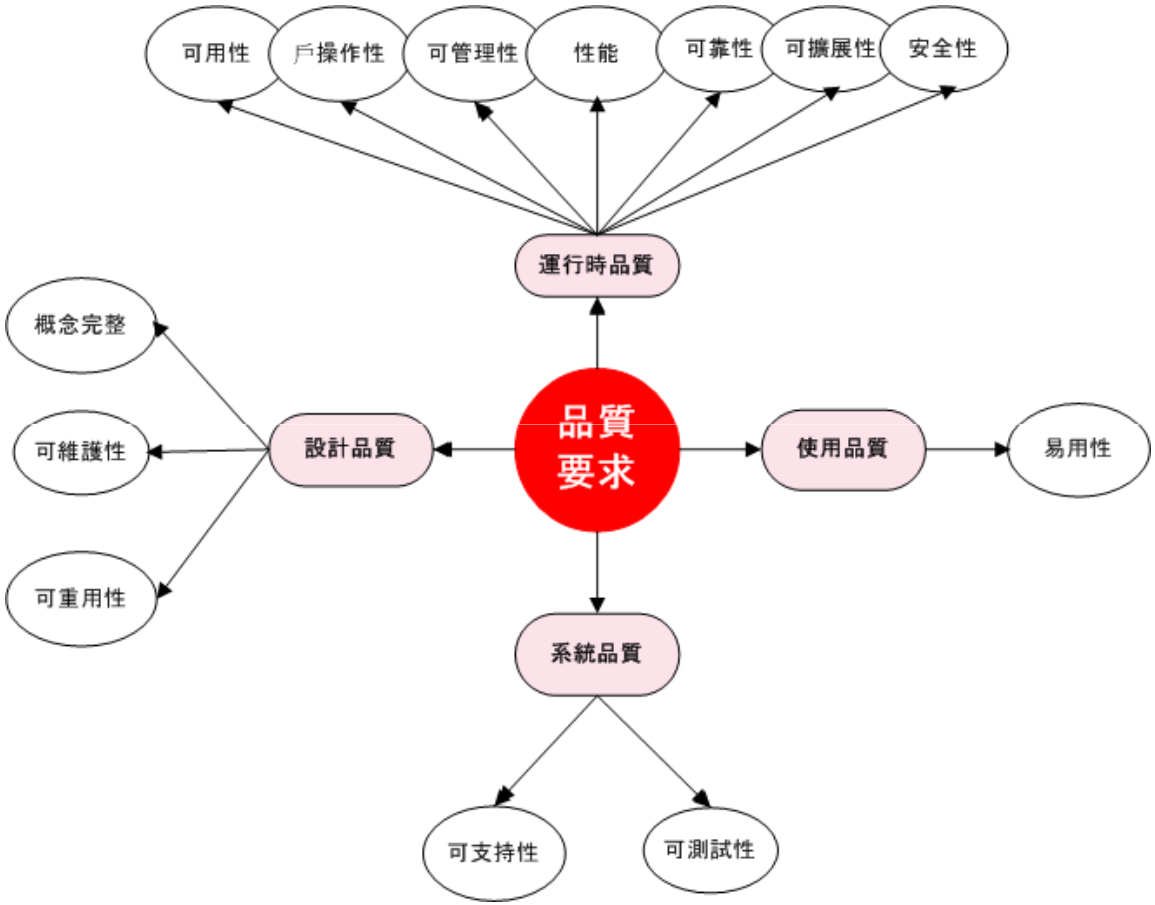
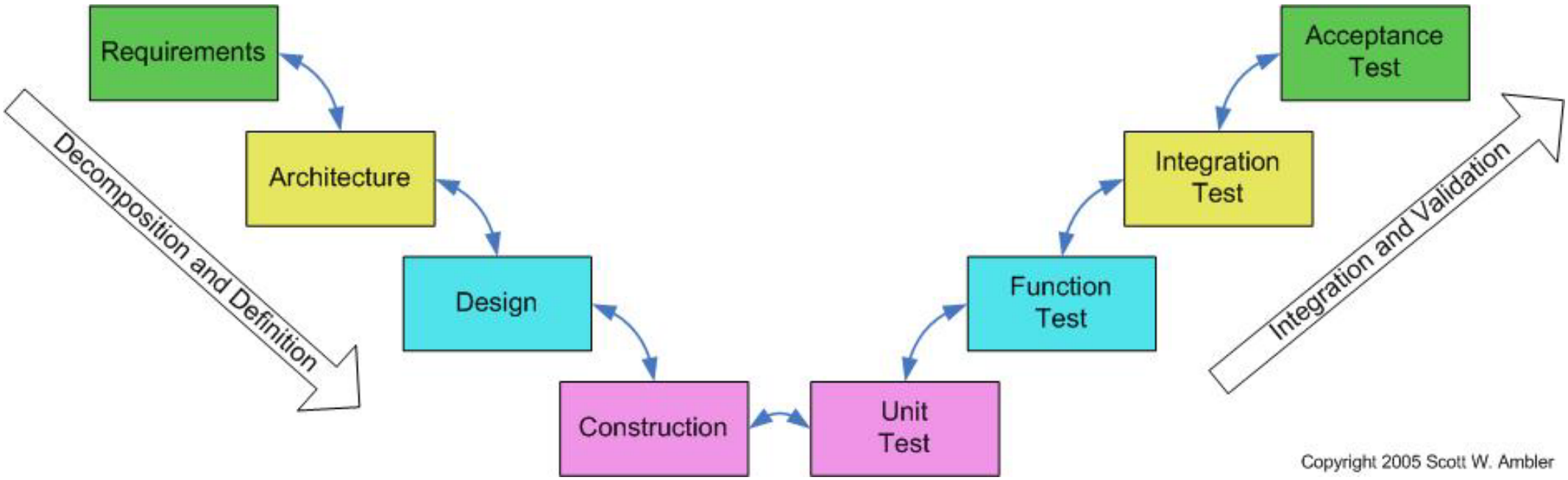
# Introduce Test

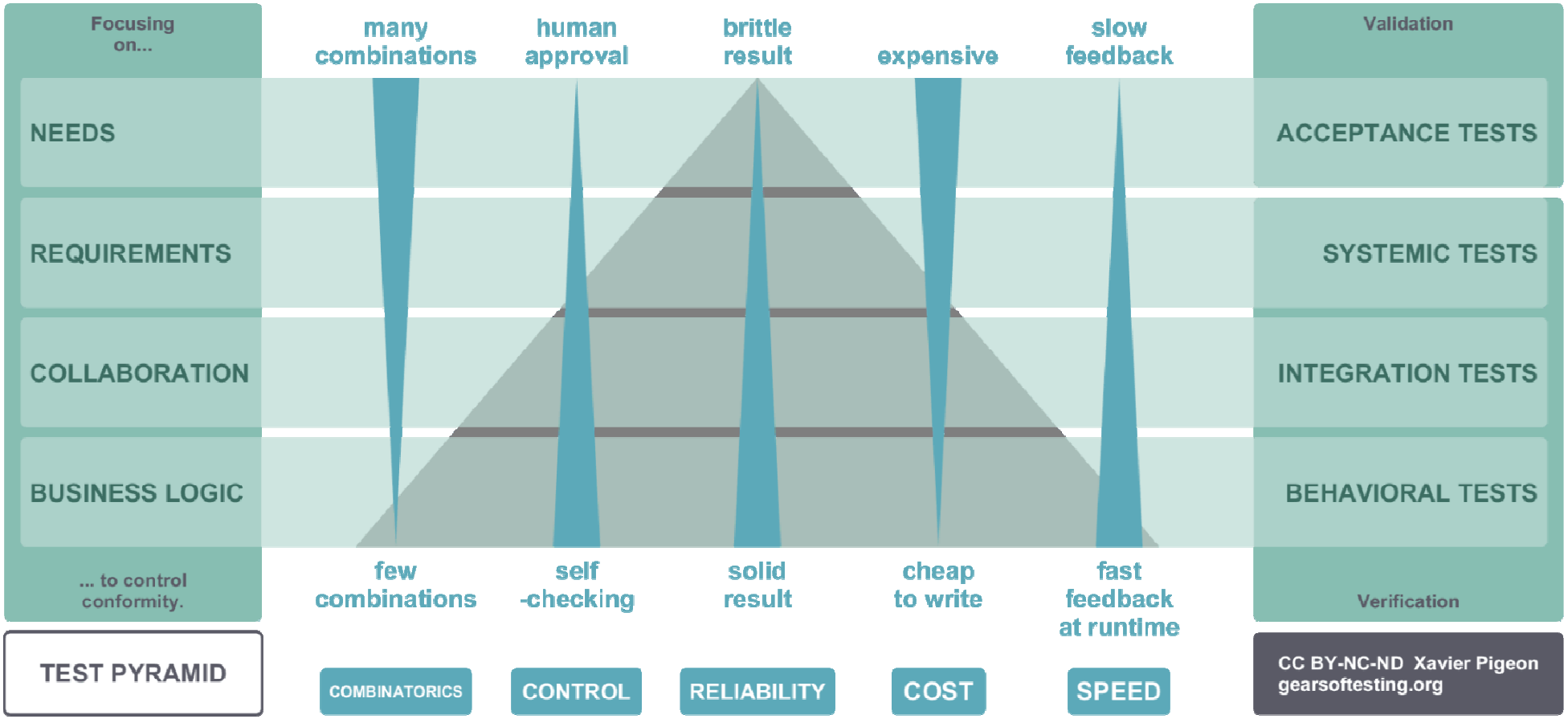
# 品質要求

- 設計品質(Scalable)
  - 概念完整
  - 可維護性
  - 可重用性
- 系統品質
  - 可測試性
  - 可支持性
- 運行品質(HA)
  - 可用性,互操作性,可管理性,
  - 性能,可靠性,可擴展性,安全性
- 使用品質
  - 易用性

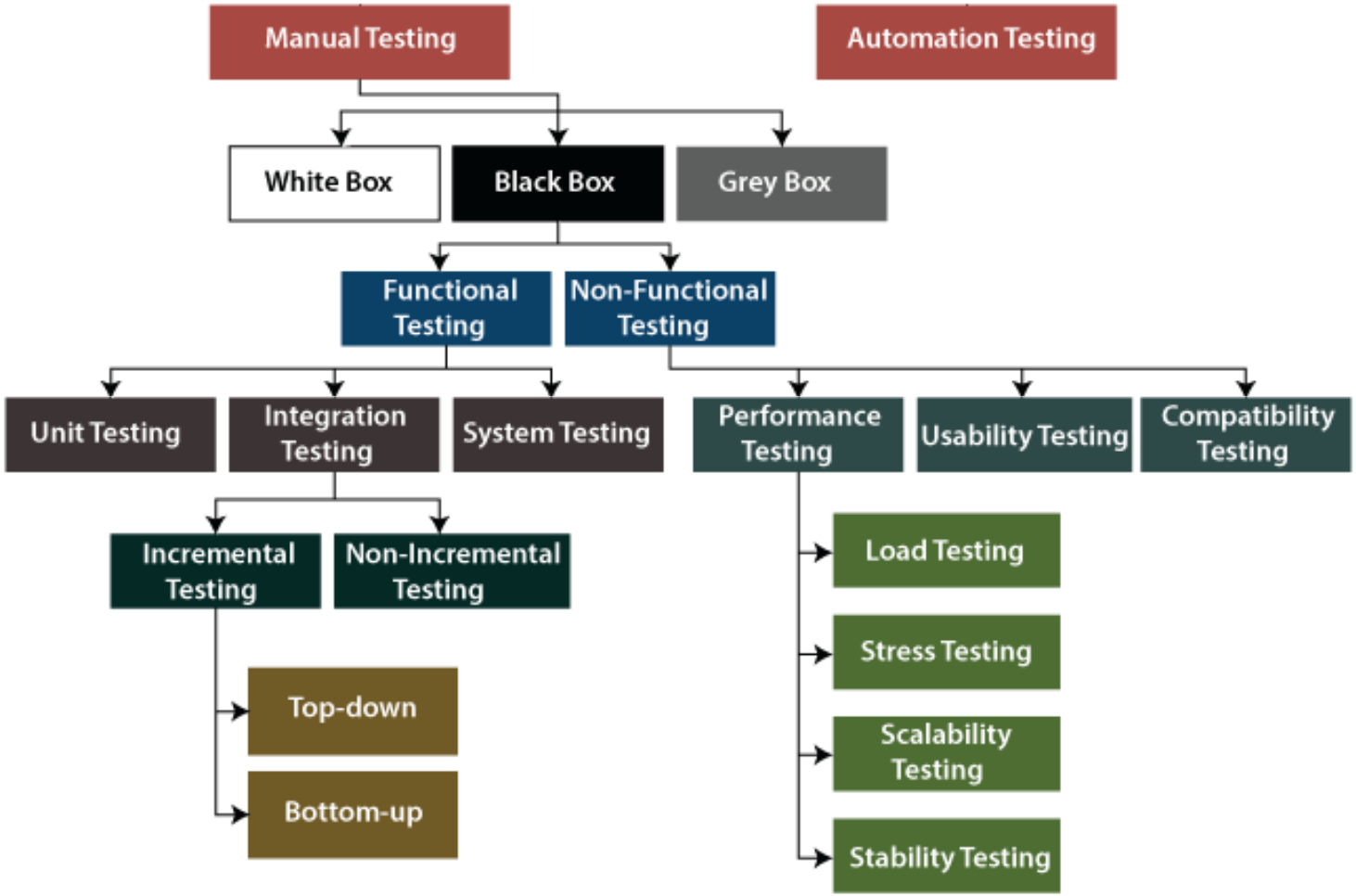


# V-Model

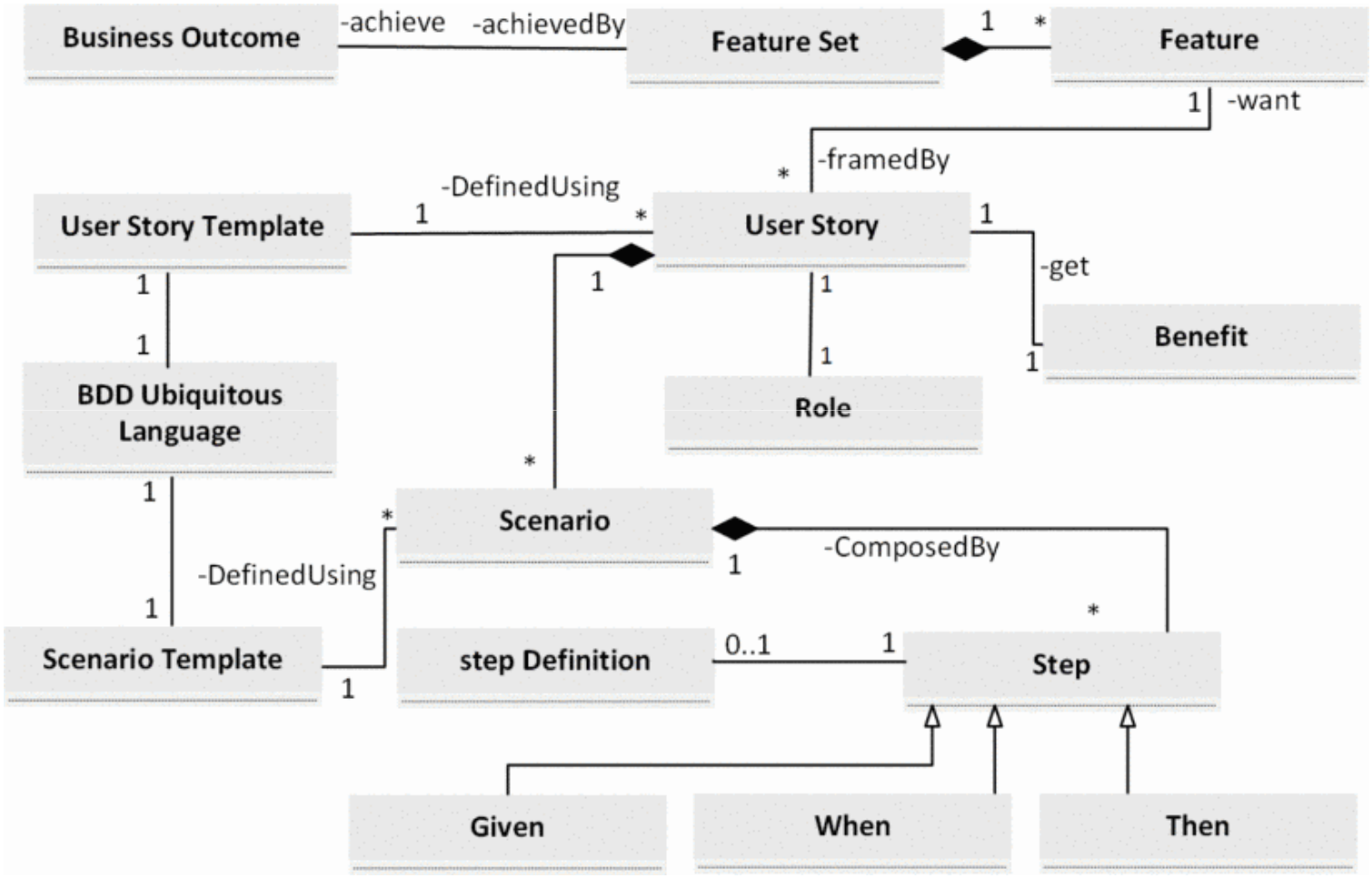

Requirements
Architecture
Design
Construction
Unit Test
Function Test
Integration Test
Acceptance Test
Decomposition and Definition
Integration and Validation
Copyright 2005 Scott W. Ambler


Focusing on...
many combinations
human approval
brittle result
expensive
slow feedback
Validation
NEEDS
ACCEPTANCE TESTS
REQUIREMENTS
SYSTEMIC TESTS
COLLABORATION
INTEGRATION TESTS
BUSINESS LOGIC
BEHAVIORAL TESTS
... to control conformity.
few combinations
self -checking
solid result
cheap to write
fast feedback at runtime
Verification
TEST PYRAMID
COMBINATORICS
CONTROL
RELIABILITY
COST
SPEED
CC BY-NC-ND Xavier Pigeon
gearsoftesting.org

