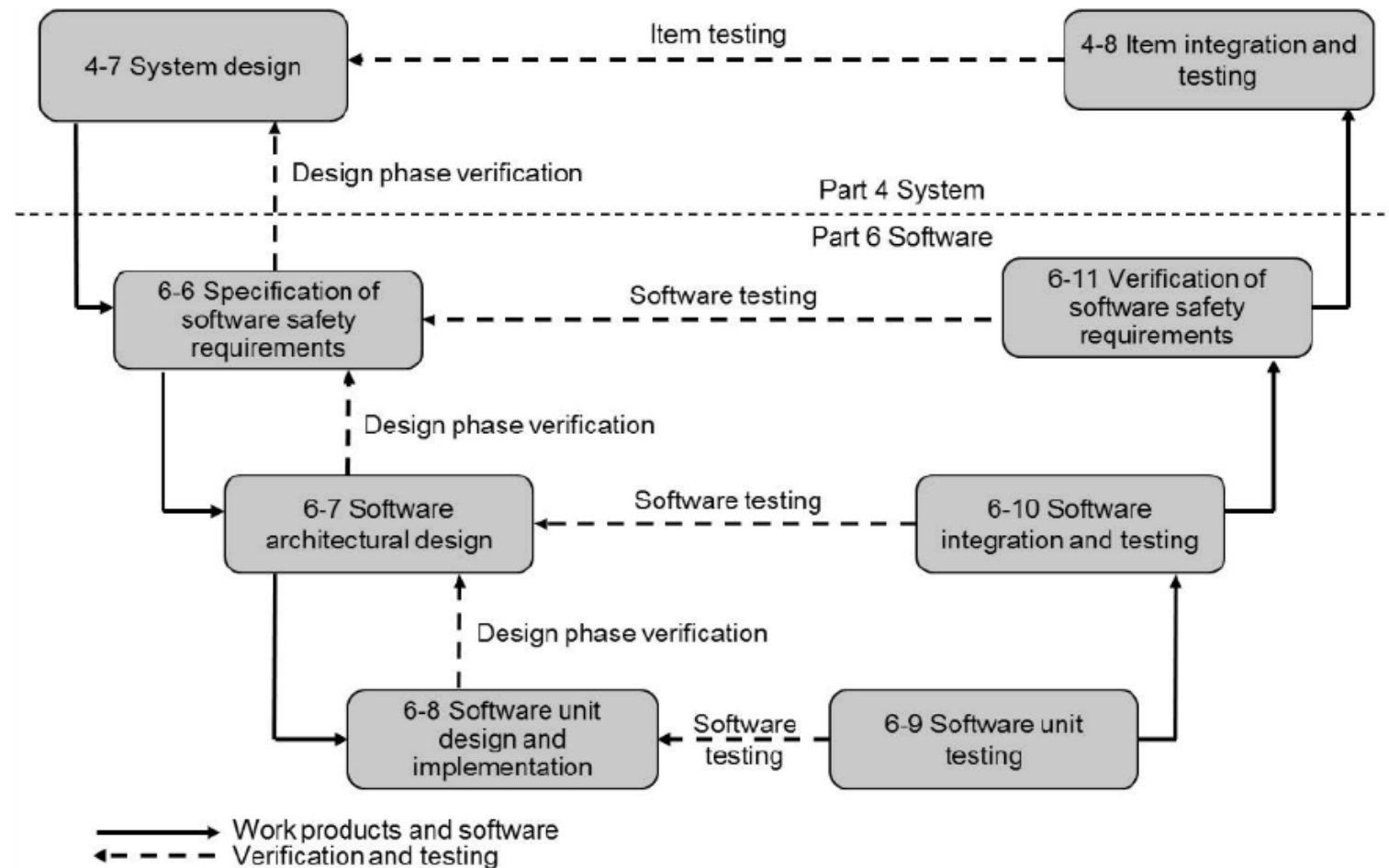




# Functional Safety Standards



# Software Testing in ISO 26262



# Program Testing

# Program testing

---



- ✧ Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- ✧ When you test software, you execute a program using artificial data.
- ✧ You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- ✧ Can reveal the presence of errors NOT their absence.
- ✧ Testing is part of a more general verification and validation process, which also includes static validation techniques.

# Verification vs validation

---



## ✧ Verification:

"Are we building the product right".

- The software should conform to its specification.

## ✧ Validation:

"Are we building the right product".

- The software should do what the user really requires.

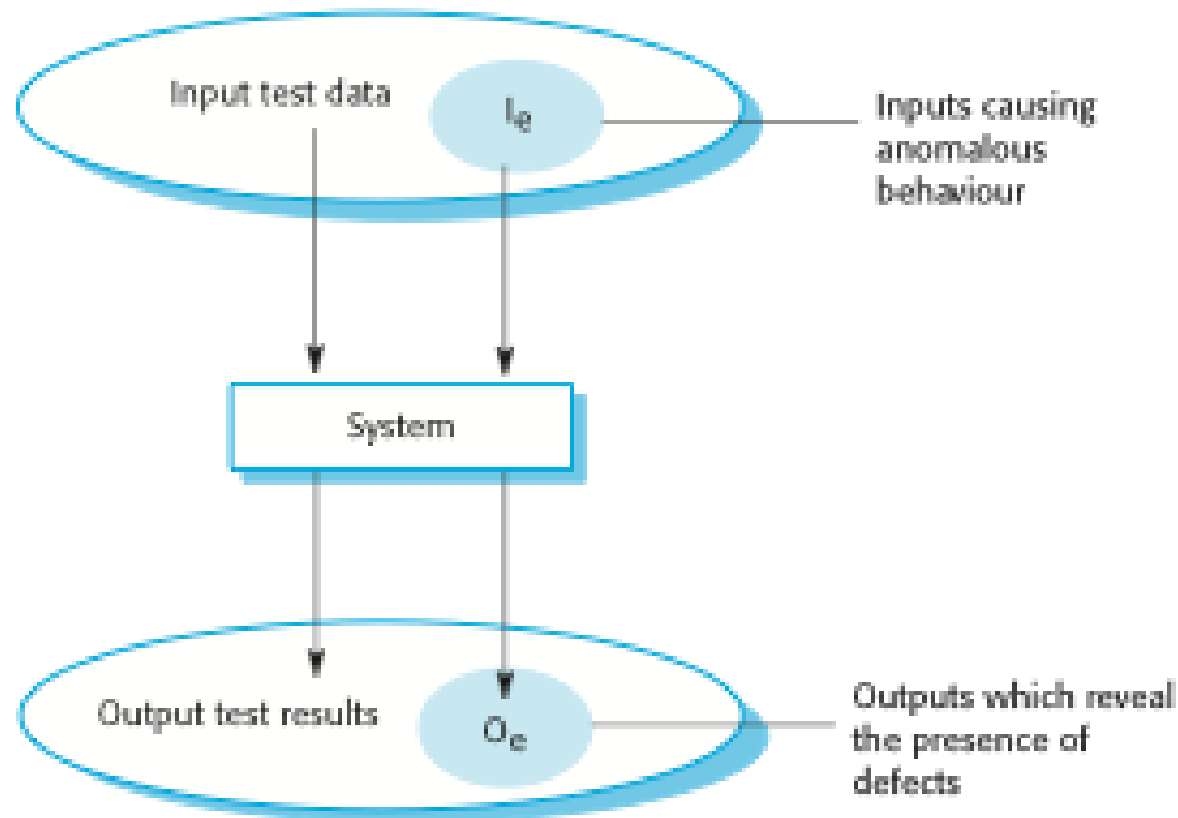
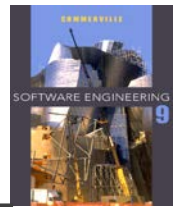
# Program testing Goals