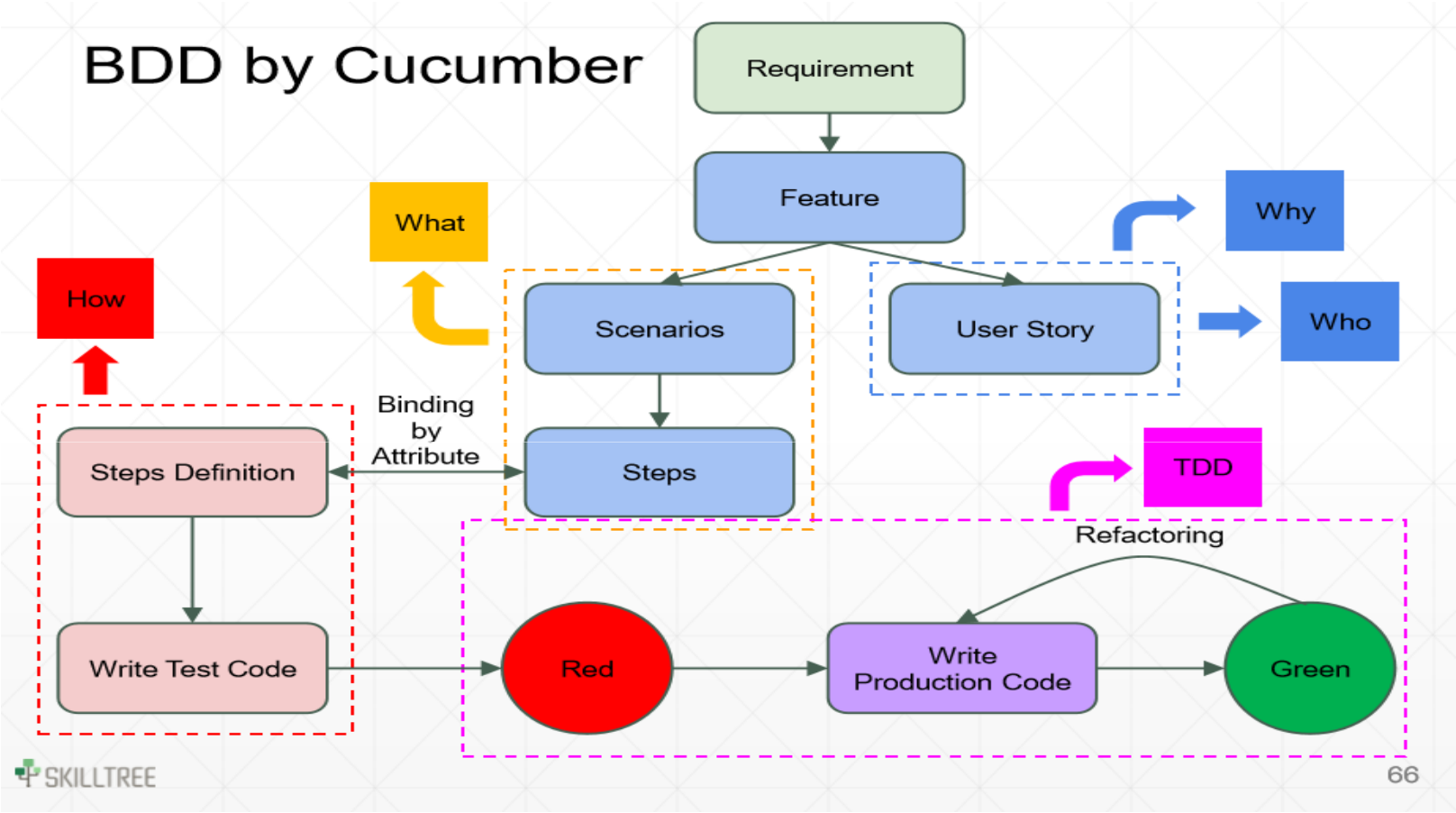
# Type of Software testing



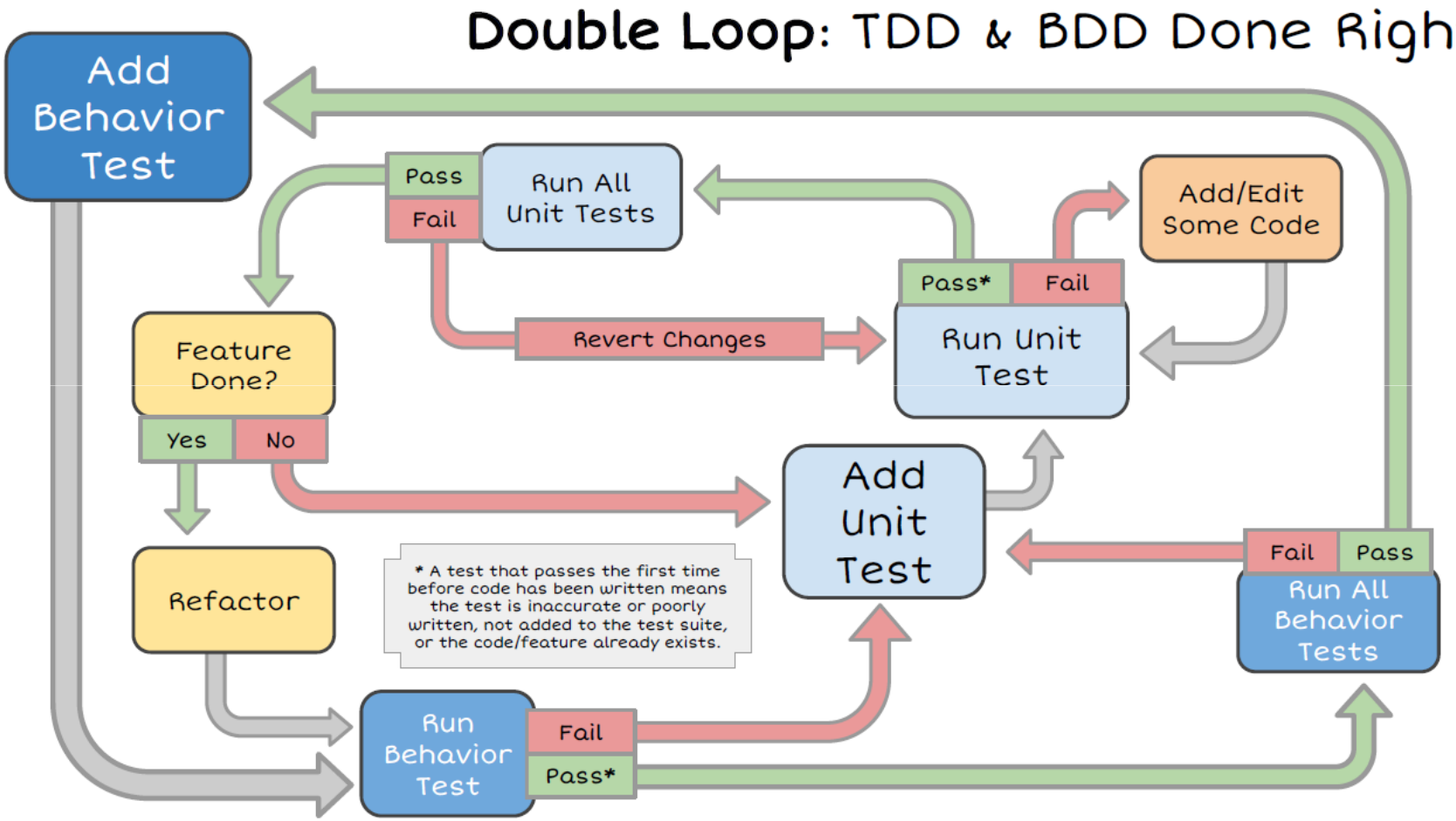
# BDD metamodel



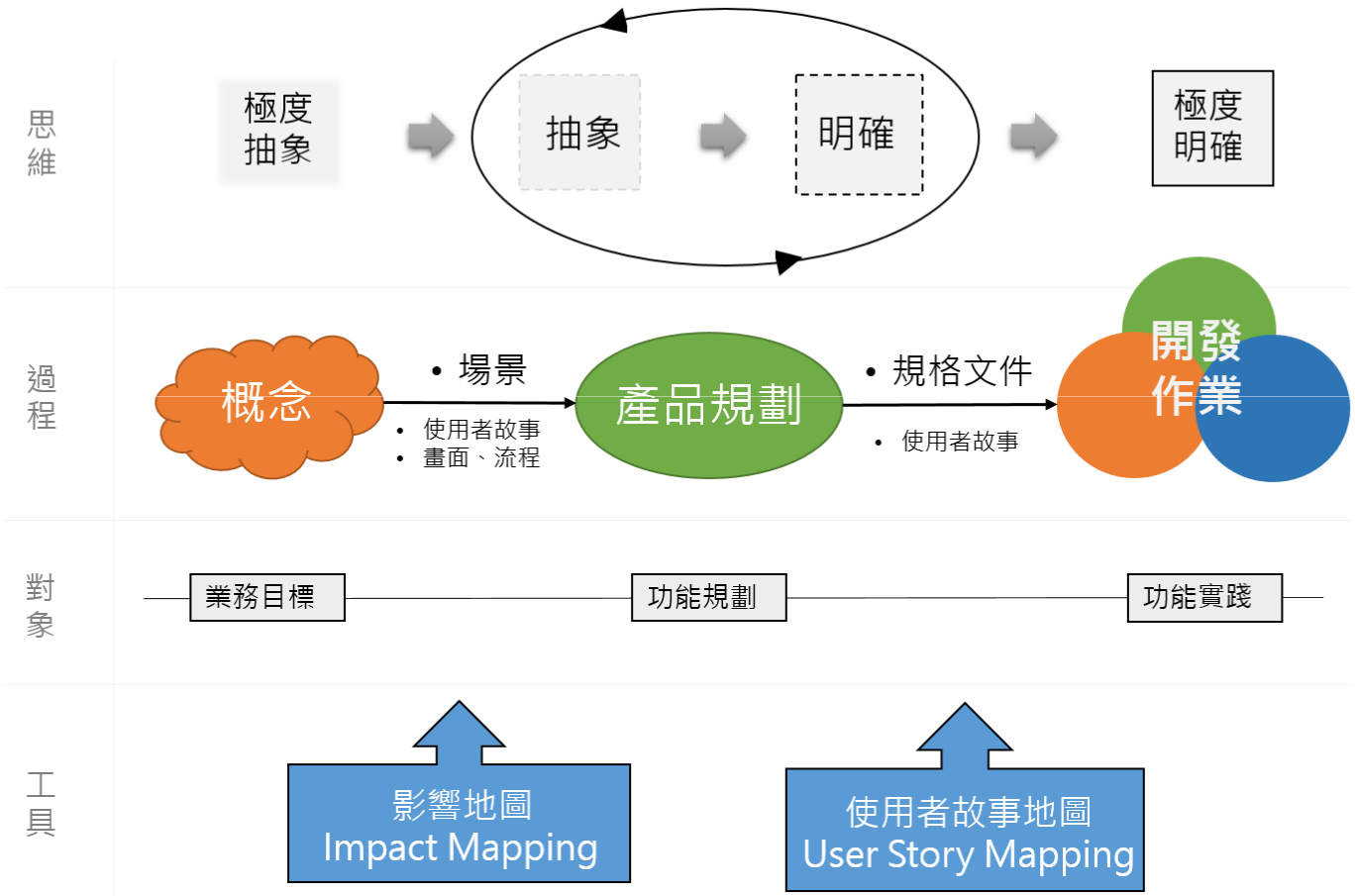
# Cucumber

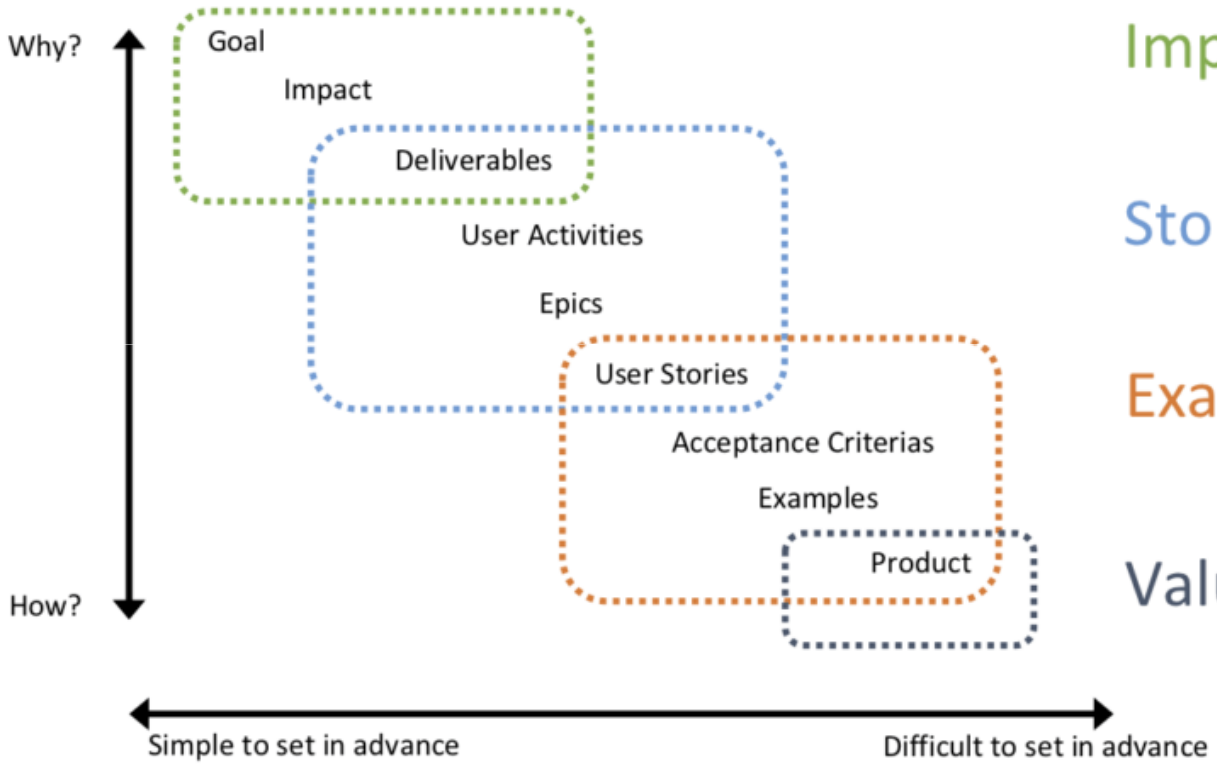

BDD by Cucumber
Requirement
Feature
What
Why
How
Scenarios
User Story
Who
Binding
by
Attribute
Steps Definition
Steps
TDD
Refactoring
Write Test Code
Red
Write
Production Code
Green
SKILLTREE
66

# Double Loop Workflow



# Requirement


思維
極度抽象
抽象
明確
極度明確
過程
概念
• 場景
• 使用者故事
• 畫面、流程
產品規劃
• 規格文件
• 使用者故事
開發作業
對象
業務目標
功能規劃
功能實踐
工具
影響地圖
Impact Mapping
使用者故事地圖
User Story Mapping