---



- ✧ To demonstrate to the developer and the customer that the software meets its requirements.
  - The first goal leads to **validation testing**
  - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
  
- ✧ To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - The second goal leads to **defect testing**
  - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

# An input-output model of program testing



# Inspections and testing

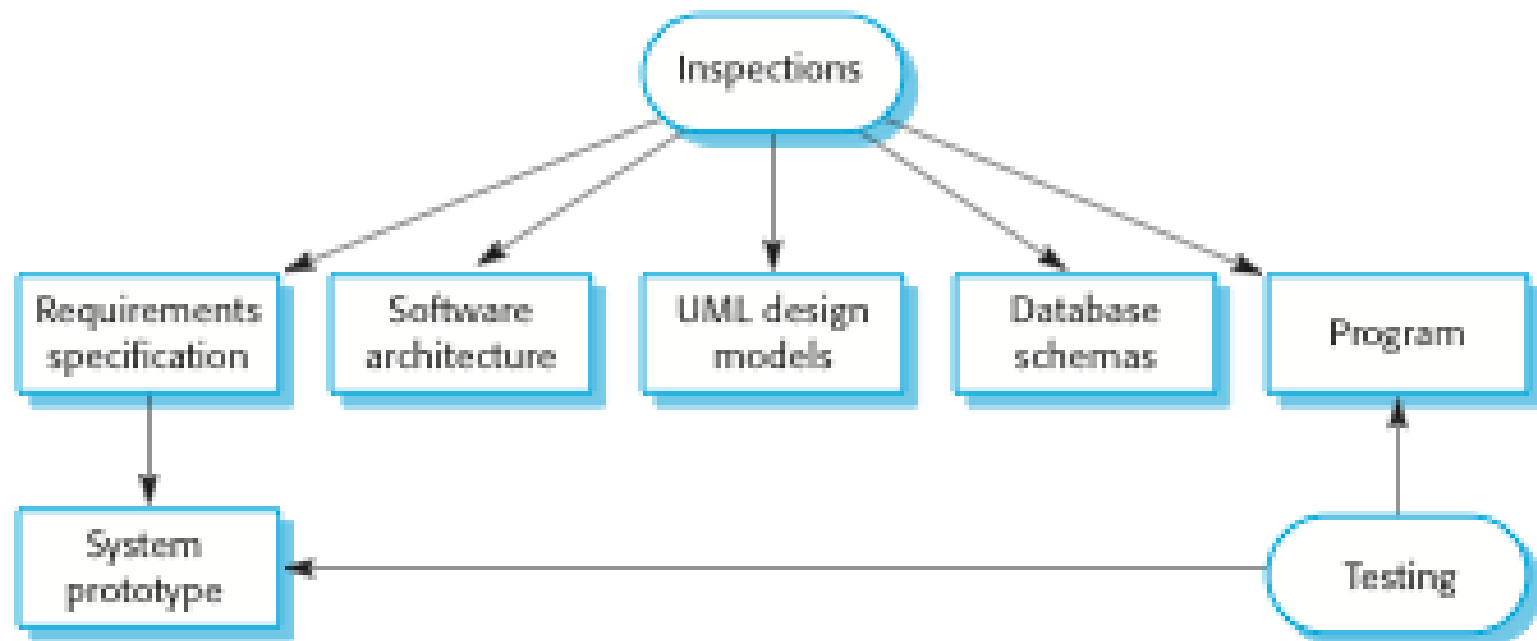
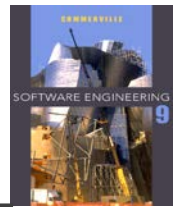
---



- ✧ **Software inspections** Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplemented by tool-based document and code analysis.
  
- ✧ **Software testing** Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.



# Inspections and testing



# Advantages of inspections

---



- ✧ During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- ✧ Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- ✧ As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

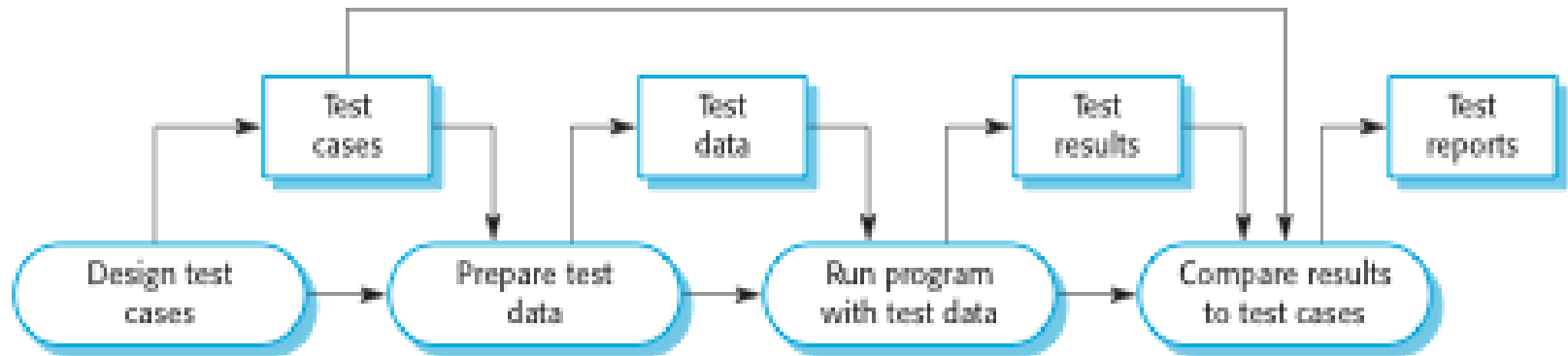
# Inspections and Testing

---



- ✧ Inspections and testing are complementary and not opposing verification techniques.
- ✧ Both should be used during the V & V process.
- ✧ Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- ✧ Inspections cannot check non-functional characteristics such as performance, usability, etc.

# Software Testing Process



\*Test Case : Test Data + Expected Result

# Stages of testing

---



- ✧ Stage 1 : **Development testing**, where the system is tested during development to discover bugs and defects.
- ✧ Stage 2 : **Release testing**, where a separate testing team test a complete version of the system before it is released to users.
- ✧ Stage 3 : **User testing**, where users or potential users of a system test the system in their own environment.

# S1. Development testing

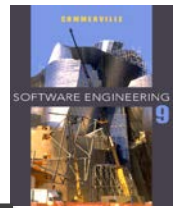
---



- ✧ Development testing includes all testing activities that are carried out by the team developing the system.
  - Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  - System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

# S1.1 Unit testing

---



- ✧ Unit testing is the process of testing individual components in isolation.
- ✧ It is a defect testing process.
- ✧ Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality.