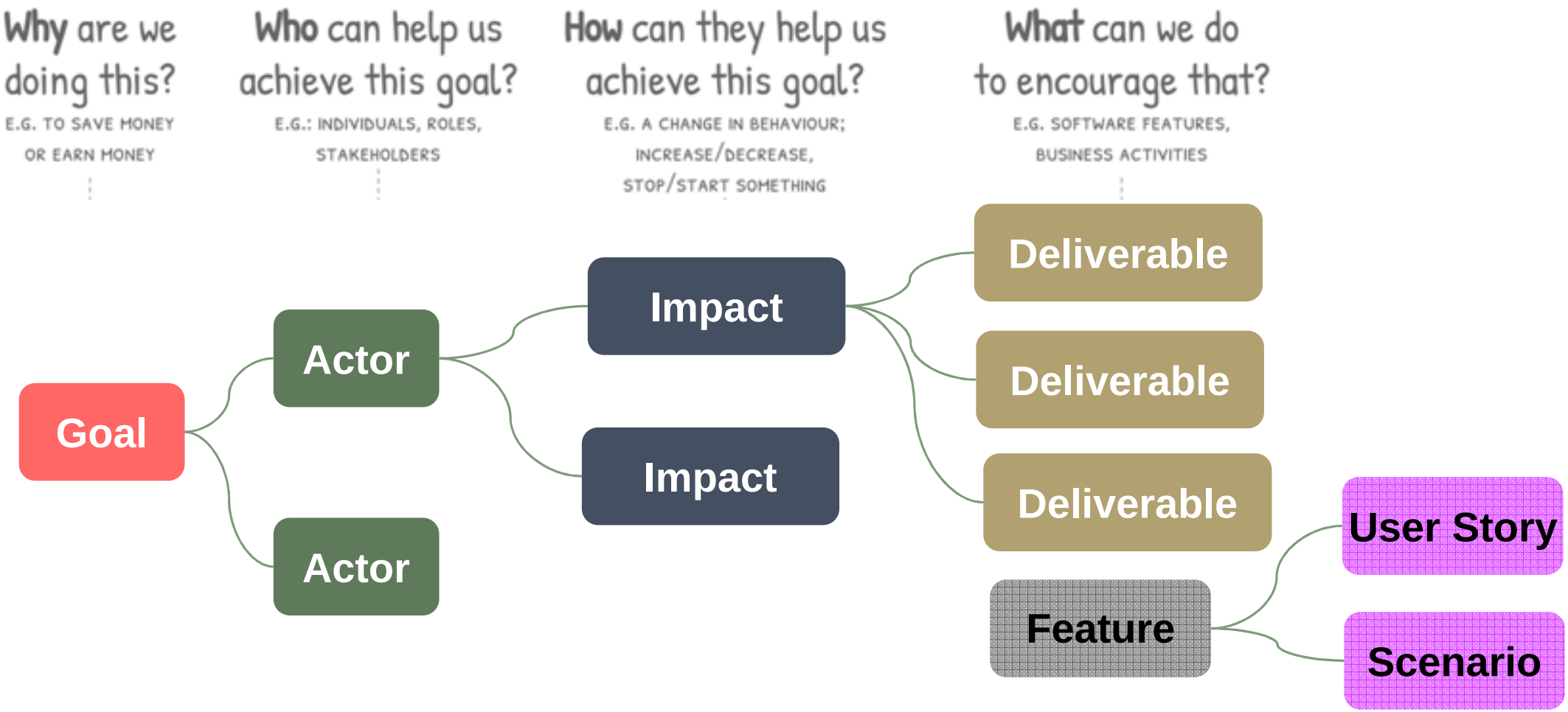

Impact Mapping

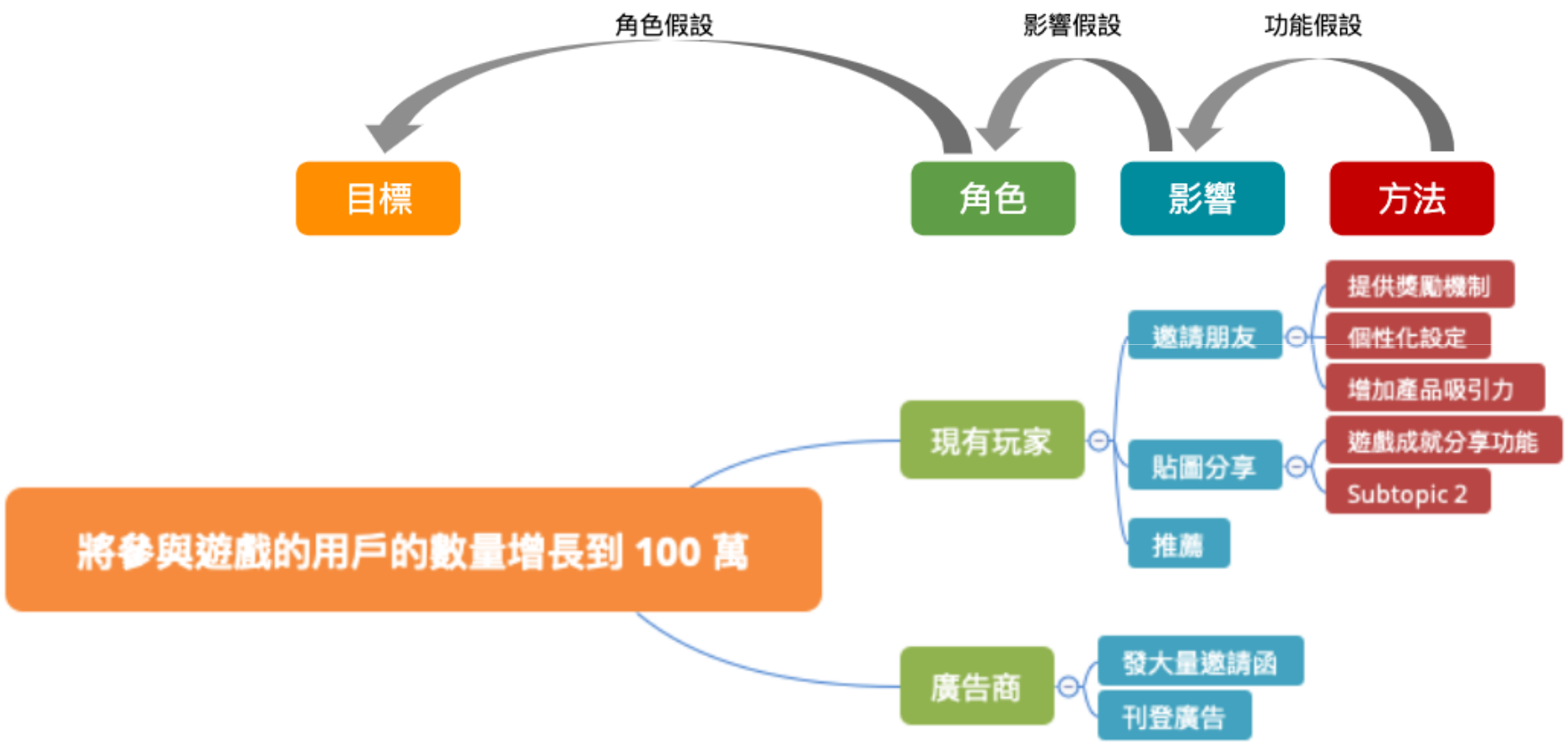
Story Mapping

Example Mapping

Valuable Solution

# Impact Mapping

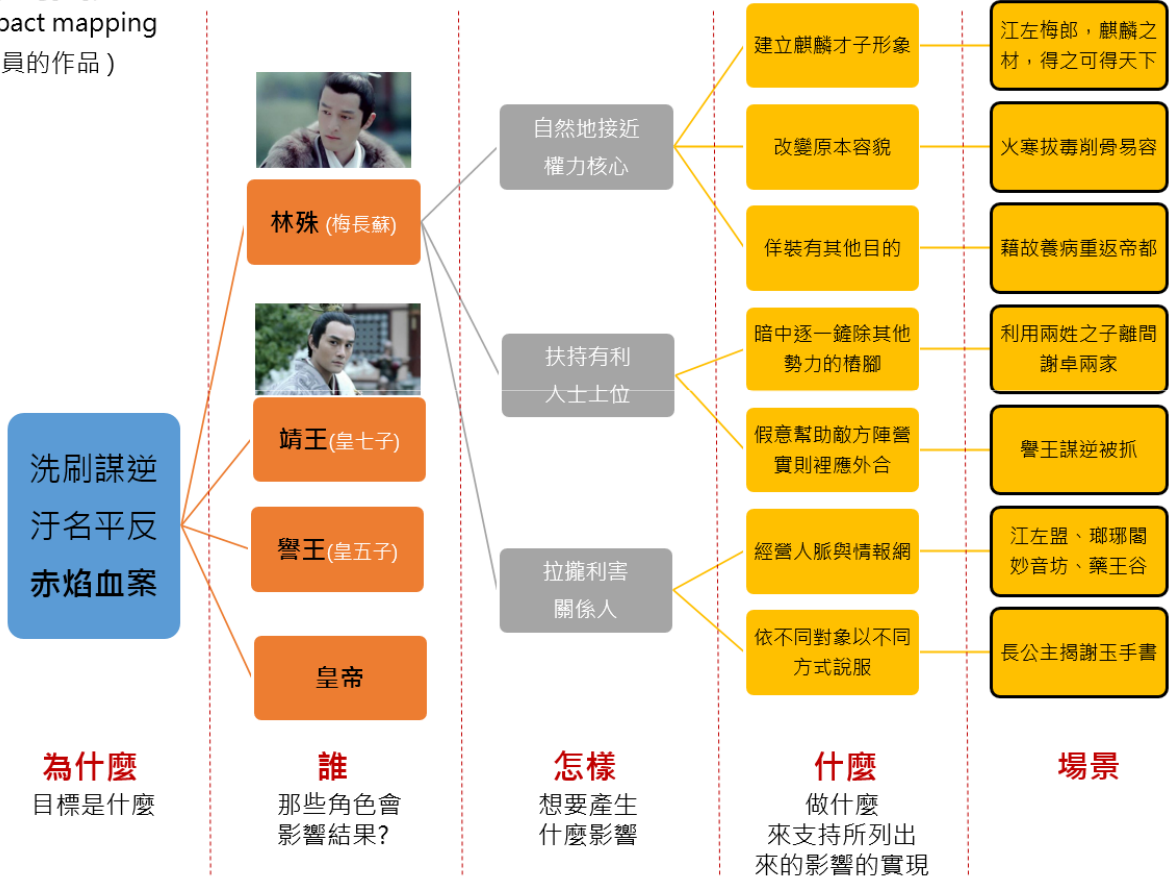




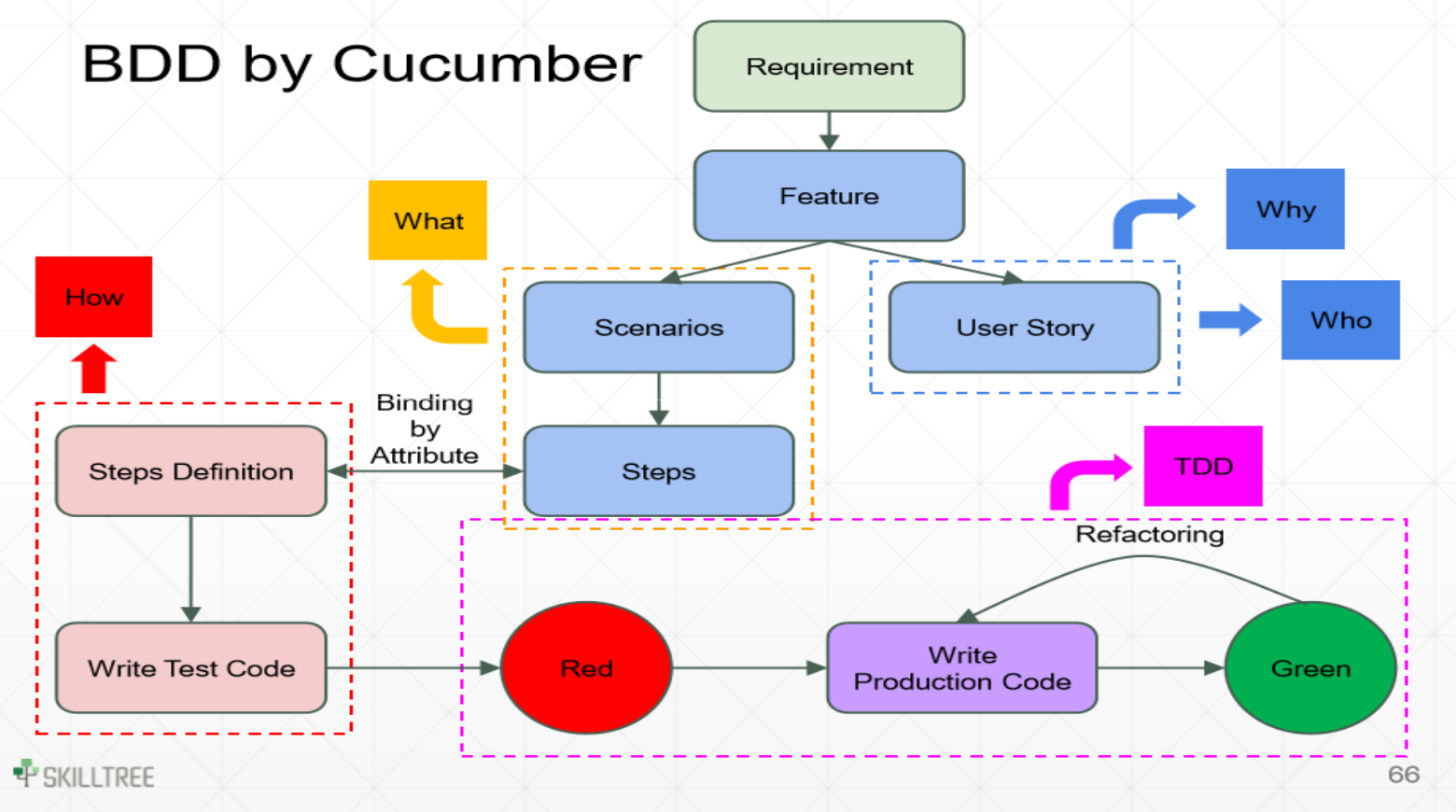
# 瑯琊榜

Impact mapping

( 學員的作品 )



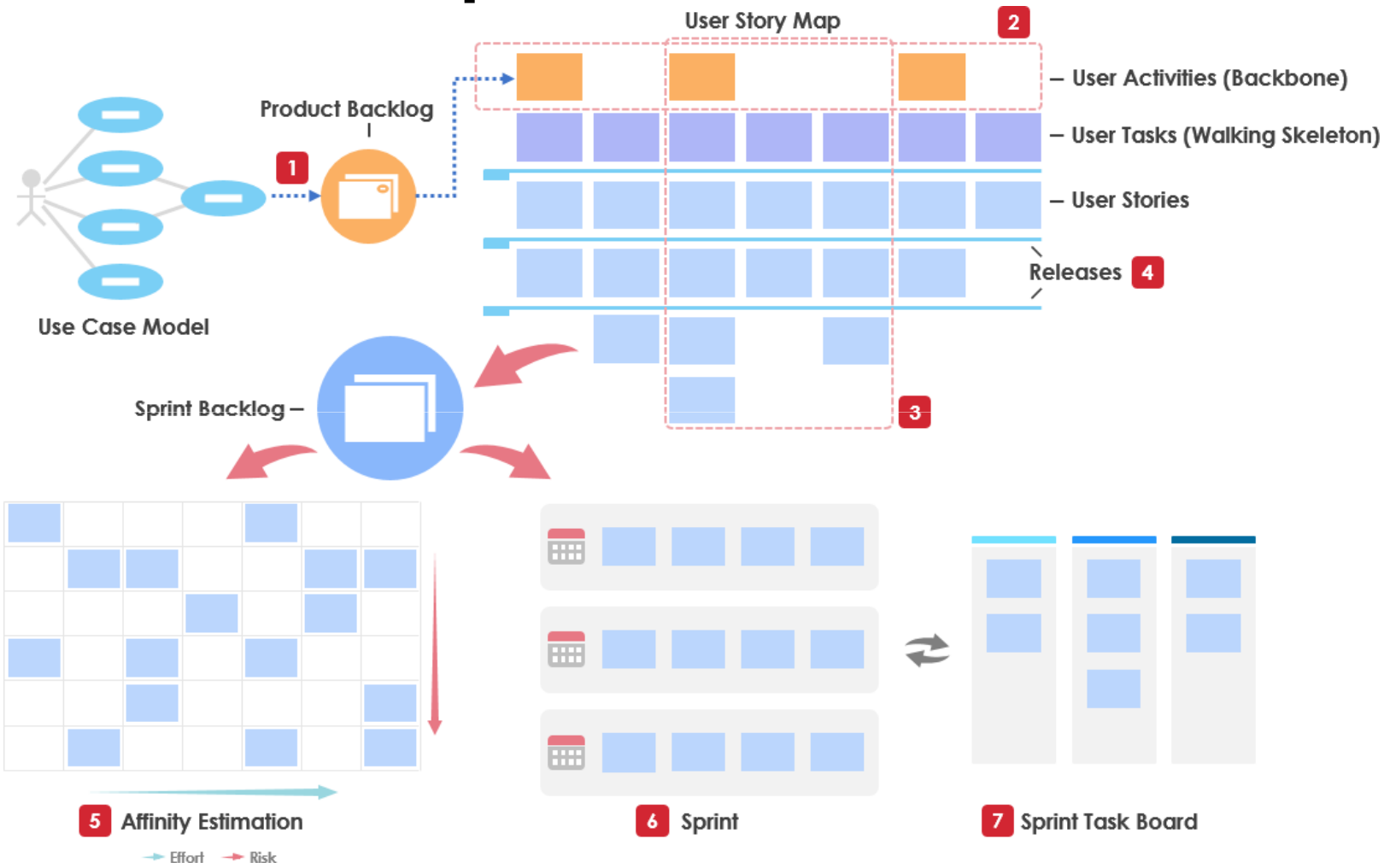
# Cucumber



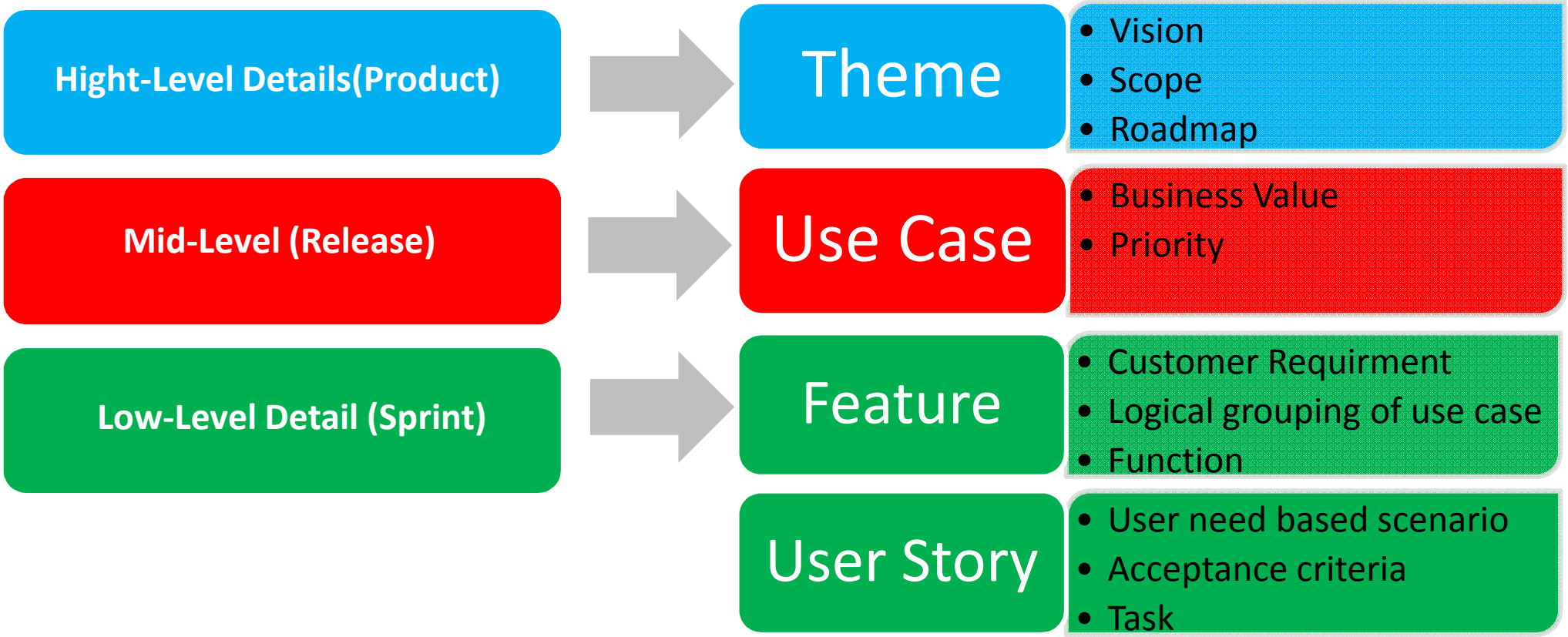

User Story Template:

As a < **role,who** >, I want to **<do something,what** > so that I can **<achieve some benifit ,why**>.