# Automated Unit testing (JUnit)

---



- ✧ Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- ✧ Kent Beck developed the first xUnit automated test tool for Smalltalk in mid-90's
- ✧ JUnit test generators now part of many Java IDEs (Eclipse, BlueJ, Jbuilder, DrJava)
- ✧ JUnit has become the standard tool for Test-Driven Development in Java (see [JUnit.org](http://JUnit.org))
- ✧ xUnit tools have since been developed for many other languages (Perl, C++, Python, Visual Basic, C#, ...)



# S1.2 Component testing

---

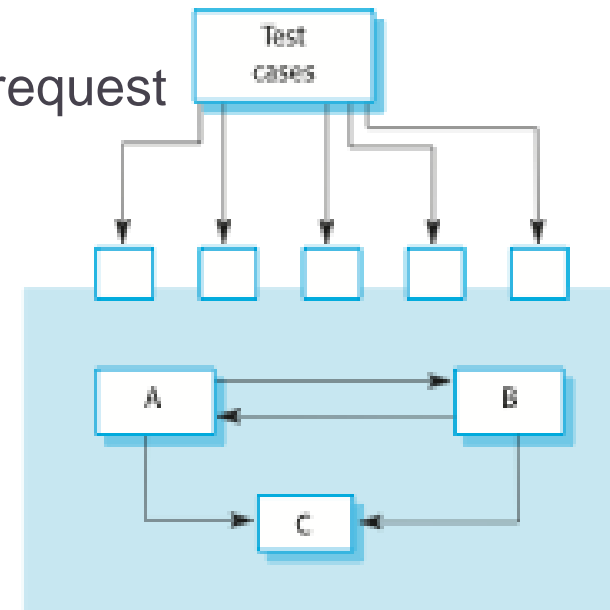


- ✧ Software components are often composite components that are made up of several interacting objects.
- ✧ You access the functionality of these objects through the defined component interface.
- ✧ Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - You can assume that unit tests on the individual objects within the component have been completed.

# Interface testing



- ✧ Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- ✧ Interface types
  - Parameter interfaces Data passed from one method or procedure to another.
  - Shared memory interfaces Block of memory is shared between procedures or functions.
  - Message passing interfaces Sub-systems request services from other sub-systems



# Interface errors

---



## ✧ Interface misuse

- A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.

## ✧ Interface misunderstanding

- A calling component embeds assumptions about the behaviour of the called component which are incorrect.

## ✧ Timing errors

- The called and the calling component operate at different speeds and out-of-date information is accessed.

# S1.3 System testing

---



- ✧ System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- ✧ The focus in system testing is testing the interactions between components.
- ✧ System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- ✧ System testing tests the emergent behavior of a system.

# System and component testing

---



- ✧ During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- ✧ Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
  - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

## S2. Release testing

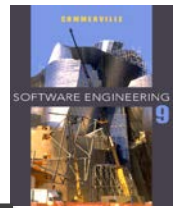
---



- ✧ Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- ✧ The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
  - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- ✧ Release testing is usually a black-box testing process where tests are only derived from the system specification.

# Release testing and system testing

---



- ✧ Release testing is a form of system testing.
- ✧ Important differences:
  - A separate team that has not been involved in the system development, should be responsible for release testing.
  - System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

# Performance testing

---



- ✧ Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- ✧ Tests should reflect the profile of use of the system.
- ✧ Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- ✧ Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.



# S3. User testing

---



- ✧ User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- ✧ User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

# Types of user testing

---



## ✧ Alpha testing

- Users of the software work with the development team to test the software at the developer's site.

## ✧ Beta testing

- A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

## ✧ Acceptance testing

- Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

# Black-Box Testing Techniques

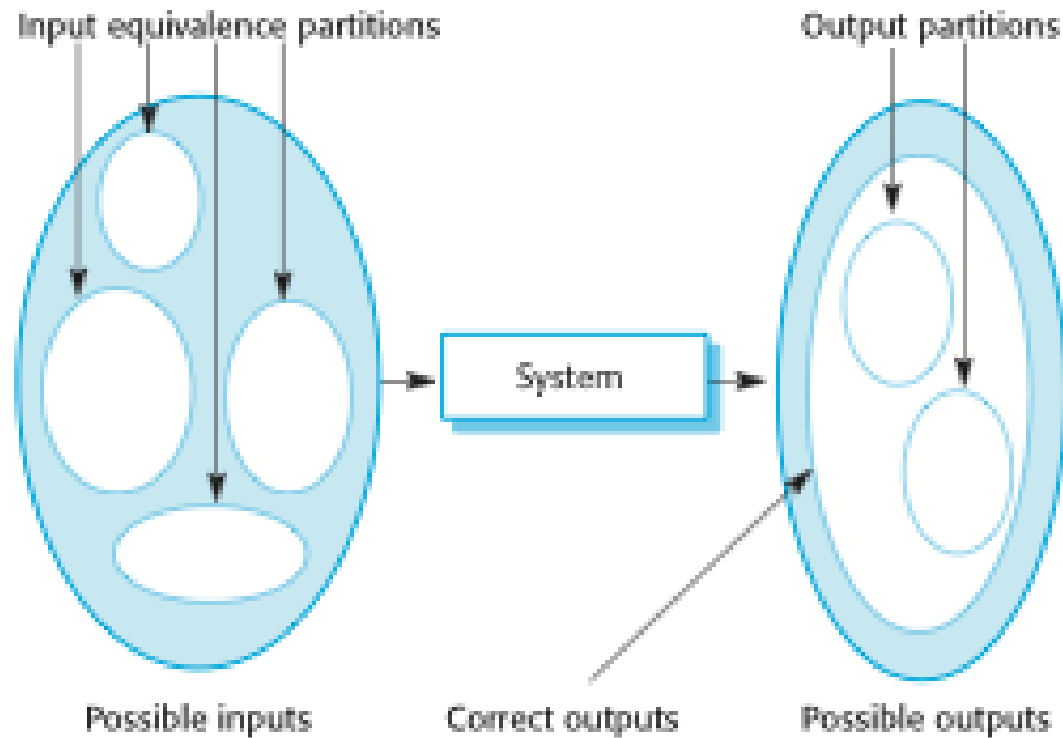
# Partition testing

---

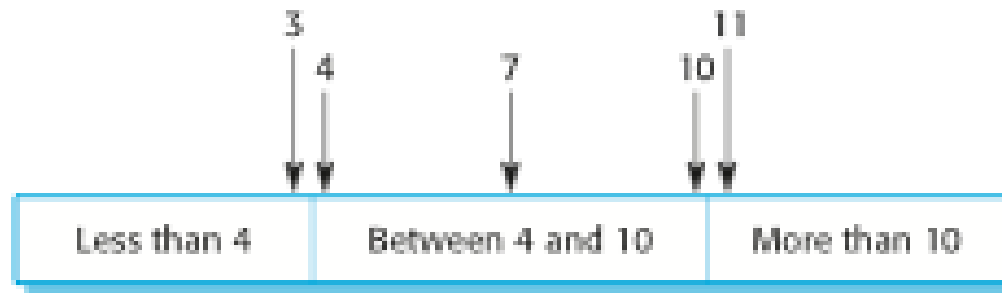


- ✧ Input data and output results often fall into different classes where all members of a class are related.
- ✧ Each of these classes is an **equivalence partition** or domain where the program behaves in an equivalent way for each class member.
- ✧ Test cases should be chosen from each partition.

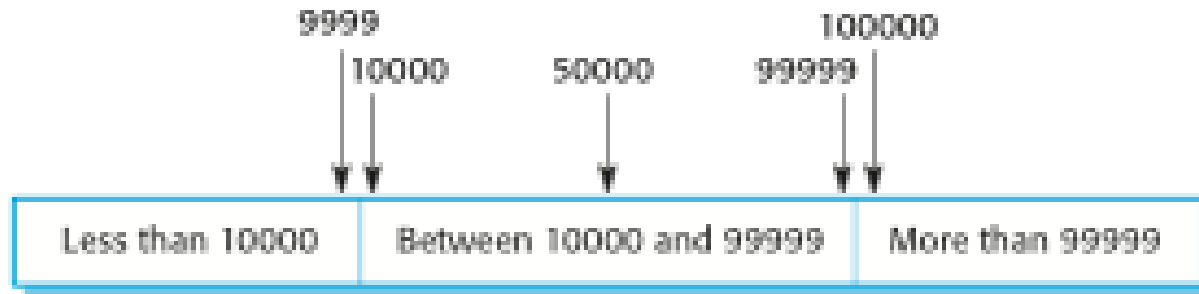
# Equivalence partitioning



# Equivalence partitions



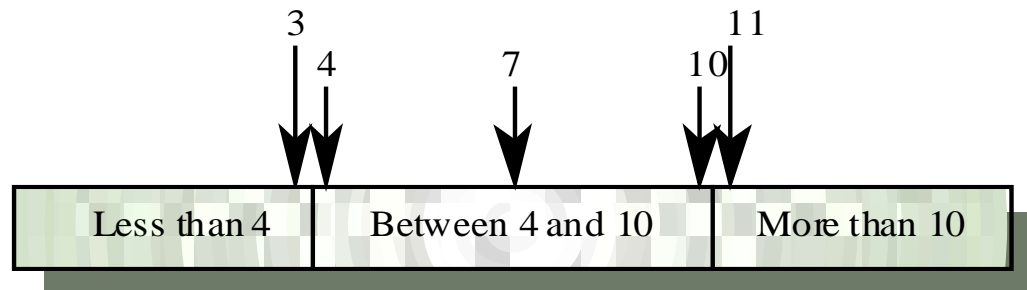
Number of input values



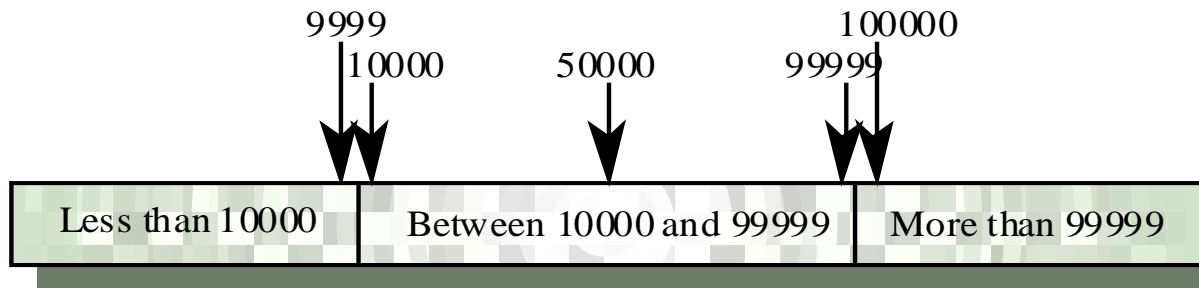
Input values

# Equivalence partitioning

- ✧ Partition system inputs and outputs into 'equivalence sets'
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <0-9999>, <10000-99999> and <100000 - >
- ✧ Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 100000



Number of input values

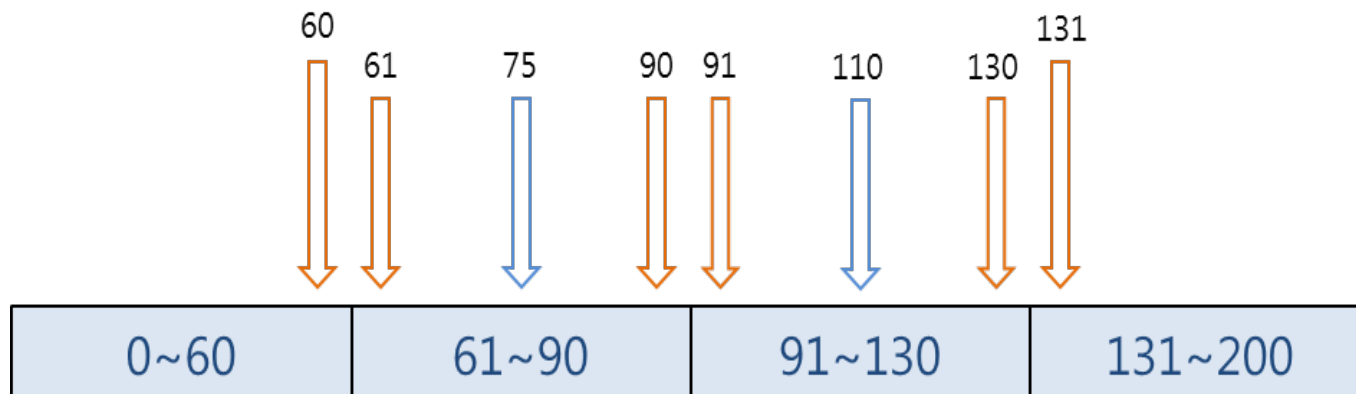


# Testing Example 1



☆ 디지털 계기화면의 Oil 압력 게이지의 PSI 값의 범위에 대해서 출력 압력 값과 색상으로 경고 메시지를 화면에 출력한다. 범위는 다음과 같다.