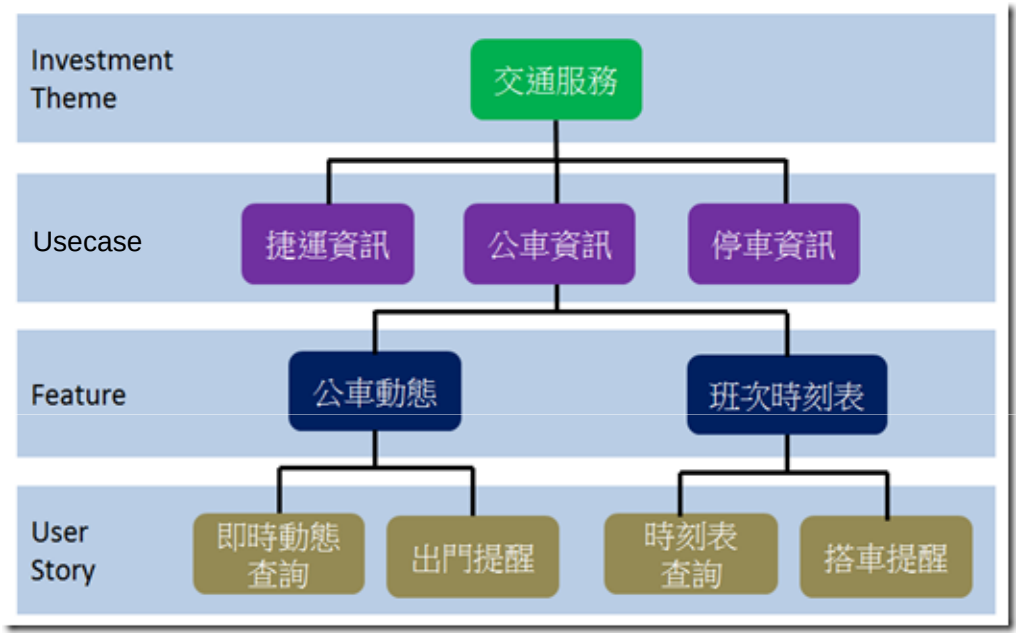
# Usecase to Sprint

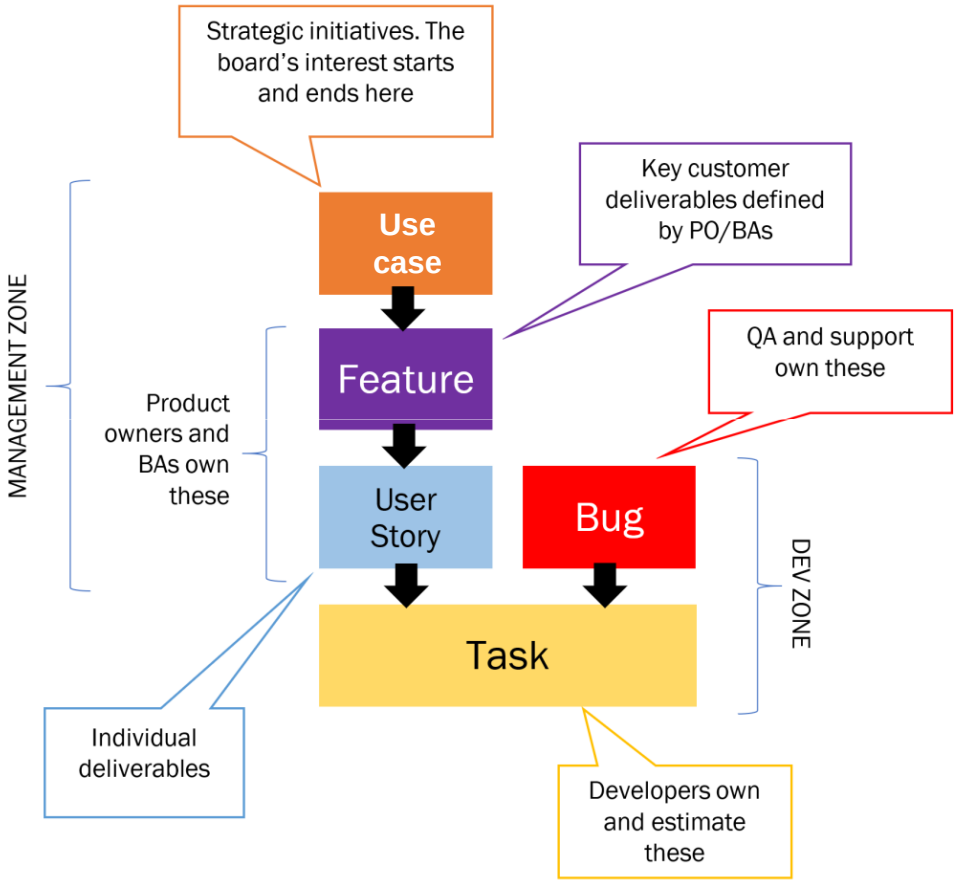


# Level of Requirement

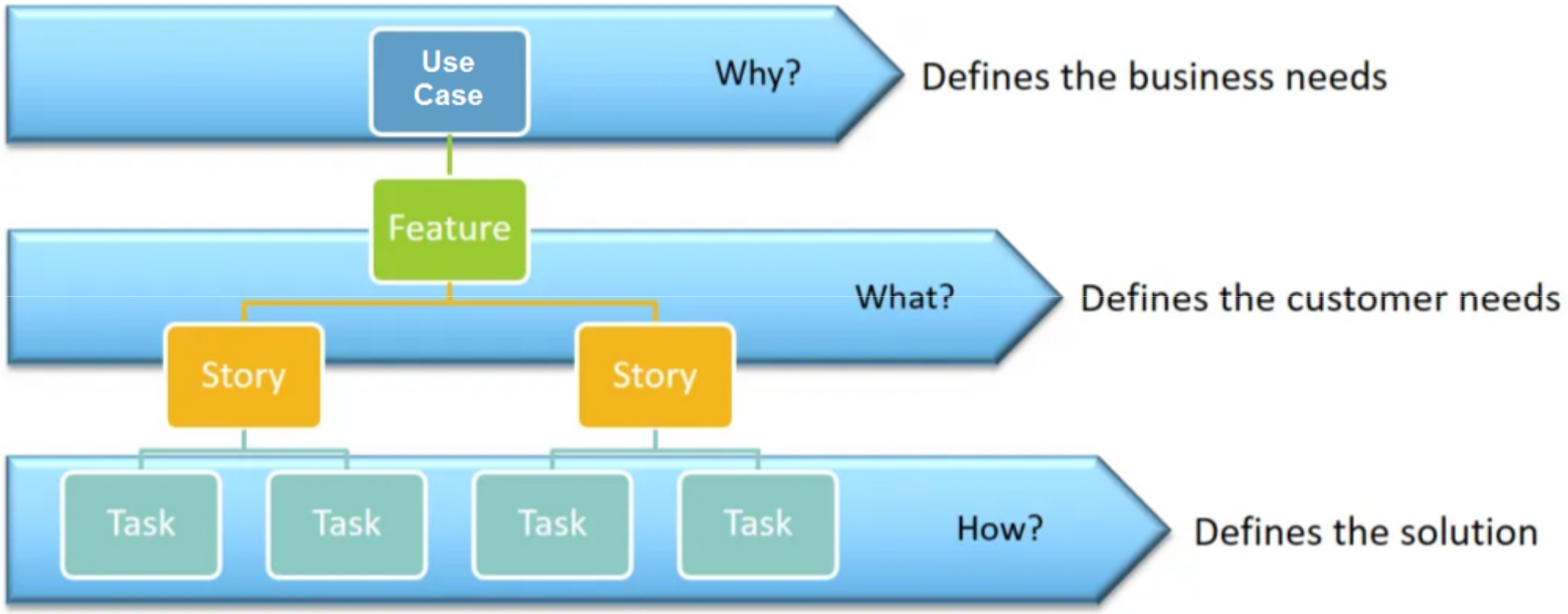


# Scope of REQ

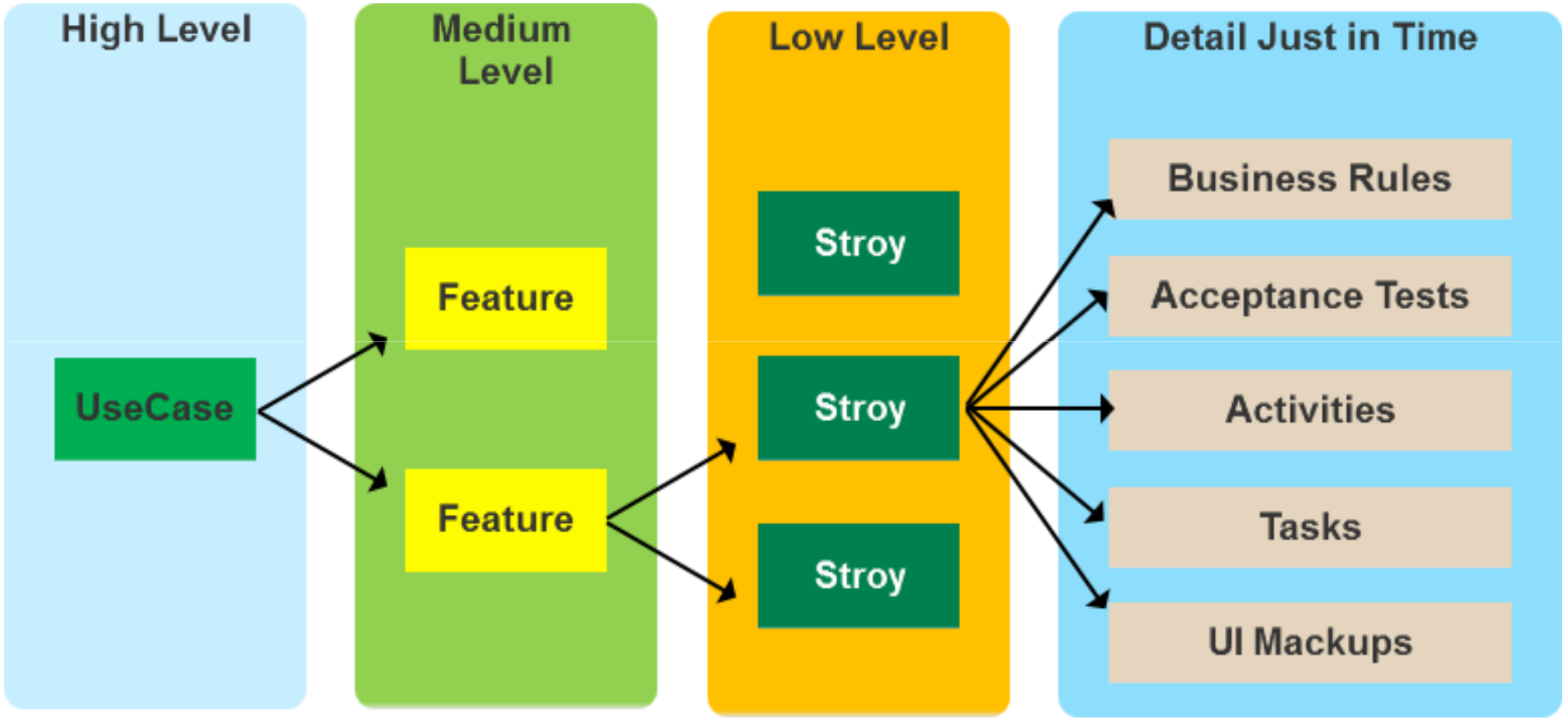

Investment Theme
交通服務
Usecase
捷運資訊
公車資訊
停車資訊
Feature
公車動態
班次時刻表
User Story
即時動態查詢
出門提醒
時刻表查詢
搭車提醒


Strategic initiatives. The board's interest starts and ends here
Key customer deliverables defined by PO/BAs
QA and support own these
MANAGEMENT ZONE
Product owners and BAs own these
Use case
Feature
User Story
Bug
Task
DEV ZONE
Individual deliverables
Developers own and estimate these

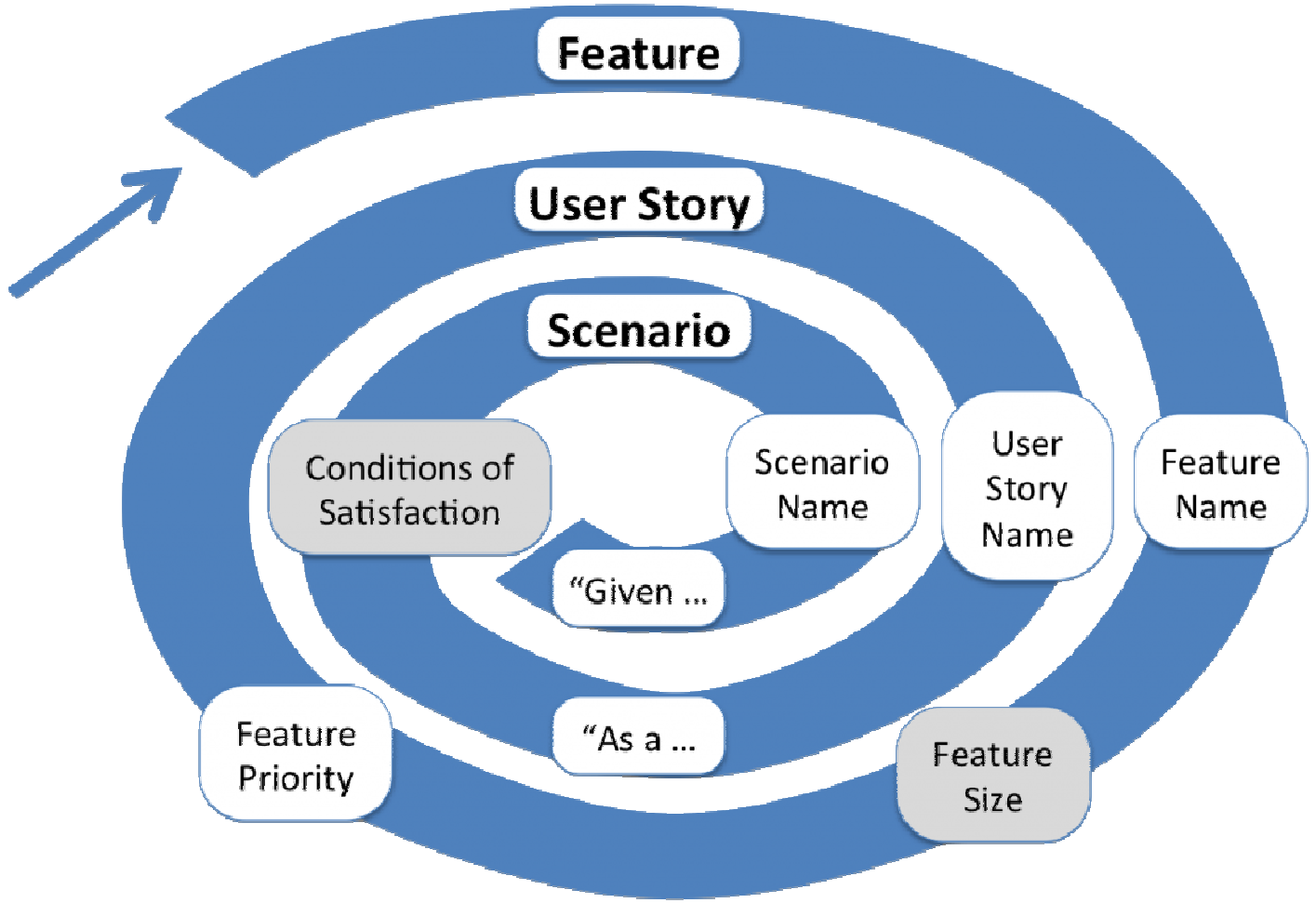
# UseCase TO Tasks



# Requirement SPEC



# Detail Just in Time



# User Story vs Use Case

| User Stories | Use Cases |
| --- | --- |
| As a < **role** >, I want to <**do something**> so that I can <**achieve some benifit**>. | **#Title:** "goal the use case is trying to satisfy"<br>#**Main Success Scenario:** numbered list of steps<br>+**Step:** "**a simple statement of the interaction between the actor and a system**"<br>#**Extensions:** separately numbered lists, one per Extension<br>+**Extension:** "a condition that results in different interactions from .. the main success scenario". An extension from main step 3 is numbered 3a, etc. |

# Use Cases vs User Stories – The Similarity

If we consider the key component in both approaches:

- Use Cases contain equivalent elements: **an actor**, **flow of events** and **post conditions** respectively (a detailed Use Case template may contain many more other elements).

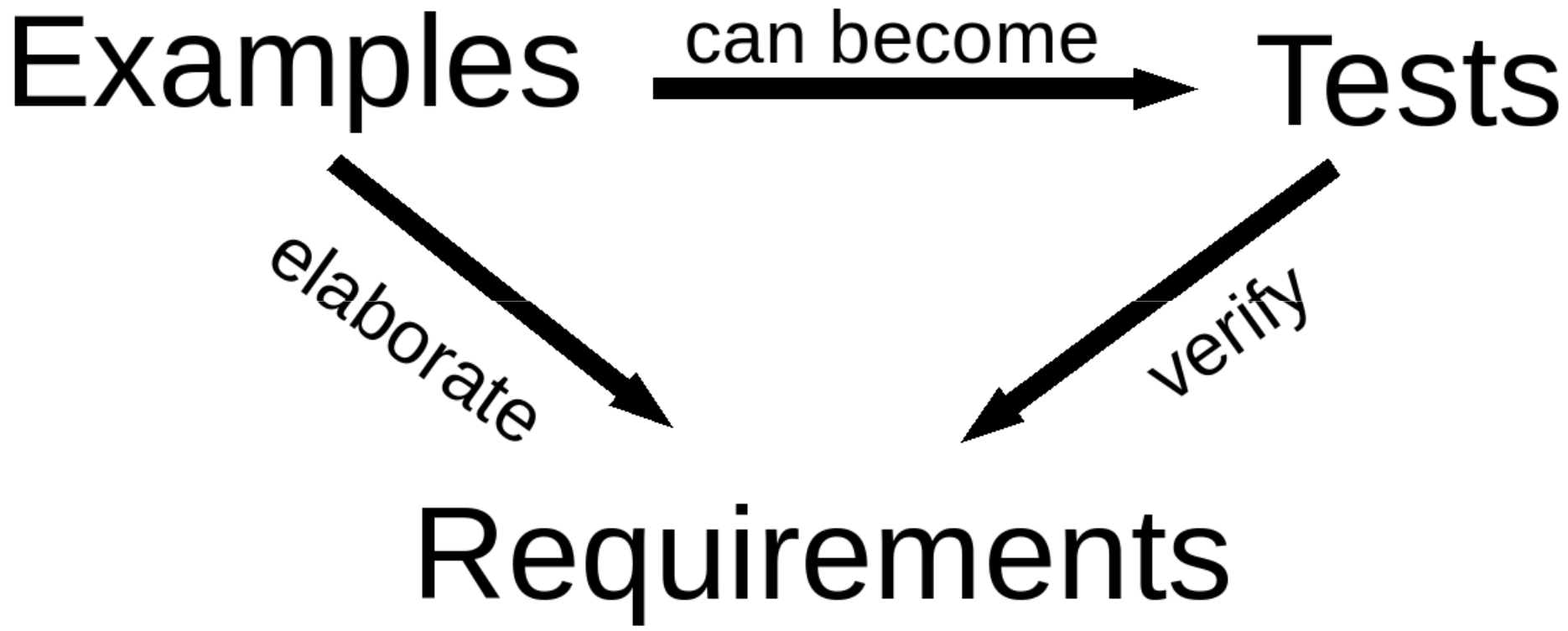
- User Stories contain, with **user role**, **goal** and **acceptance criteria**.

# Use Cases vs User Stories – The Difference

The details of a User Story may not be documented to the same extreme as a Use Case.

- User Stories deliberately **leave out a lot of important details.** User Stories are meant to elicit conversations by **asking questions during scrum meetings**.
- Small increments for **getting feedback more frequently**, rather than having more detailed up-front requirement specification as in Use Cases.

# Specification by example (SBE)



# Requirement Description

**Feature**

Serve coffee

→

**User Story**

In order to earn money Customers should be able to buy coffee at all times

**Template:**

As a < **role** >, I want to <**do something**> so that I can <**achieve some benifit**>.