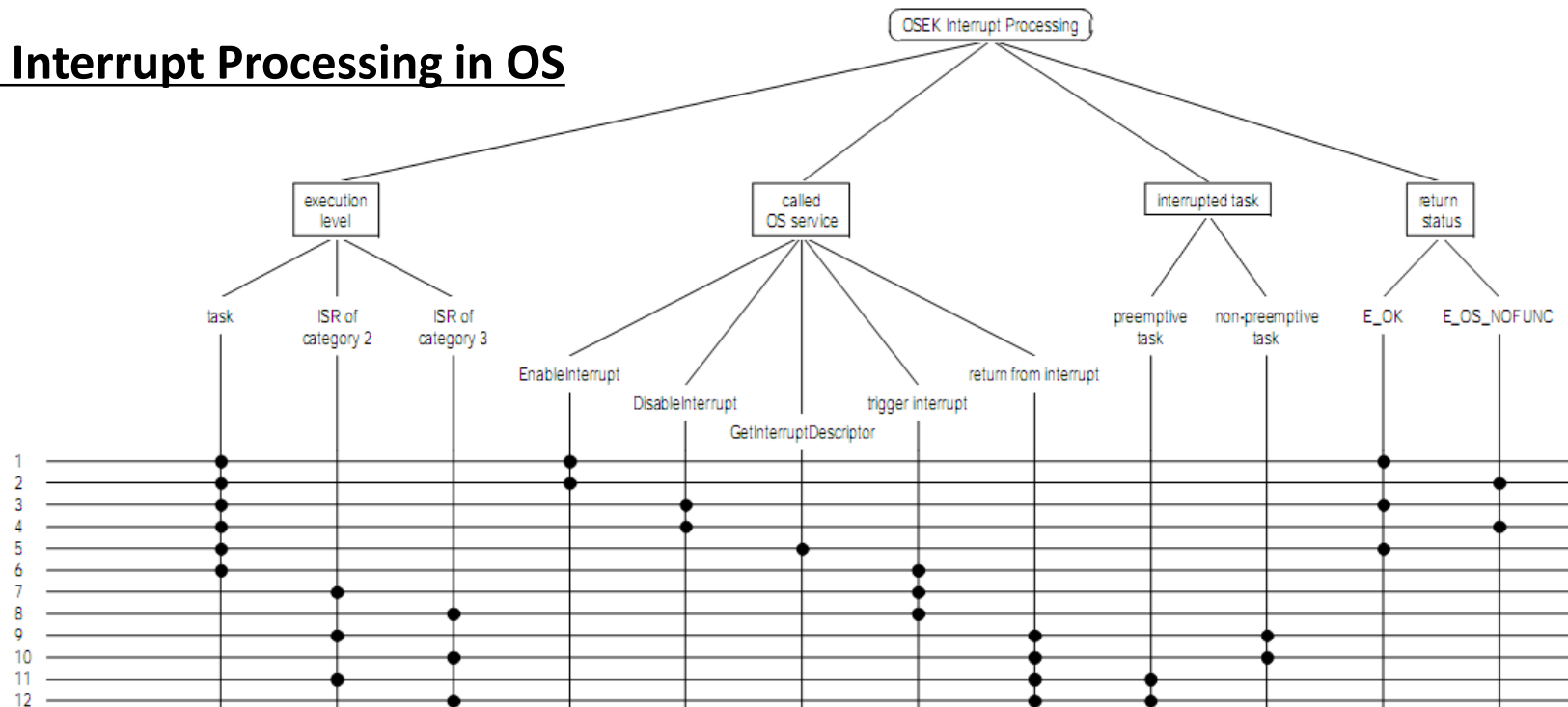
- 0~60 PSI – Red Display, Very Low Pressure
- 61~90 PSI – Yellow Display, Low Pressure
- 91~130 PSI – Green Display, Normal Pressure
- 131~200 PSI – Red Display, High Pressure





# Testing Example 2

## Interrupt Processing in OS



Test case No.	Sched. policy Conf. class Status	Action	Expected Result
1	n, m, f B1, B2, E1, E2 s, e	Call <code>EnableInterrupt()</code> . All requested interrupts are disabled	Enable interrupts. Service returns <code>E_OK</code>
2	n, m, f B1, B2, E1, E2 e	Call <code>EnableInterrupt()</code> . At least one of the requested interrupts is already enabled	Enable interrupts. Service returns <code>E_OS_NOFUNC</code>

# Testing Example 3

---



## ✧ MHC-PMS requirements:

- If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
- If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

# Testing Example 3

---



## ✧ **Test Case #1**

- Set up a patient record with no known allergies.
- Prescribe medication for allergies that are known to exist.
- Check that a warning message is not issued by the system.

## ✧ **Test Case #2**

- Set up a patient record with a known allergy.
- Prescribe the medication to that the patient is allergic to
- Check that the warning is issued by the system.

## ✧ **Test Case #3**

- Set up a patient record in which allergies to two or more drugs are recorded.
- Prescribe both of these drugs separately
- Check that the correct warning for each drug is issued.

## ✧ **Test Case #4**

- Prescribe two drugs that the patient is allergic to.
- Check that two warnings are correctly issued.

## ✧ **Test Case #5**

- Prescribe a drug that issues a warning and overrule that warning.
- Check that the system requires users the reason of overruling warning

---

# White Box Testing

# White-box testing

---

- ✧ Sometime called **structural testing**
- ✧ Derivation of test cases according to program structure.
- ✧ Knowledge of the program is used **to identify additional test cases**
- ✧ Objective is to exercise all program statements, all conditions, all decisions (not all path combinations)

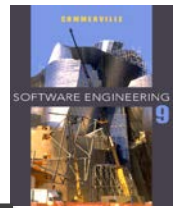


## class BinSearch {

```
public static void search ( int key, int [] elemArray, Result r )
{
    int bottom = 0 ;
    int top = elemArray.length - 1 ;
    int mid ;
    r.found = false ; r.index = -1 ;
    while ( bottom <= top )
    {
        mid = (top + bottom) / 2 ;
        if (elemArray [mid] == key)
        {
            r.index = mid ;
            r.found = true ;
            return ;
        } // if part
        else
        {
            if (elemArray [mid] < key)
                bottom = mid + 1 ;
            else
                top = mid - 1 ;
        }
    } //while loop
} // search
} //BinSearch
```

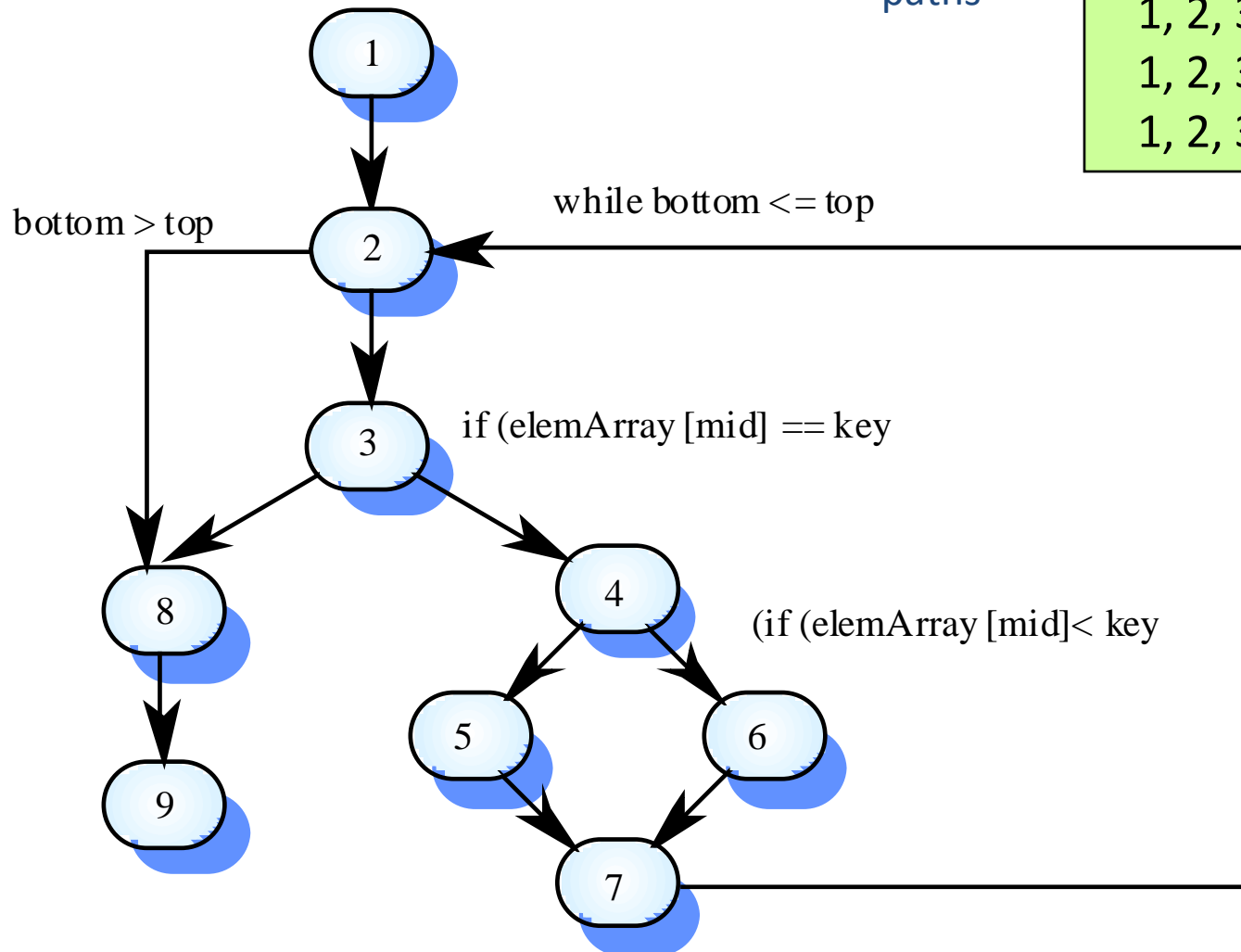
## Binary search (Java)