→

**Scenario**

**Scenario:** Buy last coffee

**Given** there are 1 coffees left in the machine

**And** I have deposited 1 dollar

**When** I press the coffee button

**Then** I should be served a coffee

**Template:**

A simple statement of the interaction between the actor and a system

# Specification by Example

Feature: 訂單結案時獲得紅利點數 # Title

身為一名銷售主管 # Description

我想要讓顧客在訂單結案後獲得點數

這樣就能提供誘因繼續再我們的商城消費

**Scenario**: 普通等級客戶可以獲得 10 點

**Given** 我是有一名普通等級客戶

**And** 我有一筆待結案訂單

**When** 我的訂單結案時

**Then** 我應該獲得 10 點紅利點數

**Scenario**: VIP 等級客戶可以獲得 20 點

**Given** 我是有一名 VIP 等級客戶

**And** 我有一筆待結案訂單

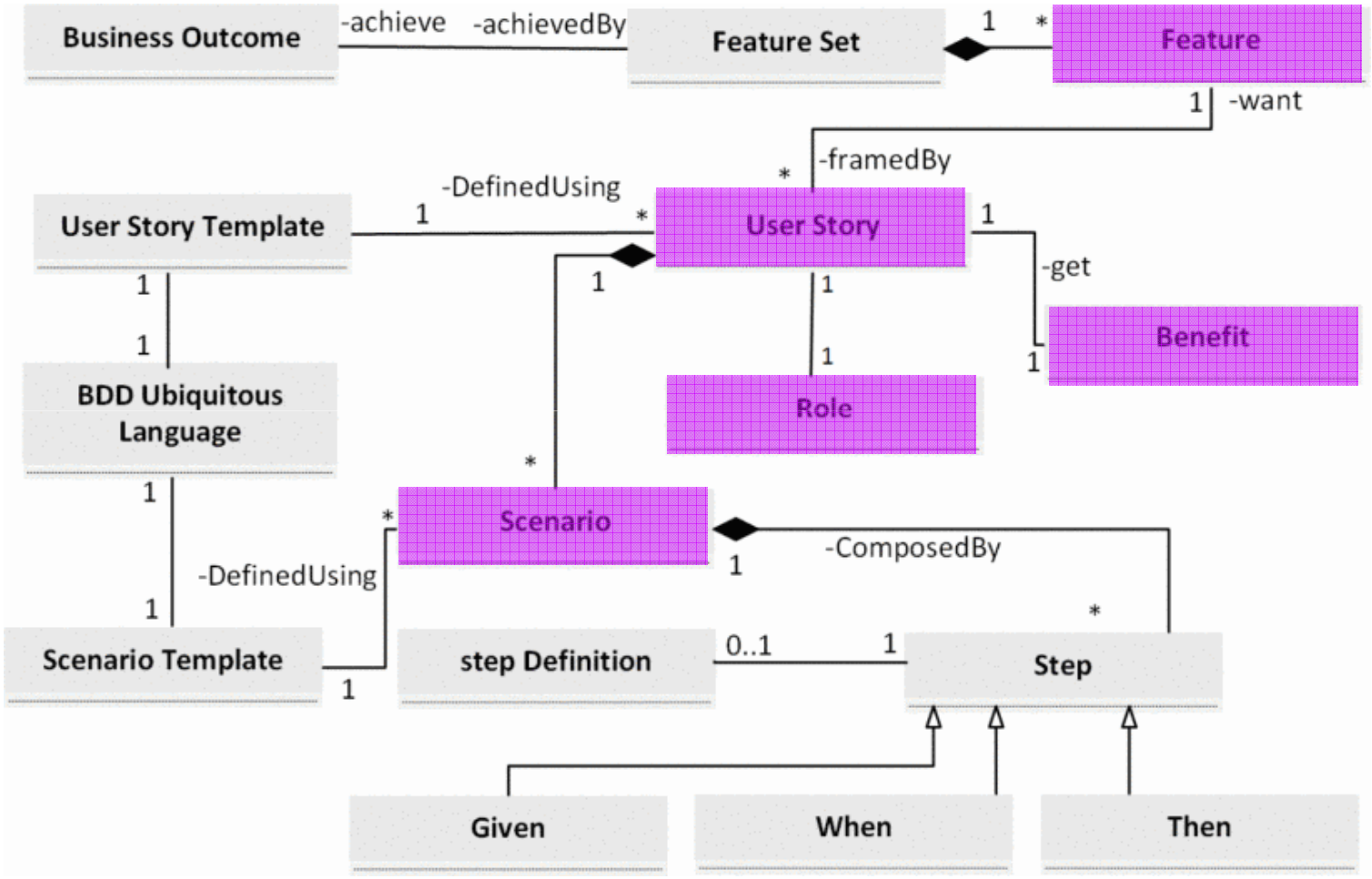
**When** 我的訂單結案時

**Then** 我應該獲得 20 點紅利點數

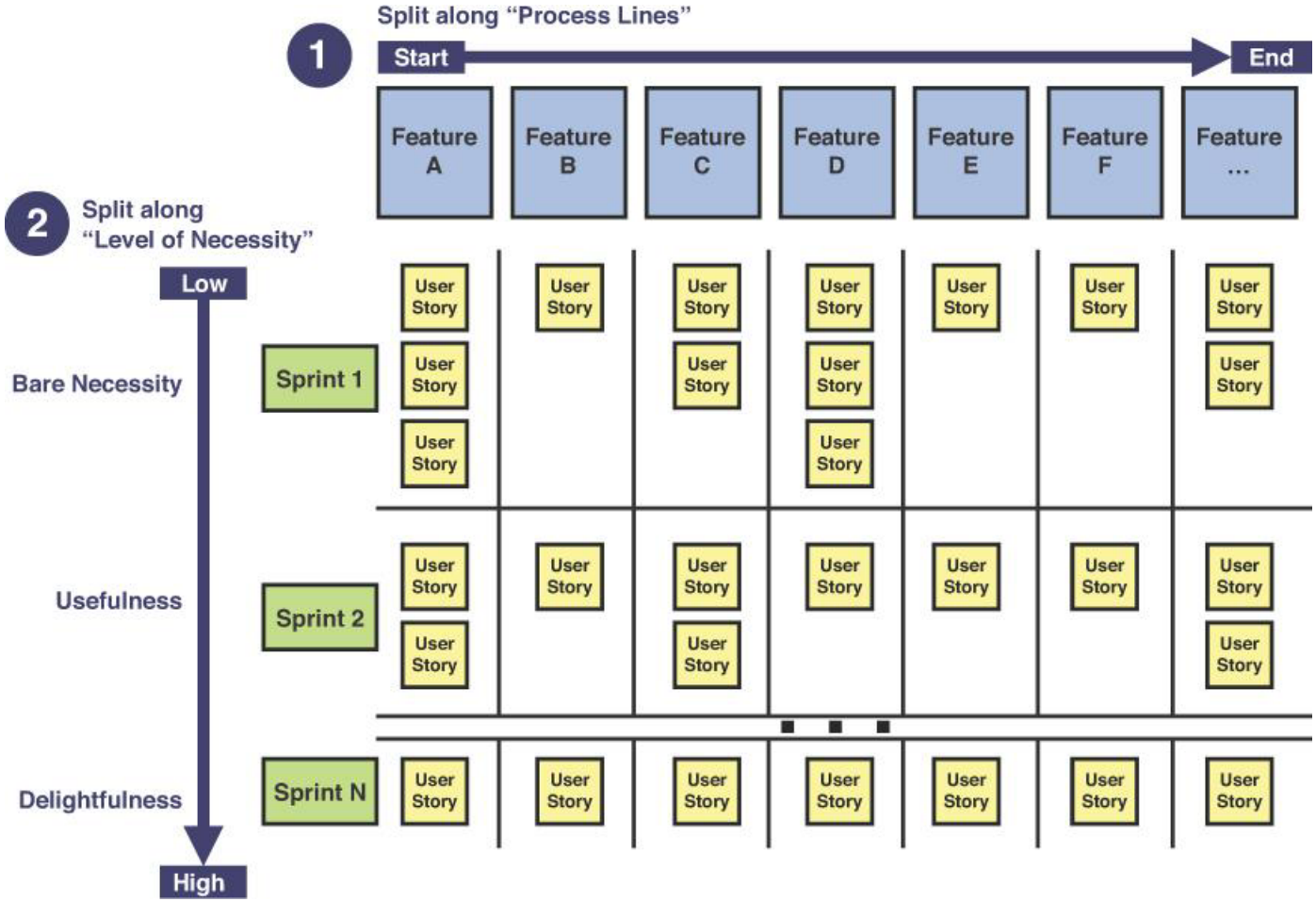
- ...

```
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario Outline: Add two numbers
    Given I have entered <input_1> into the calculator
    And I have entered <input_2> into the calculator
    When I press <button>
    Then the result should be <output> on the screen

  Examples:
    | input_1 | input_2 | button | output |
    | 20      | 30      | add    | 50     |
    | 2       | 5       | add    | 7      |
    | 0       | 40      | add    | 40     |
```

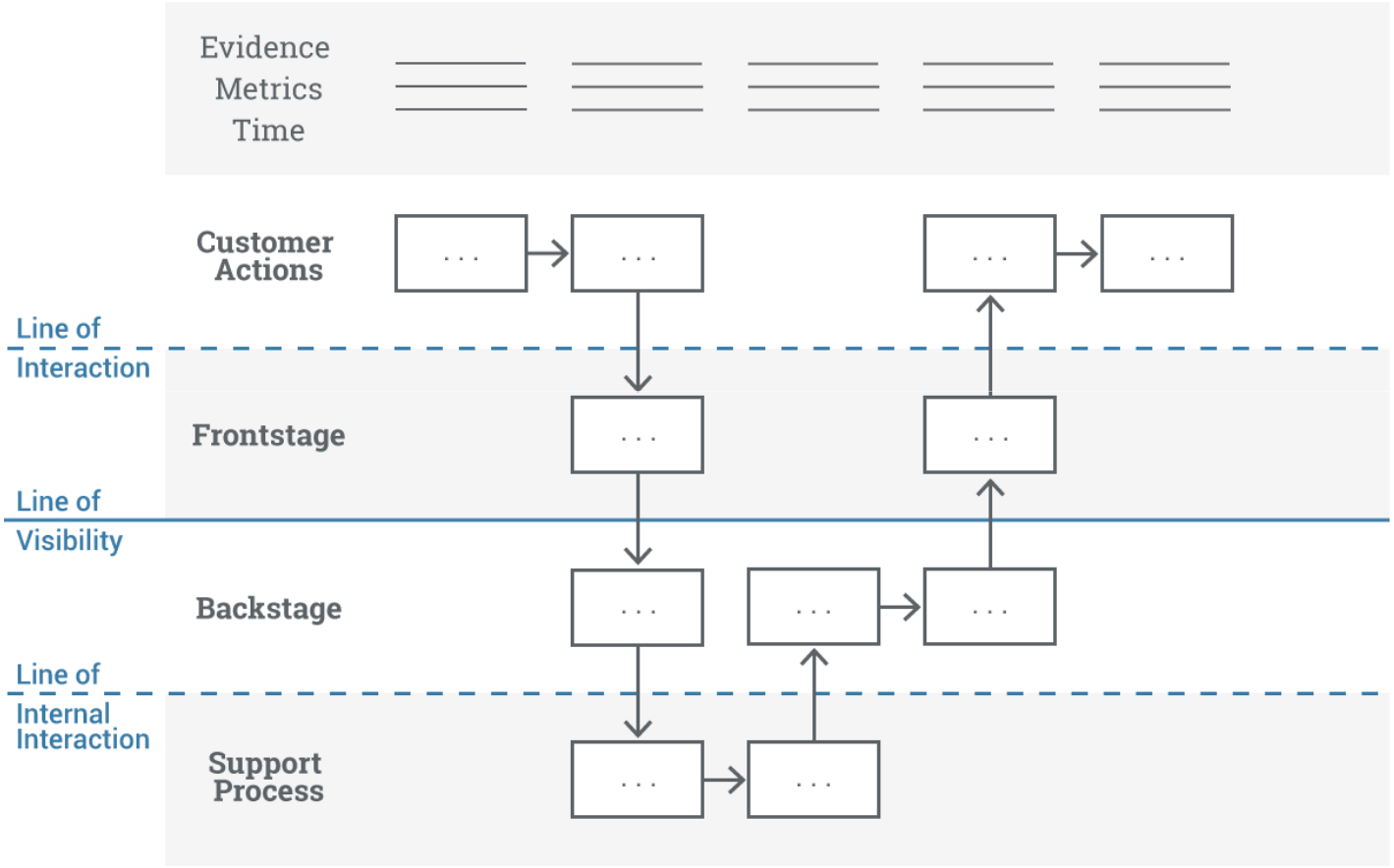
# BDD metamodel



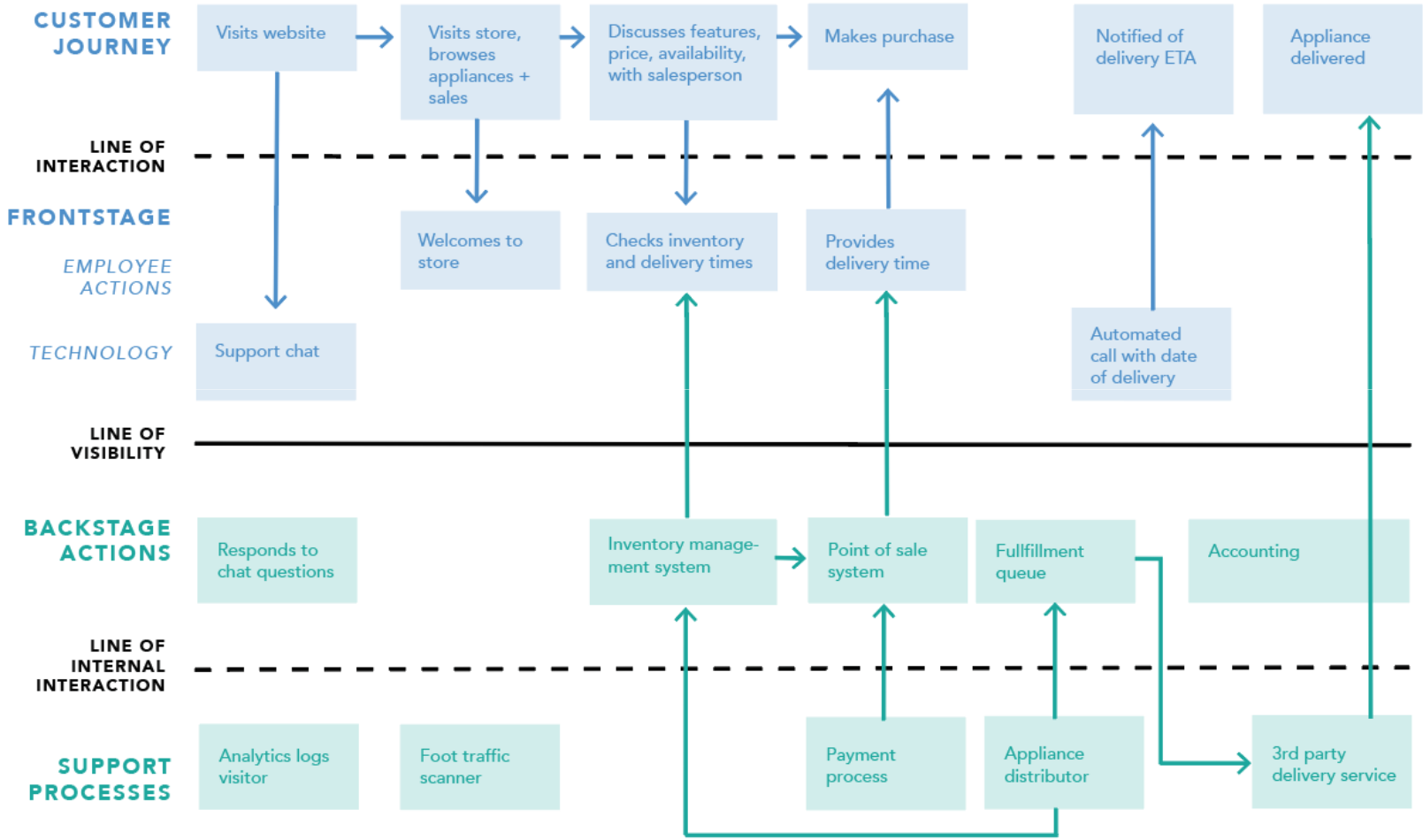
# User Story Mapping



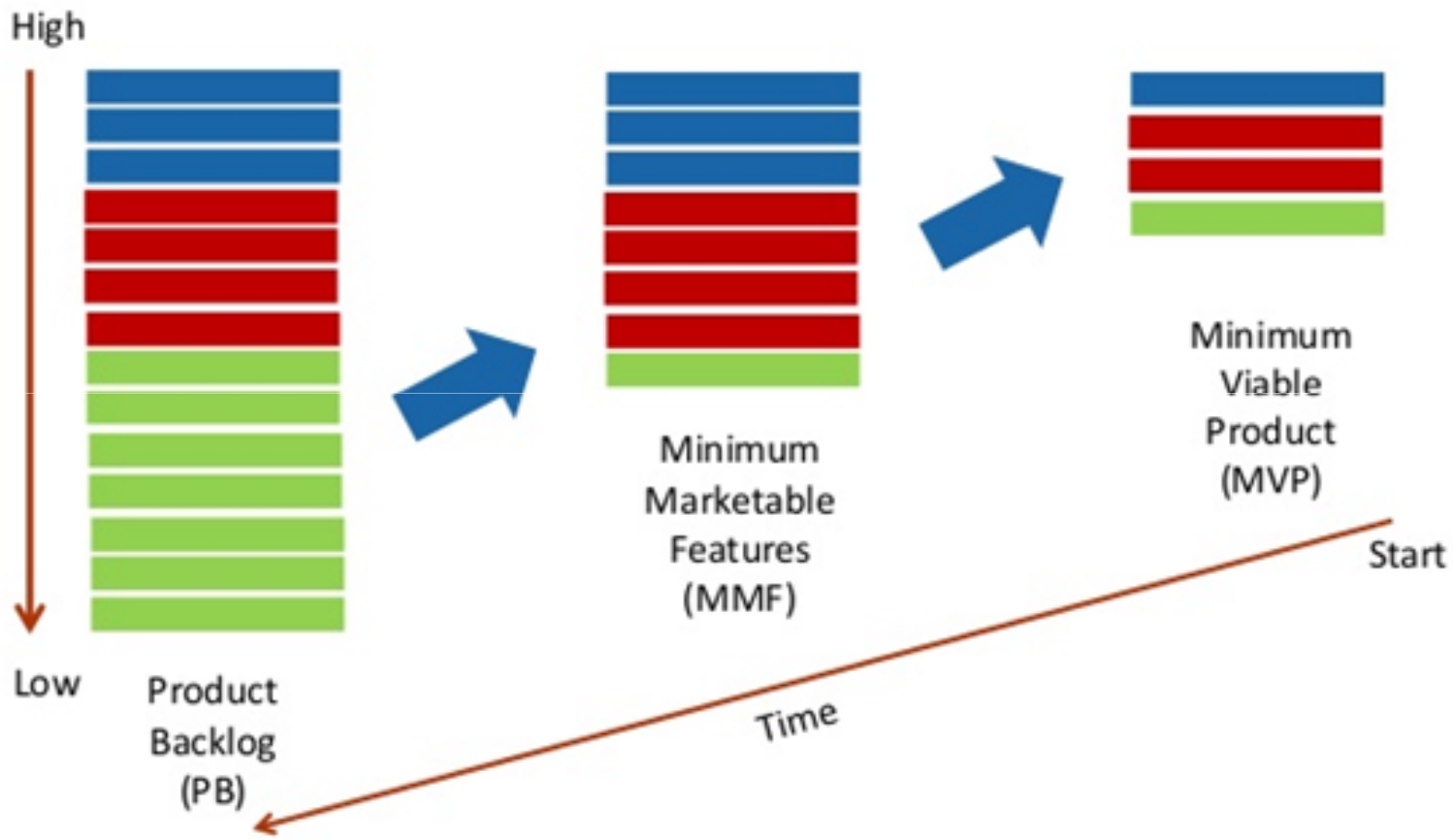
# Service blueprint



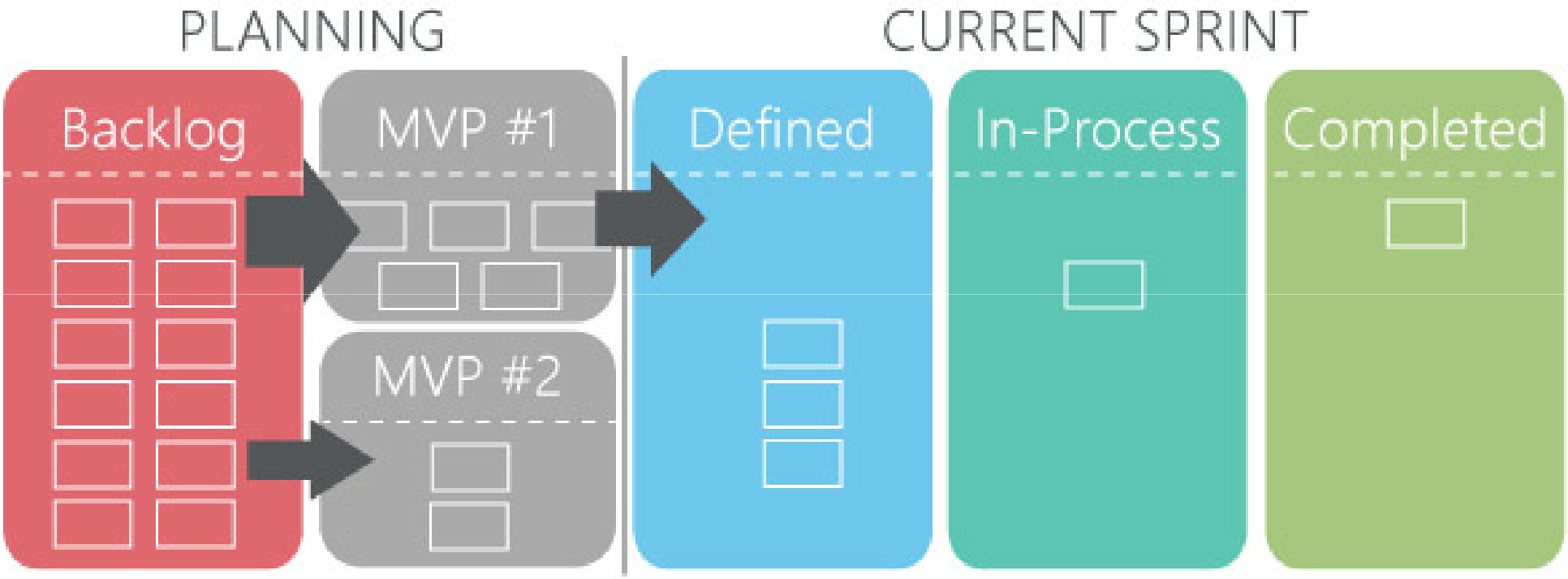
# Service blueprint



# MVP vs MMF vs Product Backlog

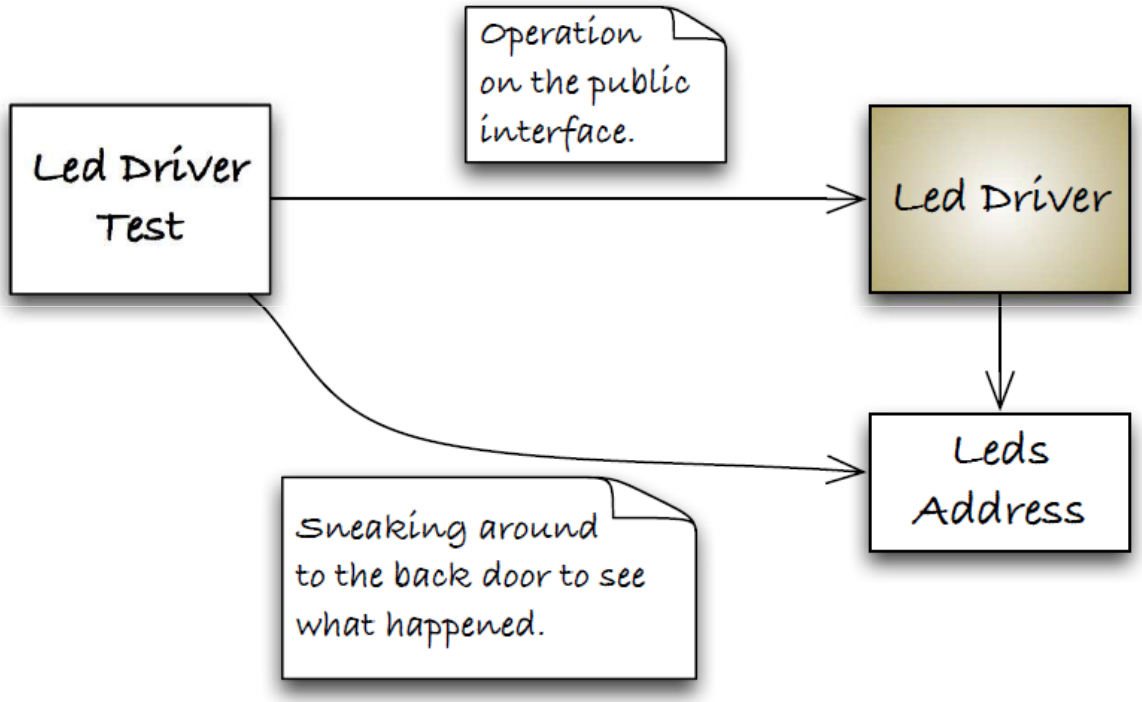


# Backlog to MVP



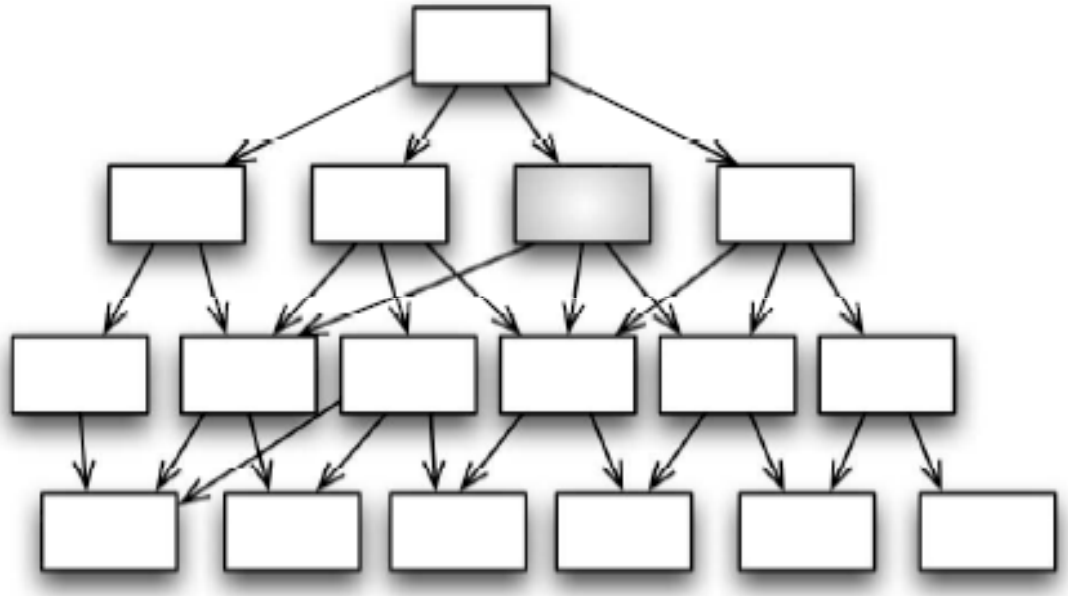
# Test Double

# Collaborators(DOC)

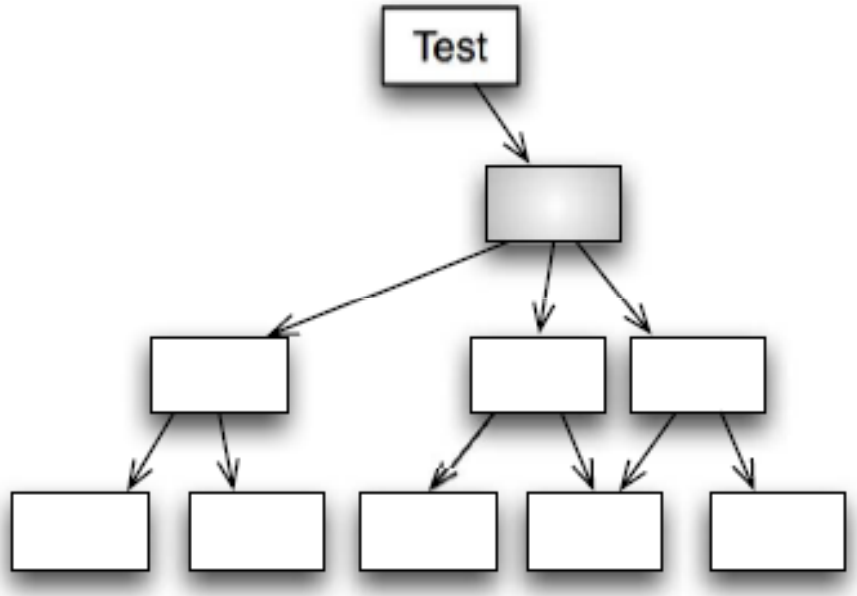



- A collaborator is some function, data, module, or device outside the code under test (CUT) that the CUT depends upon.
- **DOC** : Depended on Components

# Dependency mess

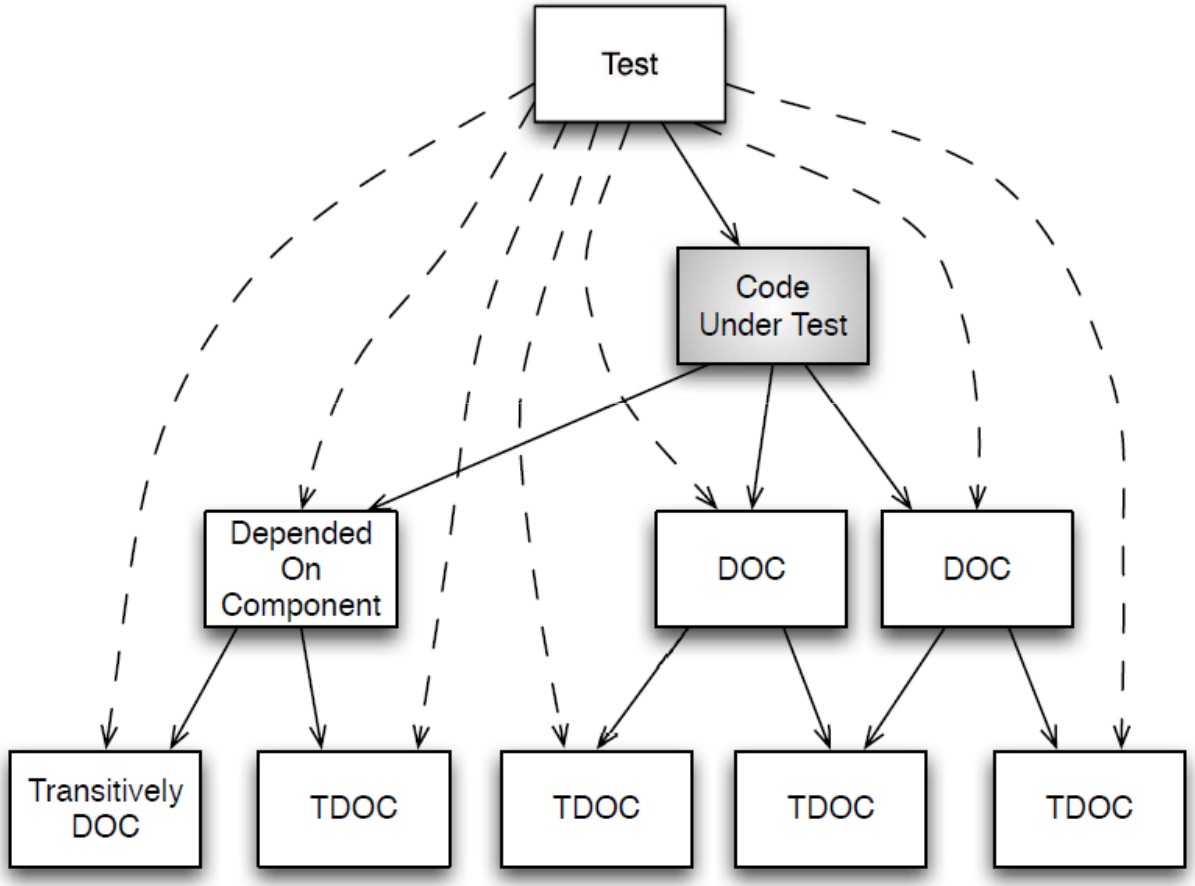


The module in the middle...



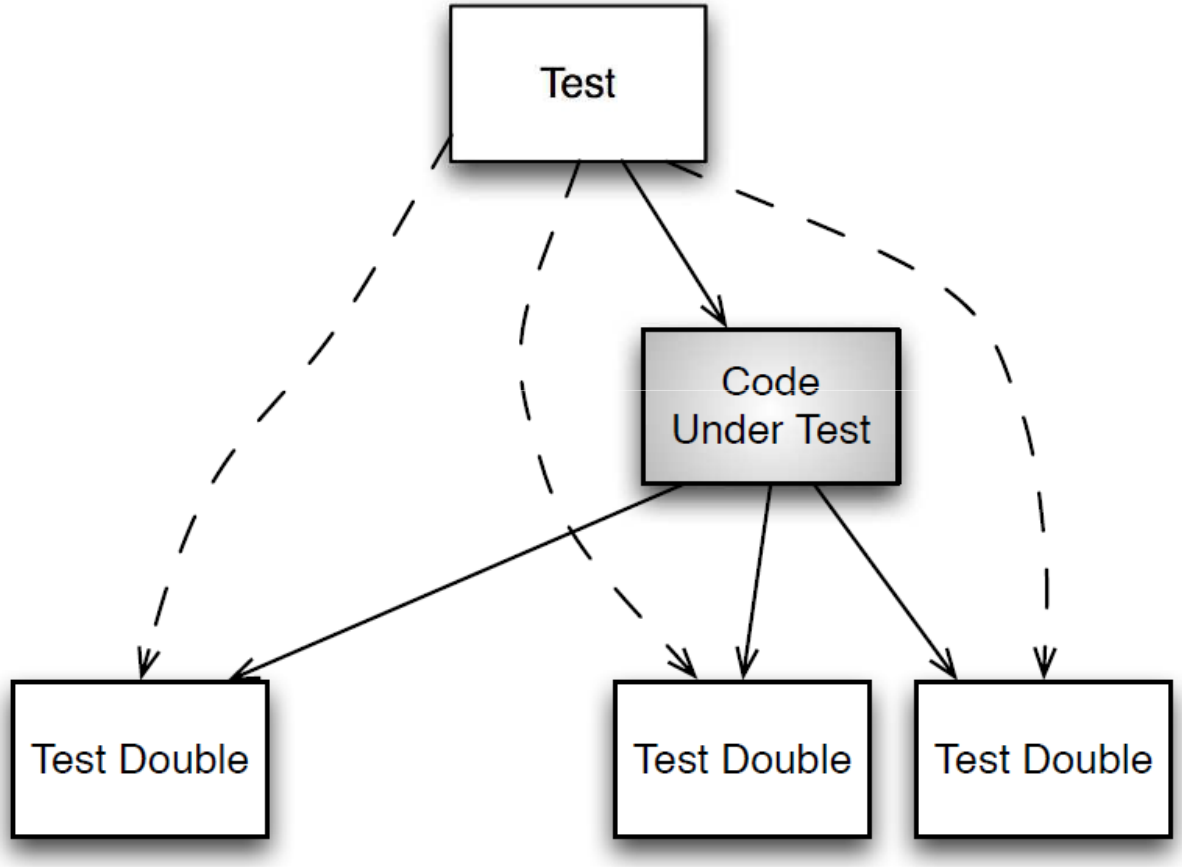

...has baggage.

# Test dependency tree

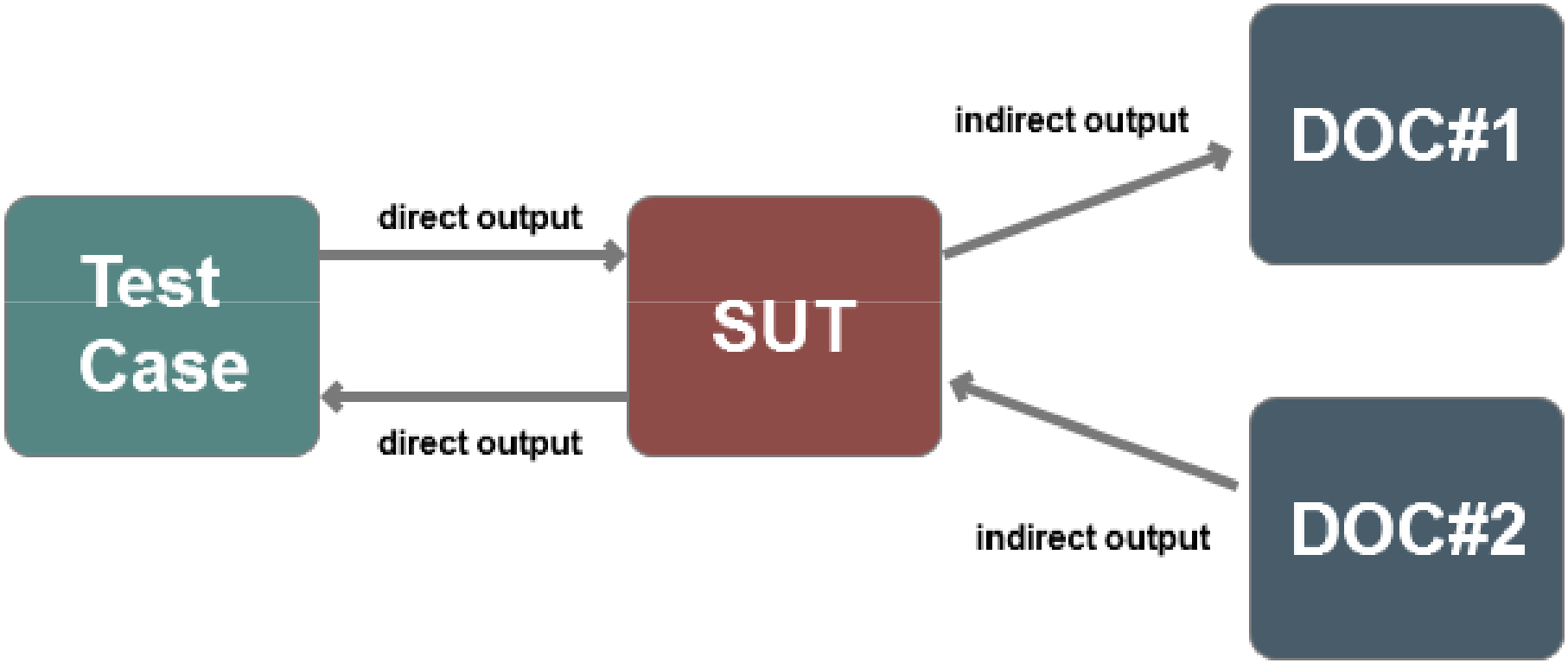



- **SUT** : System Under Test
- **CUT** : Code Under Test
- **DOC** : Depended on Components

# Breaking Dependencies



# Input and Output

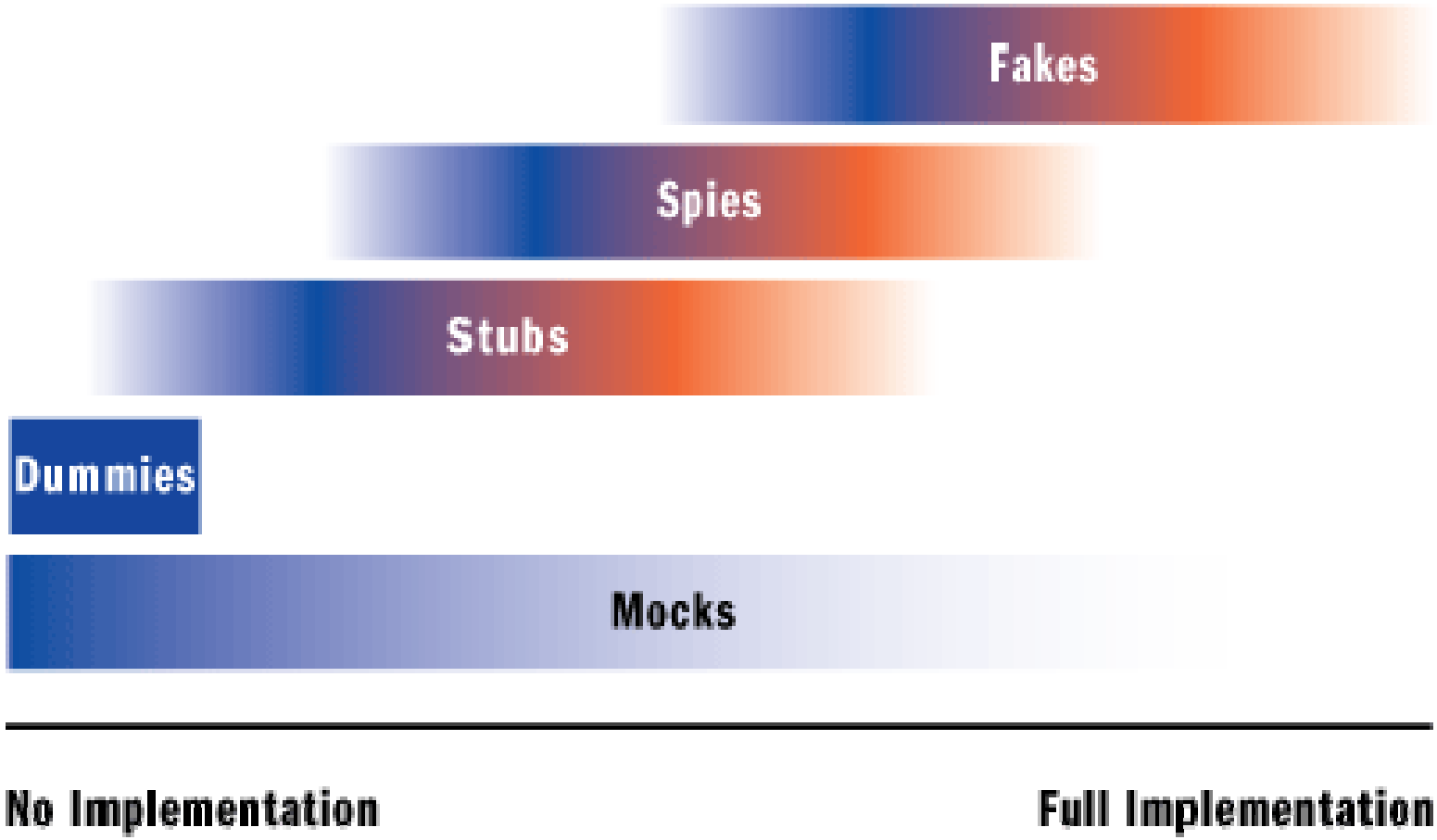


# Test Doubles Variations

| Name | Variation |
|---|---|
| Test dummy | Keeps the linker from rejecting your build. A dummy is a simple stub that is never called. It is provided to satisfy the complier, linker, or runtime dependency. |
| Test stub | Returns some value, as directed by the current test case. |
| Test spy | Captures the parameters passed from the CUT so the test can verify that the correct parameters have been passed to the CUT. The spy can also feed return values to the CUT just like a test stub. |
| Mock object | Verifies the functions called, the call order and the parameters passed from the CUT to the DOC. It also is programmed to return specific values to the CUT. The mock object is usually dealing with a situation where multiple calls are made to it, and each call and response are potentially different. |
| Fake object | Provides a partial implementation for the replaced component. The fake usually has a simplified implementation when compared to the replaced implementation. |
| Exploding fake | Causes the test to fail if it is called. |

- Mock maintenance costs are quite high. In practice, **Stub: Mock** is about 90%: 10%. **Stub** is used in most cases.