# Program Flow Graph



Independent  
paths

1, 2, 8, 9  
1, 2, 3, 8, 9  
1, 2, 3, 4, 5, 7, 2, 8, 9  
1, 2, 3, 4, 6, 7, 2, 8, 9



# Test Coverage I

---



## ✧ Statement Coverage

- Every statement in the program has been executed at least once

## ✧ Decision/Branch) Coverage

- Every point of entry and exit in the program has been invoked at least once
- Every decision in the program has taken all possible outcomes at least once

## ✧ Condition/Decision Coverage(C/DC)

- Every point of entry and exit in the program has been invoked at least once
- Every condition in a decision in the program has taken all possible outcomes at least once
- Every decision in the program has taken all possible outcomes at least once

**Ref :** John Joseph Chilenski and Steven P. Miller, “Applicability of modified condition/decision coverage to software testing,” Software Engineering Journal, Sep. 1994



# C/DC 와 MC/DC의 차이점



## ✧ Or 연산의 Decision Coverage

- Minimal Test Set : (TT, FF), (TF, FF), (FT, FF)

## ✧ Or 연산의 Condition Coverage

- Minimal Test Set : (TT, FF), (TF, FT)

## ✧ Or 연산의 C/DC Coverage

- Minimal Test Set (TT, FF)

A	B	A  B
T	T	T
T	F	T
F	T	T
F	F	F

# Test Coverage II

---



## ✧ Modified Condition/Decision Coverage (MC/DC)

- Every point of entry and exit in the program has been invoked at least once
- Every condition in a decision in the program has taken all possible outcomes at least once
- Each condition has been shown to independently affect the decision output.
  - A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions

# MC/DC's Example 1



✧ A or B

- $\{2, 3, 4\} = \{(T,F), (F,T), (F,F)\}$

Index	A	B	A    B	A	B
1	T	T	T		
2	T	F	T	4	
3	F	T	T		4
4	F	F	F	2	3

# Test Driven Development(TDD)

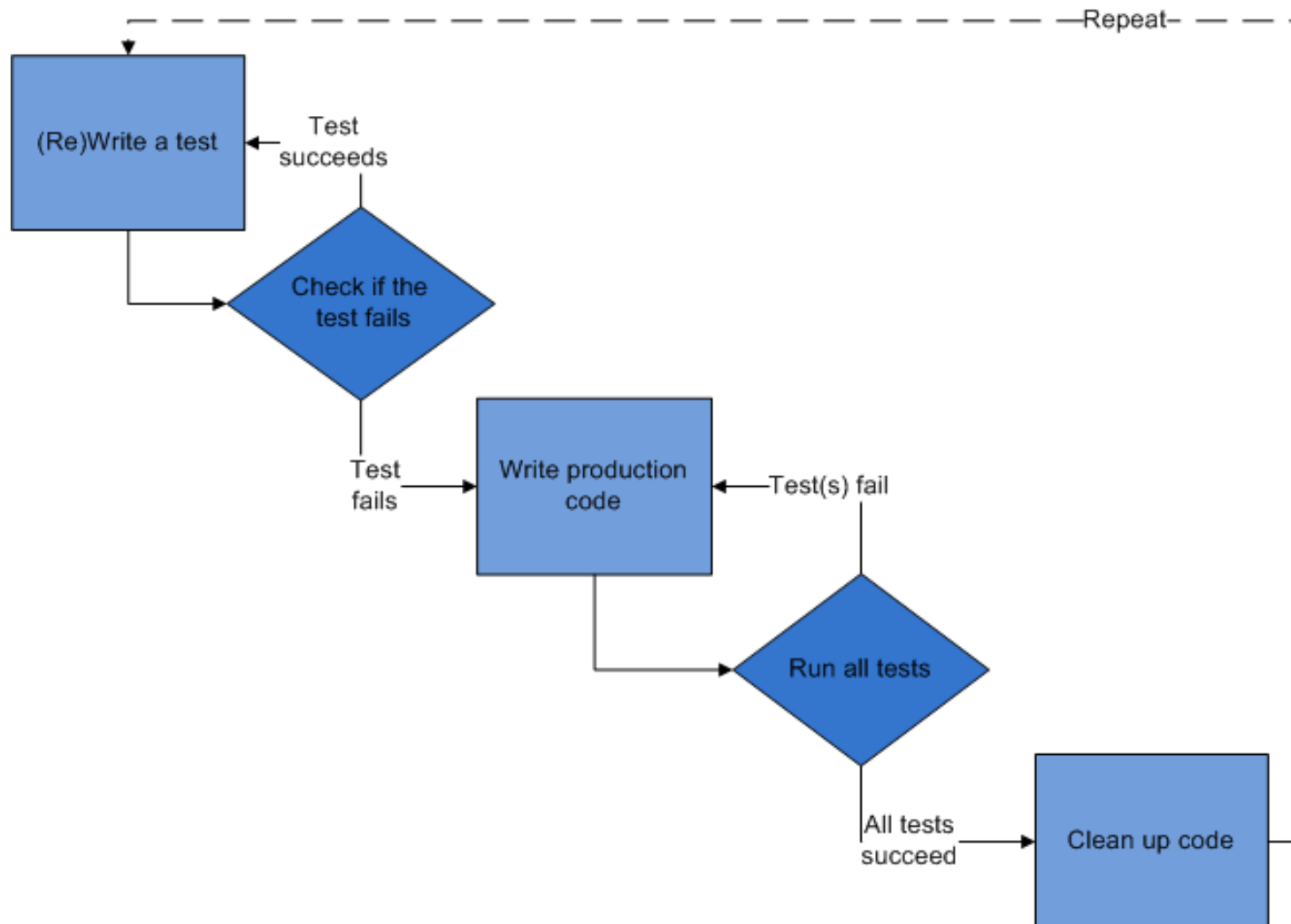
# Test-driven development

---



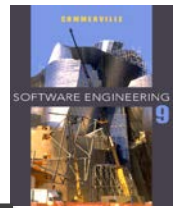
- ✧ Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- ✧ Tests are written before code and ‘passing’ the tests is the critical driver of development.
- ✧ You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.
- ✧ TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

# Test-driven development



# TDD process activities

---



- ✧ Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- ✧ Write a test for this functionality and implement this as an automated test.
- ✧ Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- ✧ Implement the functionality and re-run the test.
- ✧ Once all tests run successfully, you move on to implementing the next chunk of functionality.