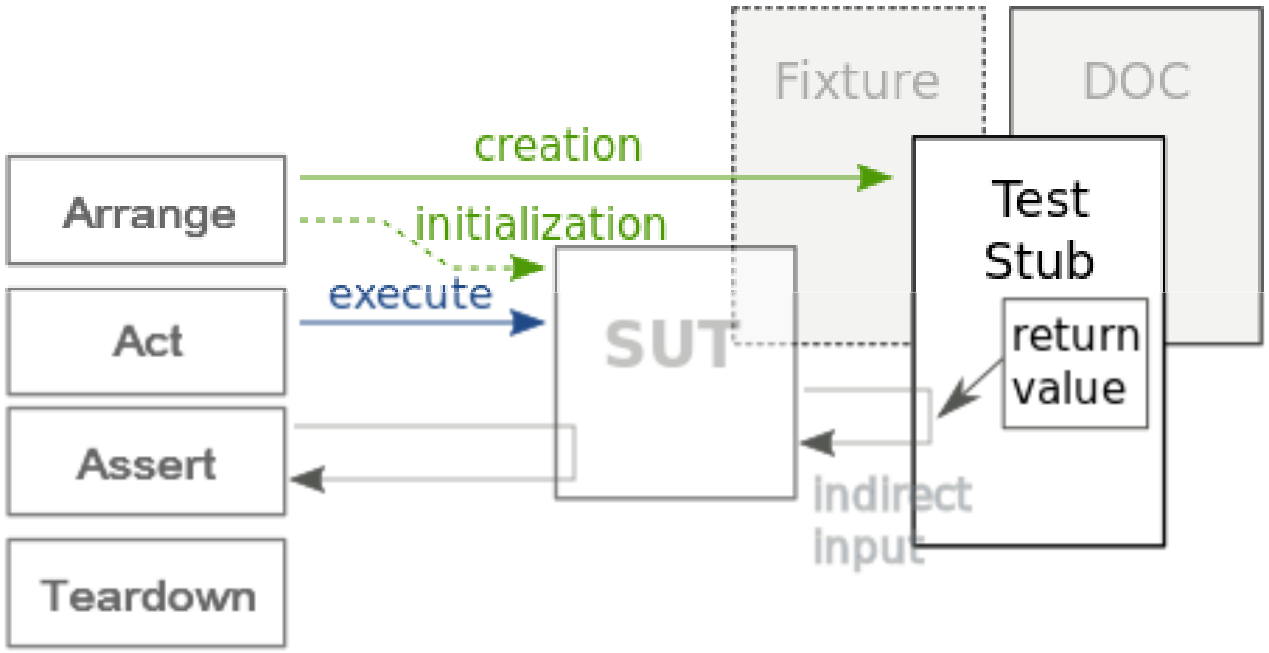
# Spectrum of Test Doubles



# When to Use aT est Double

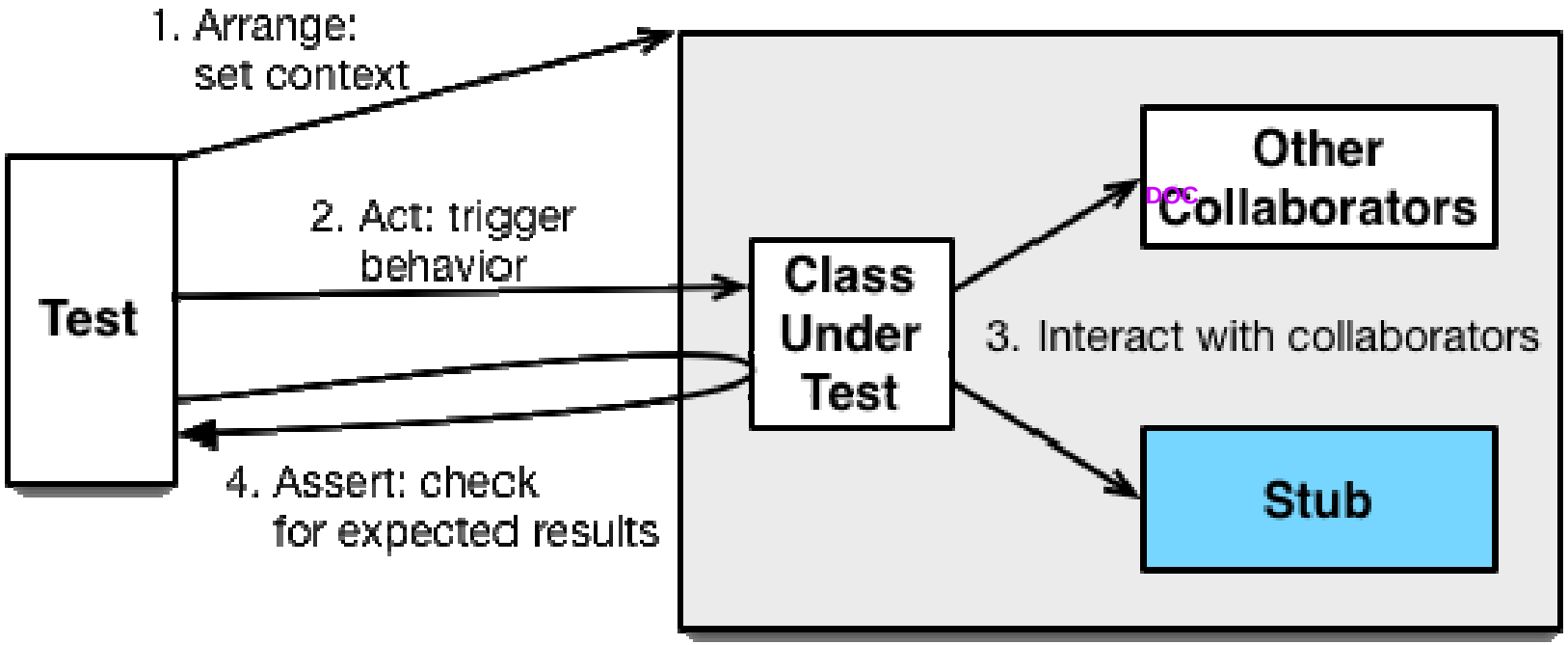
- **Hardware independence**
- Inject **difficult to produce input(s)**
- Speed up a **slow collaborator**
- Dependency on something **unstable**
- Dependency on something **under development**
- Dependency on something that is **difficult to configure**

# Test Stub

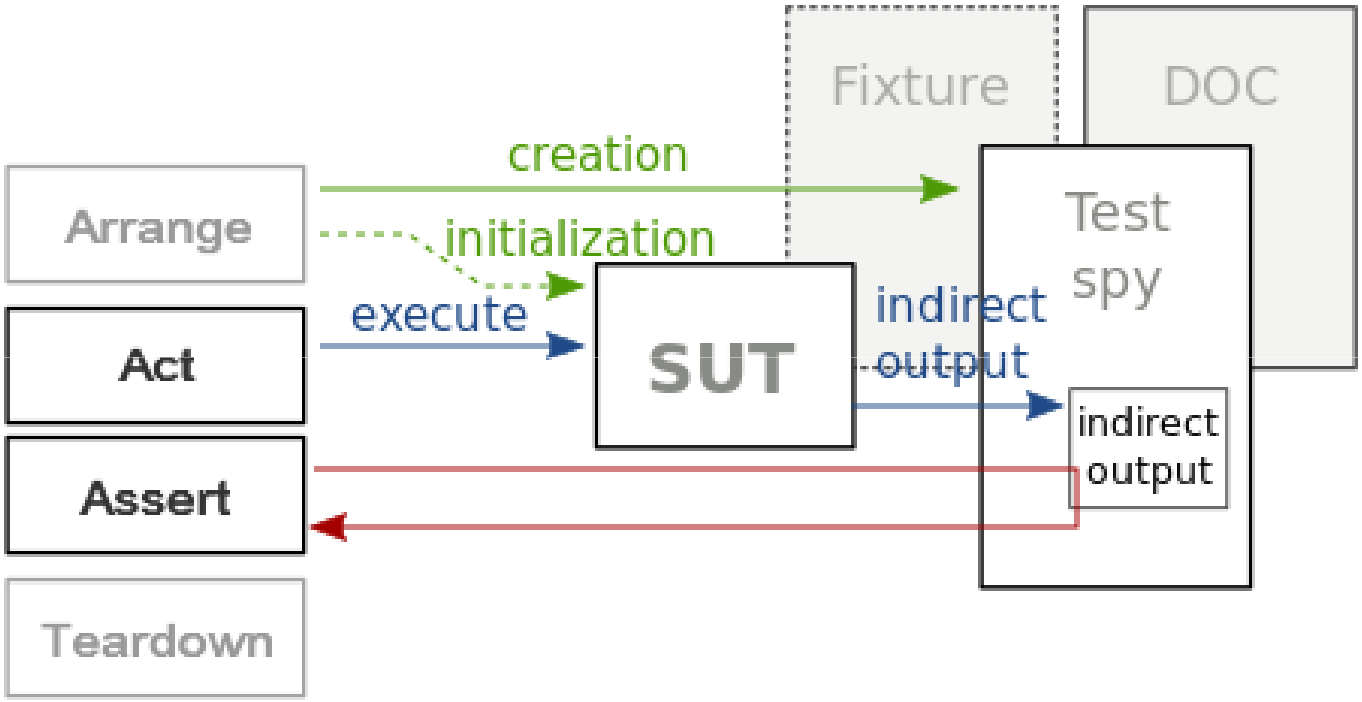



We replace a real object with a test-specific object that **feeds the desired indirect inputs** into the **SUT**.

# Test Stub

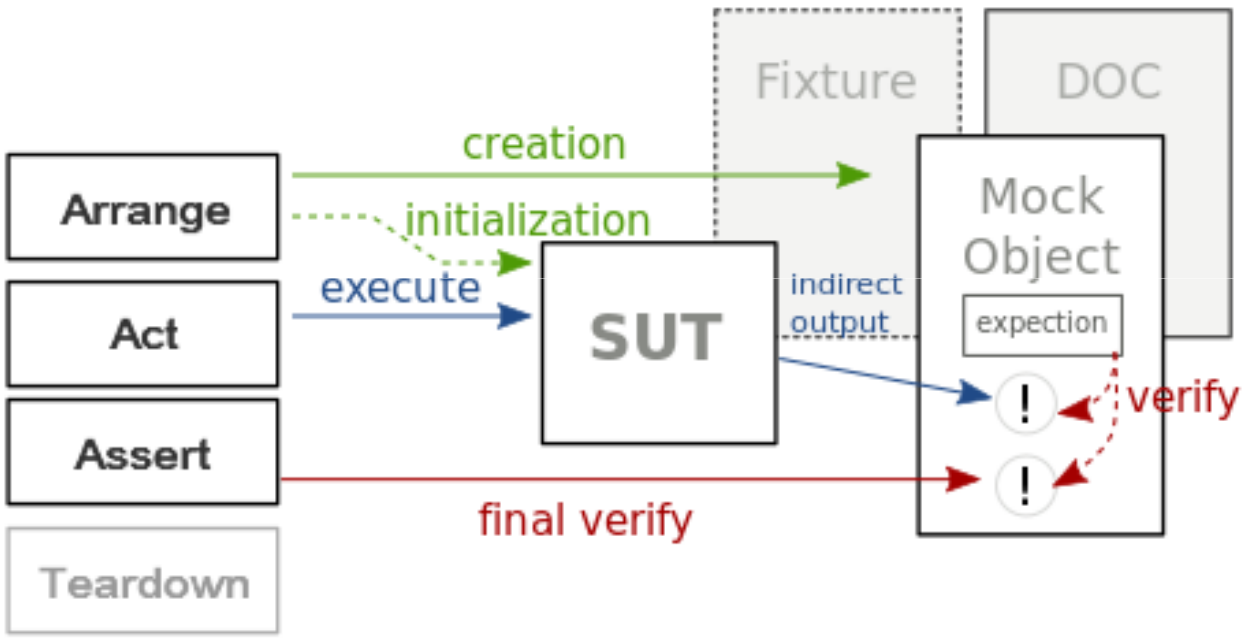

1. Arrange:
set context
Test
2. Act: trigger
behavior
4. Assert: check
for expected results
Class
Under
Test
Other
Collaborators
3. Interact with collaborators
Stub

# Test Spy



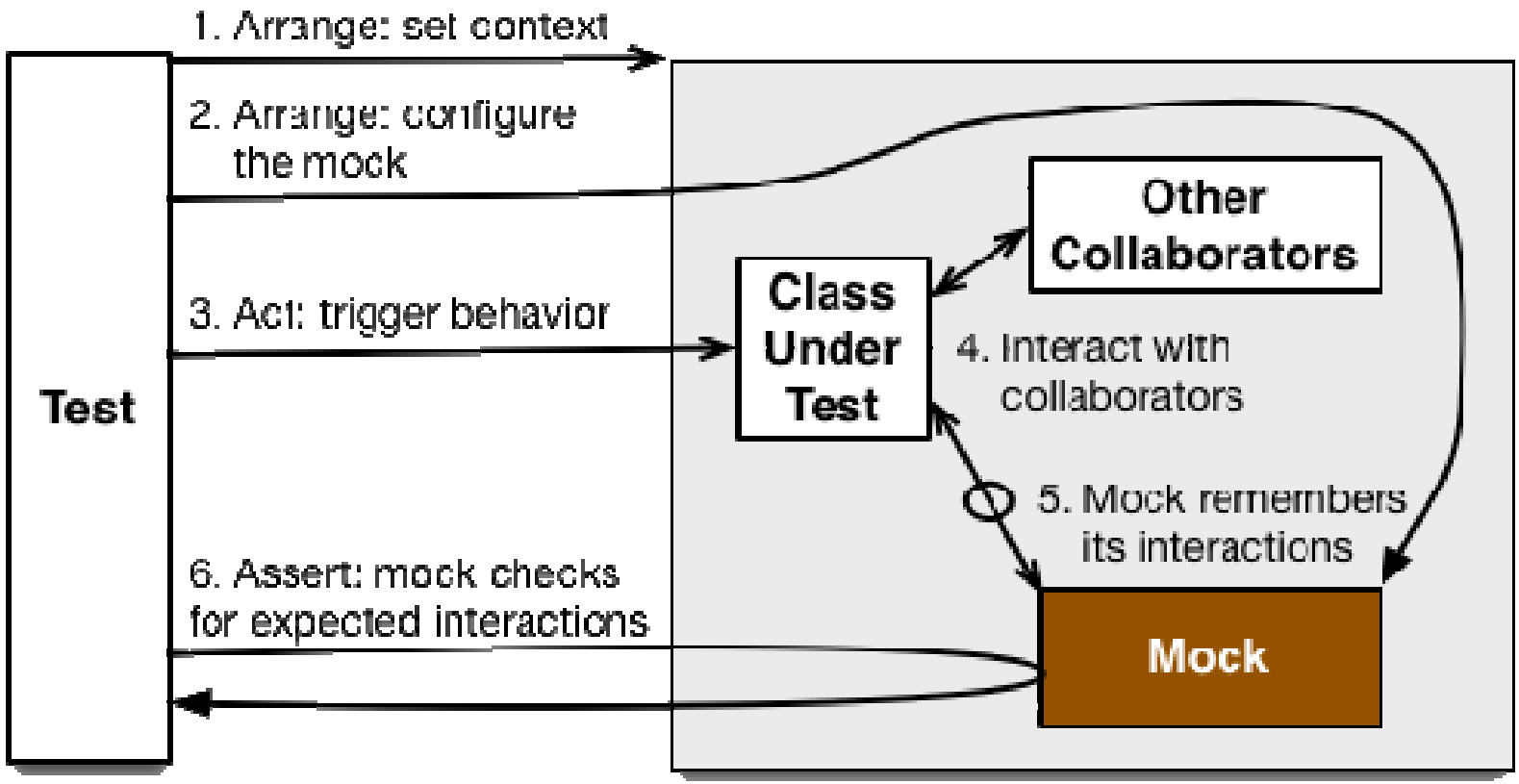

Use a Test Double to capture the **indirect output calls** made to another component by the **SUT** for later verification by the test.

# Mock Object

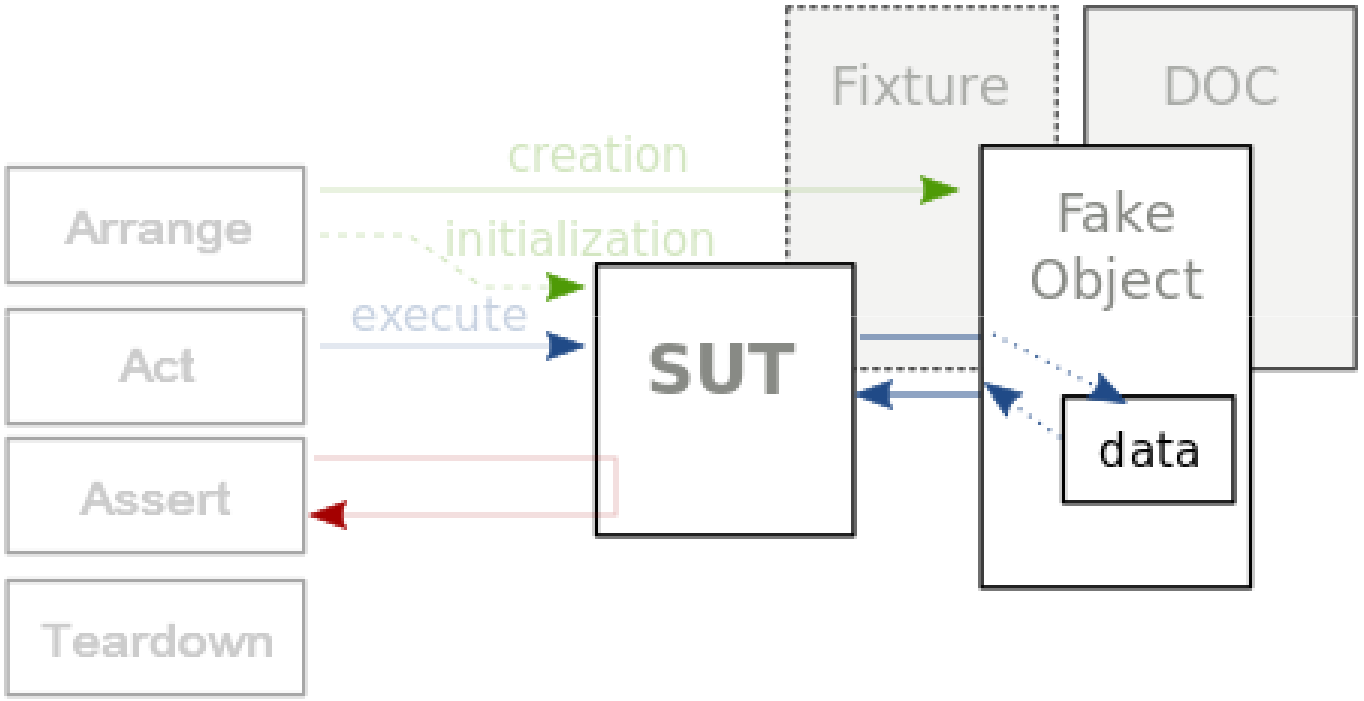



Replace an object the **SUT** depends on with a test-specific object that verifies **it is being used correctly** by the SUT.