# Benefits of test-driven development

---



## ✧ Code coverage

- Every code segment that you write has at least one associated test so all code written has at least one test.

## ✧ Regression testing

- Regression testing is testing the system to check that changes have not 'broken' previously working code.
- A regression test suite is developed incrementally as a program is developed.

## ✧ Simplified debugging

- When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.

## ✧ System documentation

- The tests themselves are a form of documentation that describe what the code should be doing.



# TDD 적용 현안

---



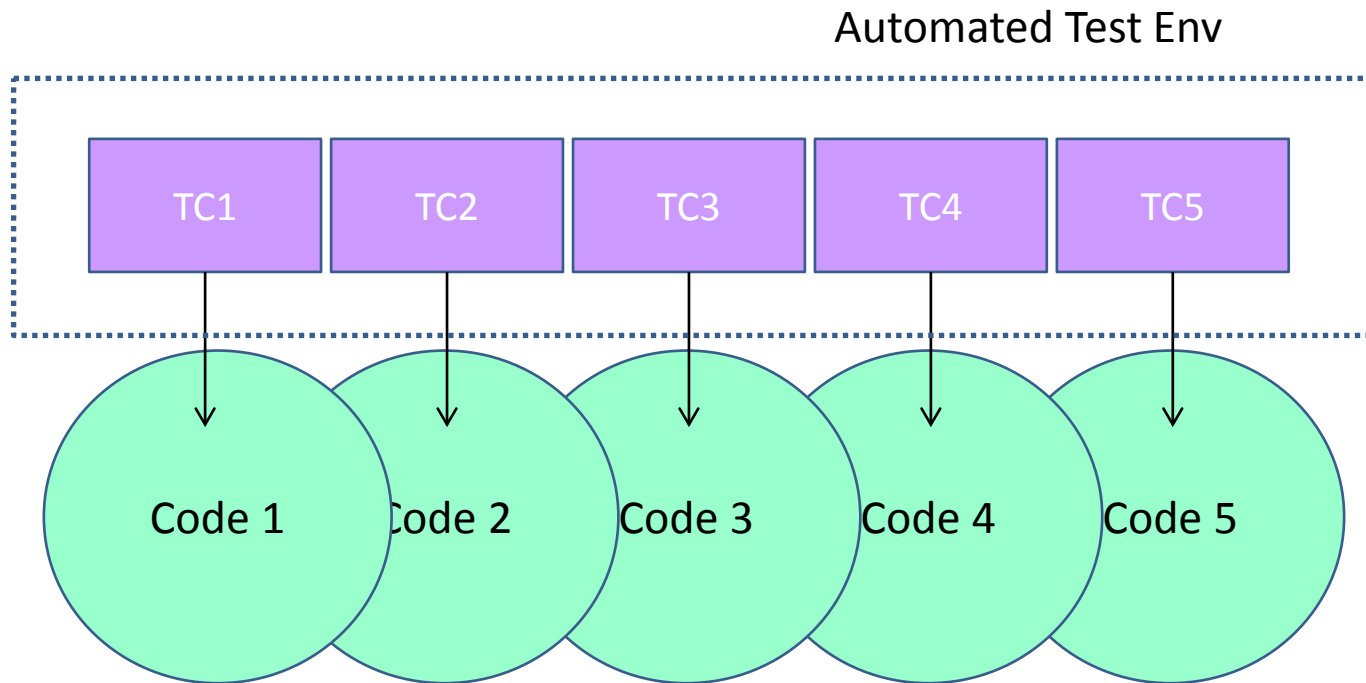
✧ Test Case 생성 시점을 어디에 배치하느냐

- All Development → All Test Cases
- One Module Development → One Test Case
- One Test Case → One Module Development
- All Test Cases → All Development

# TDD 적용 예



- Iterative and Incremental Development
  - Interweaving development and testing repetitively



# Embedded SW Testing

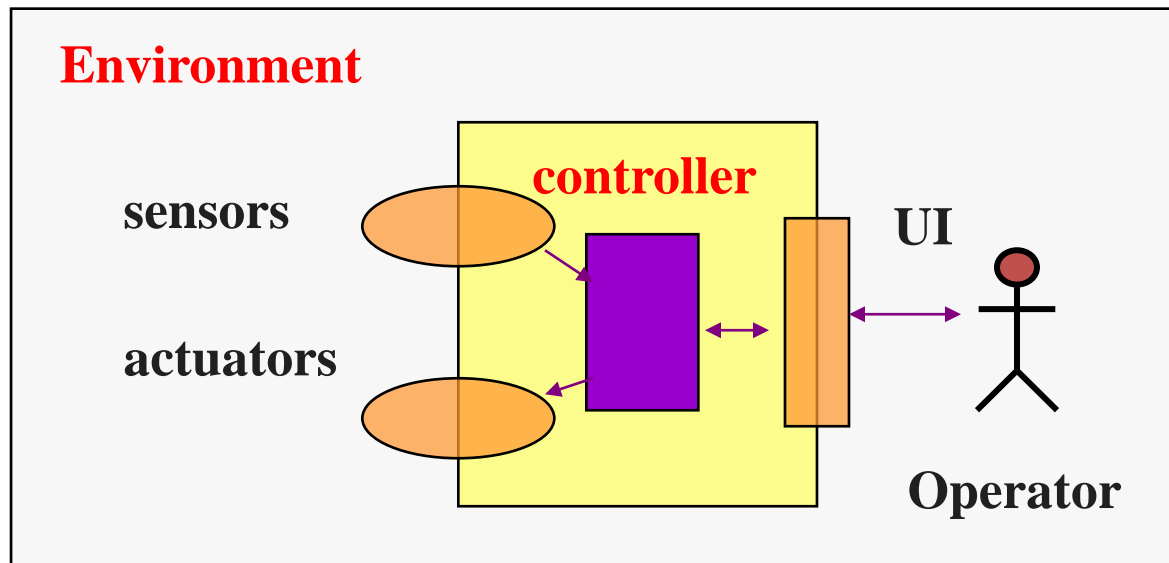
# Embedded Software

## ✧ Embedded systems의 정의

- Computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function

## ✧ Embedded system의 주요 구성요소

- Sensors, Actuators, Controller

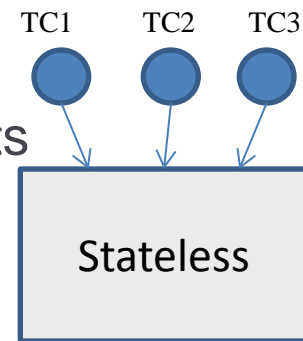


# Difficulty of Embedded SW Testing (I)

✧ In the perspective of unit testing of a component

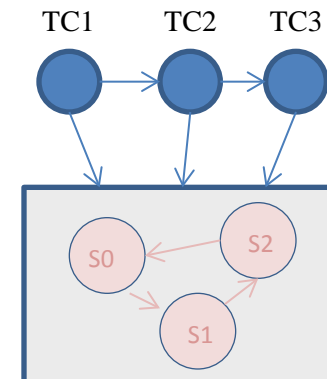
✧ Function

- Inputs → Outputs
- Generate independent Test Cases by considering inputs



✧ Embedded Software

- The same inputs may have different results due to memory feature
- Perform time-critical operations
- Usually use input sequence
  - TC1 → TC2 → TC4
  - TC1 → TC3



# Difficulty of Embedded SW Testing (II)

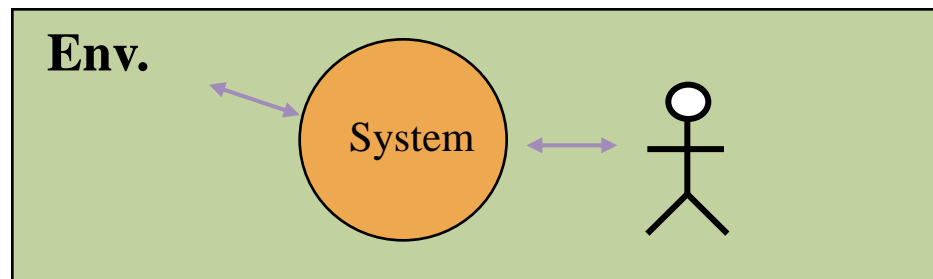
✧ In the perspective of system interaction loop

✧ Information System Software

- Information systems interact with users
- System and users make a closed interaction loop
- In users' perspective, system behaviors are tested

✧ Embedded Software

- Embedded systems interact with operators and their environment
- System and operators are in open loop without environment
- It is not easy to handle their environment for testing system



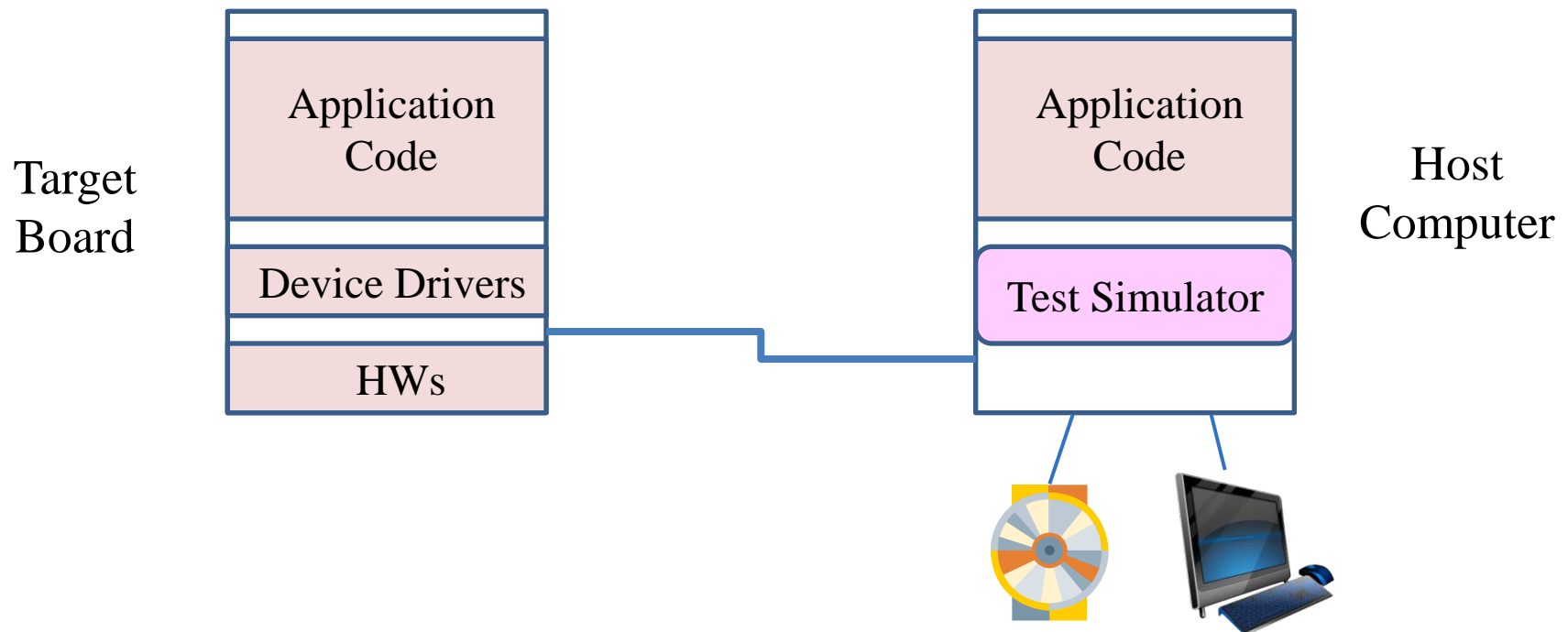
# Difficulty of Embedded SW Testing (III)

- ✧ In the perspective of development and running environment
- ✧ Information System Software
  - Usually information systems have rich resources
  - The development and running environment can be the same
- ✧ Embedded System Software
  - Embedded systems have restricted resources
  - Embedded software is developed by cross development
  - It is not possible to perform testing in target board



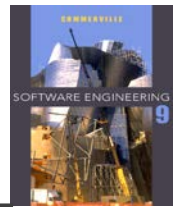
# Cross Testing Environment

- ✧ Host Computer has a test simulator which mimics HW and device driver of the target board
- ✧ For making test simulator, it is necessary to clearly separate hardware dependent and independent parts





# Testing Stage of Embedded SW



Test Level		Embedded SW	Processor	Rest of embedded System	Plant
One-way simulation	MT (Model Test)	Simulated	-	-	-
Feed-back simulation	MiL (Model-in-the-Loop)	Simulated	-	-	Simulated
Rapid prototyping	RP	Experimental	Experimental	Experimental	Real
SW Unit, SW integration (1)	SiL (SW-in-the-Loop)	Experimental (host)	Host	Simulated	Simulated
SW Unit, SW integration (2)	SiL	Real (target)	Emulator	Simulated	Simulated
HW/SW integration	HiL (HW-in-the-Loop)	Real (target)	Real (target)	Experimental	Simulated
System integration	HiL	Real (target)	Real (target)	Prototype	Simulated
Environmental	HiL/ST (System Test)	Real (target)	Real (target)	Real	Simulated
Pre-Production	ST	Real (target)	Real (target)	Real	Real