# Mock Object

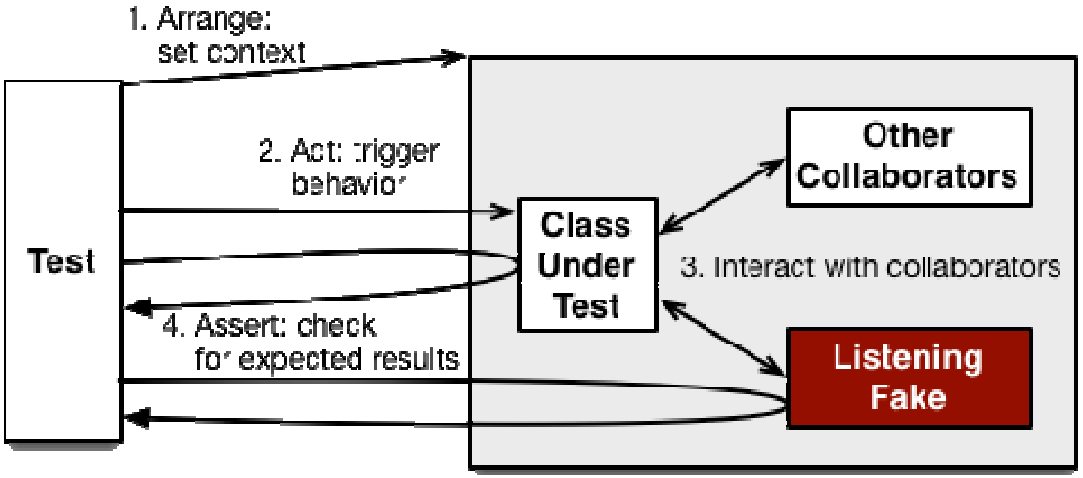

1. Arrange: set context
2. Arrange: configure the mock
3. Act: trigger behavior
4. Interact with collaborators
5. Mock remembers its interactions
6. Assert: mock checks for expected interactions
Test
Class Under Test
Other Collaborators
Mock

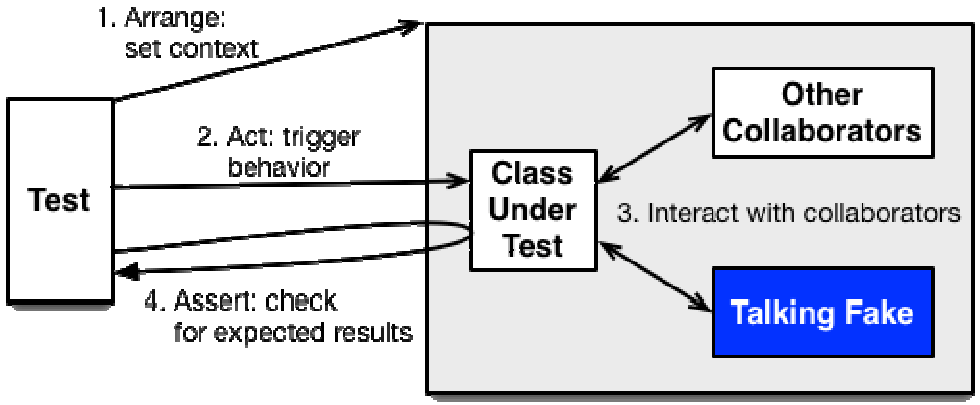
# Fake



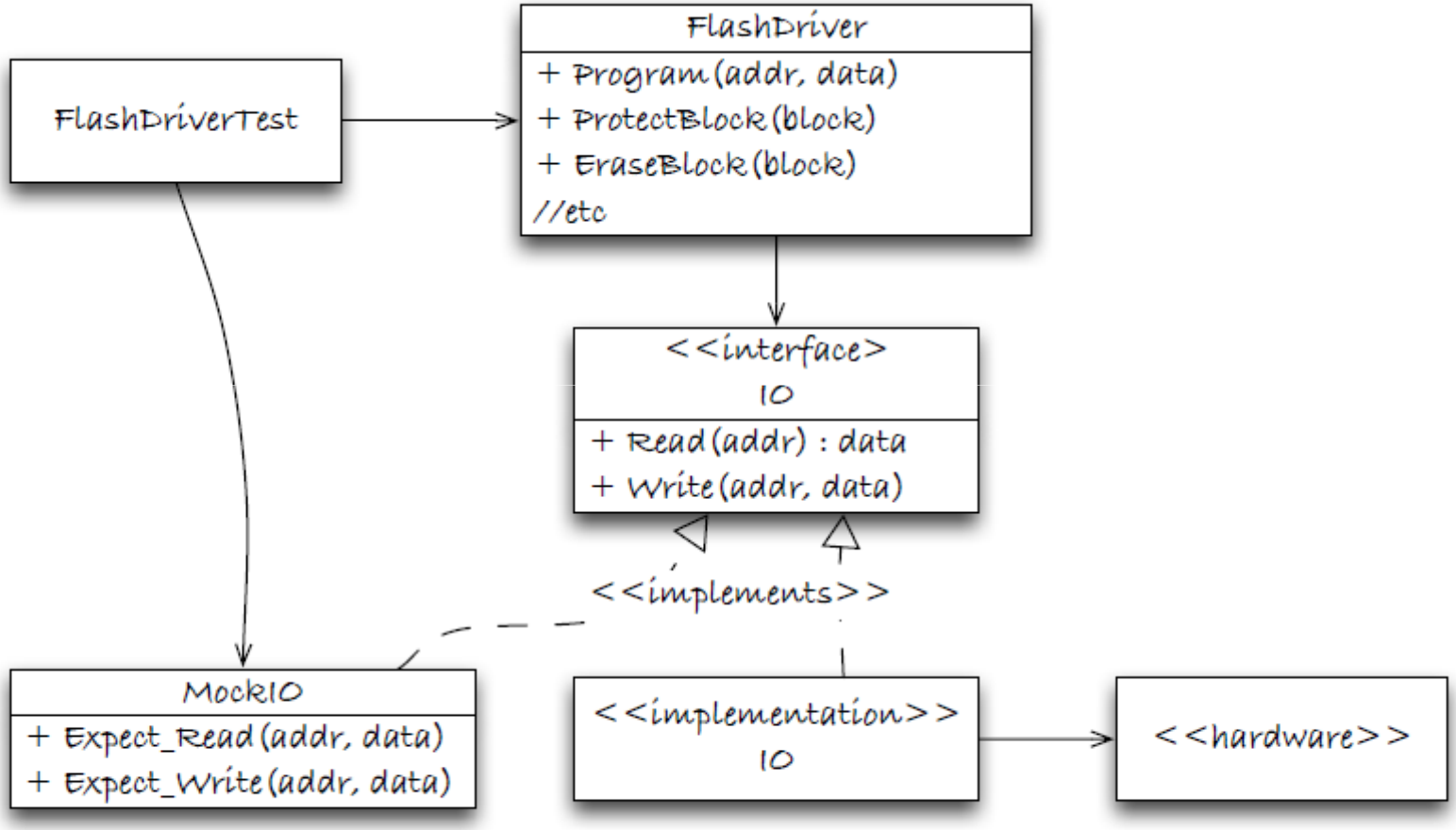

Replace a component that the system under test (SUT) depends on with a much **lighter-weight implementation**.

# Fake


1. Arrange:
set context
2. Act: trigger
behavior
Test
Class
Under
Test
Other
Collaborators
3. Interact with collaborators
4. Assert: check
for expected results
Listening
Fake


1. Arrange:
set context
2. Act: trigger
behavior
Test
Class
Under
Test
Other
Collaborators
3. Interact with collaborators
4. Assert: check
for expected results
Talking Fake

# Flash driver and its test fixture



# Substitutions in C

- 8.3 Link-Time Substitution with Object file
- 9.2 Function Pointer Substitution
- Preprocessor Substitution
- Link-Time & Function Pointer Substitution

# Object Oriented C

# Encapsulation

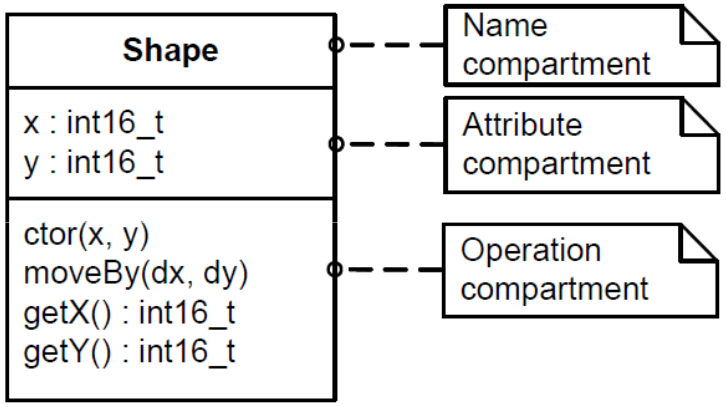
NínoX
Secure Networking Automation

# Encapsulation

## Declaration of the Shape “class” in C (shape.h header file)

```
#ifndef SHAPE_H
#define SHAPE_H
/* Shape's attributes... */
typedef struct {
          int16_t x; /* x-coordinate of Shape's position */
          int16_t y; /* y-coordinate of Shape's position */
} Shape;
/* Shape's operations (Shape's interface)... */
void Shape_ctor(Shape * const me, int16_t x, int16_t y);
void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy);
int16_t Shape_getX(Shape * const me);
int16_t Shape_getY(Shape * const me);
#endif /* SHAPE_H */
```

**UML Class Diagram of the Shape class**



# Encapsulation

## Definition of the Shape “class” in C (file shape.c)

```
#include "shape.h" /* Shape class interface */
/* constructor implementation */
void Shape_ctor(Shape * const me, int16_t x, int16_t y) {
          me->x = x;
          me->y = y;
}
/* move-by operation implementation */
void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy) {
          me->x += dx;
          me->y += dy;
}
/* "getter" operations implementation */
int16_t Shape_getX(Shape * const me) {
          return me->x;
}
int16_t Shape_getY(Shape * const me) {
          return me->y;
}
```

**NOTE: The “me” pointer in C corresponds directly to the implicit “this” pointer in C++.**

# Encapsulation

## Examples of using the Shape class in C (file main.c)

```
#include "shape.h" /* Shape class interface */
#include <stdio.h> /* for printf() */
int main() {
        Shape s1, s2; /* multiple instances of Shape */
        Shape_ctor(&s1, 0, 1);
        Shape_ctor(&s2, -1, 2);

        printf("Shape s1(x=%d,y=%d)₩n", Shape_getX(&s1), Shape_getY(&s1));
        printf("Shape s2(x=%d,y=%d)₩n", Shape_getX(&s2), Shape_getY(&s2));

        Shape_moveBy(&s1, 2, -4);
        Shape_moveBy(&s2, 1, -2);

        printf("Shape s1(x=%d,y=%d)₩n", Shape_getX(&s1), Shape_getY(&s1));
        printf("Shape s2(x=%d,y=%d)₩n", Shape_getX(&s2), Shape_getY(&s2));

        return 0;
}
```

# Inheritance

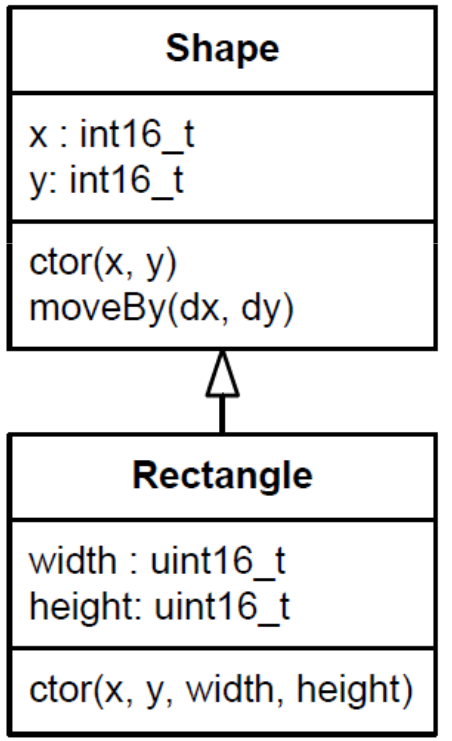
NínoX
Secure Networking Automation

# Inheritance

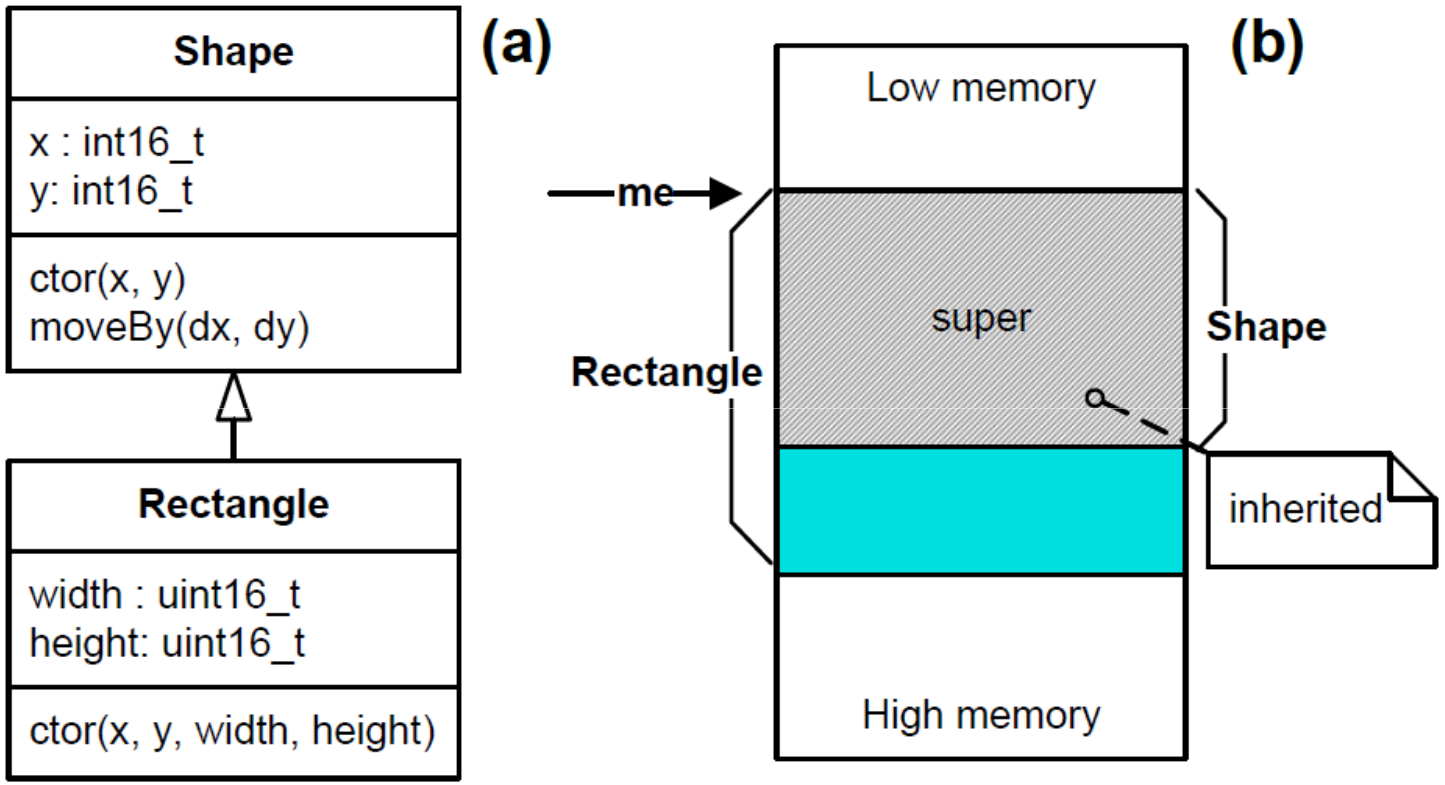
## Declaration of the Rectangle as a Subclass of Shape (file rect.h)

```
#ifndef RECT_H
#define RECT_H
#include "shape.h" /* the base class interface */
/* Rectangle's attributes... */
typedef struct {
          Shape super; /* <== inherits Shape */
          /* attributes added by this subclass... */
          uint16_t width;
          uint16_t height;
} Rectangle;
/* constructor prototype */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t
y,
uint16_t width, uint16_t height);
#endif /* RECT_H */
```

**class diagram with inheritance**



# Inheritance




Single inheritance in C:
(a) class diagram with inheritance, and
(b) memory layout for Rectangle and Shape objects

# Inheritance

**The Constructor of class Rectangle (file rect.c)**

```
#include "rect.h"
/* constructor implementation */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,uint16_t width,
uint16_t height)
{
        /* first call superclass' ctor */
        Shape_ctor(&me->super, x, y);
        /* next, you initialize the attributes added by this subclass... */
        me->width = width;
        me->height = height;
}
```

# Inheritance

## Example of Using Rectangle Objects (file main.c)

```
#include "rect.h" /* Rectangle class interface */
#include <stdio.h> /* for printf() */
int main() {
        Rectangle r1, r2; /* multiple instances of Rect */
        /* instantiate rectangles... */
        Rectangle_ctor(&r1, 0, 2, 10, 15);
        Rectangle_ctor(&r2, -1, 3, 5, 8);
        printf("Rect r1(x=%d,y=%d,width=%d,height=%d)₩n",
                r1.super.x, r1.super.y, r1.width, r1.height);
        /* re-use inherited function from the superclass Shape... */
        Shape_moveBy((Shape *)&r1, -2, 3);
        Shape_moveBy(&r2.super, 2, -1);
        printf("Rect r1(x=%d,y=%d,width=%d,height=%d)₩n",
                r1.super.x, r1.super.y, r1.width, r1.height);
        return 0;
}
```

# Polymorphism

(Virtual Functions)

NínoX
Secure Networking Automation

# Polymorphism

## Declaration of the Shape base class (file shape.h)

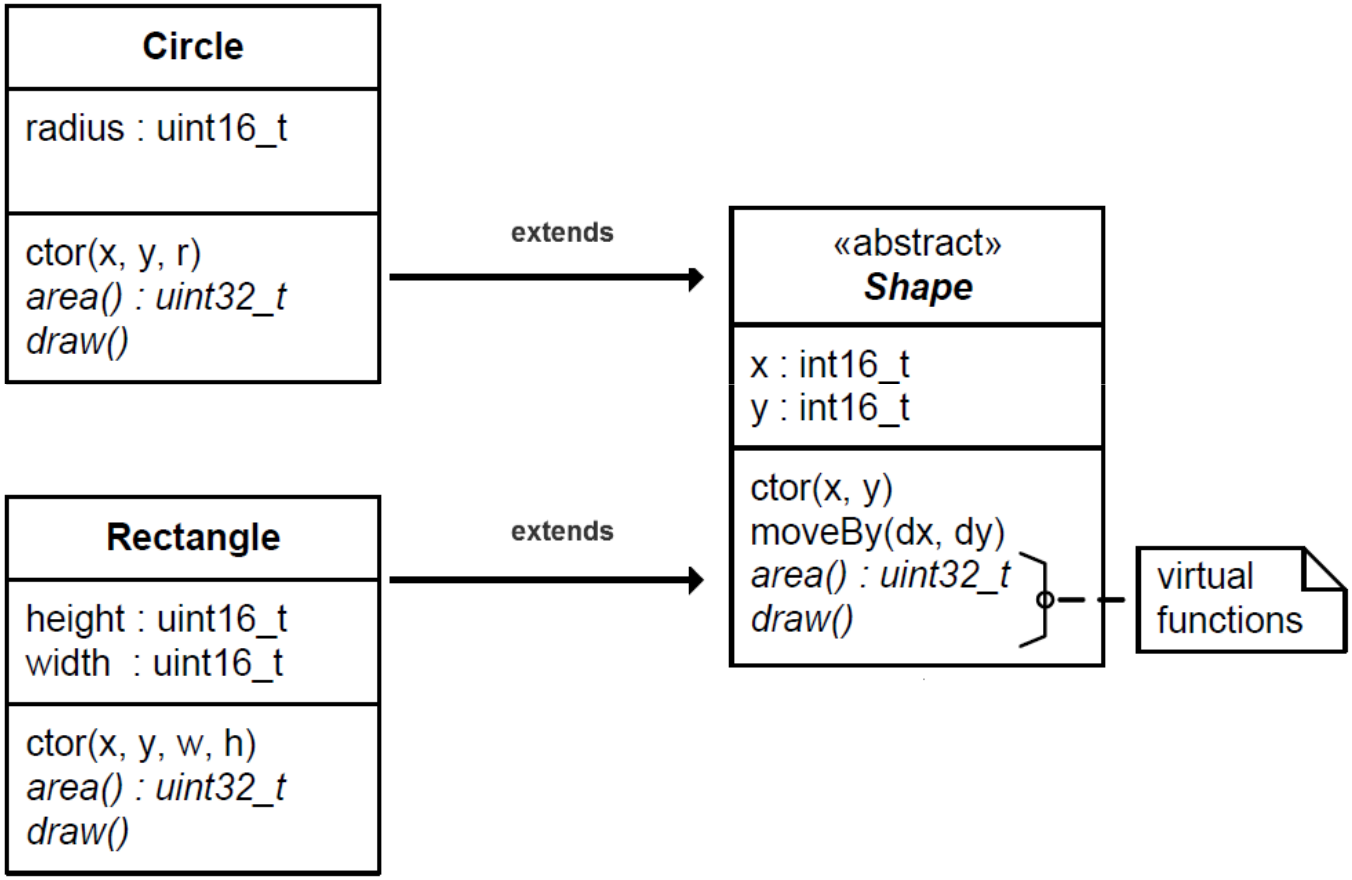
```
        #ifndef SHAPE_H
        #define SHAPE_H
        #include <stdint.h>
        /* Shape's attributes... */
        struct ShapeVtbl; /* forward declaration */
        typedef struct {
(1)     struct ShapeVtbl const *vptr; /* <== Shape's Virtual Pointer */
                  int16_t x; /* x-coordinate of Shape's position */
                  int16_t y; /* y-coordinate of Shape's position */
        } Shape;

        /* Shape's virtual table */
(2)    struct ShapeVtbl {
(3)               uint32_t (*area)(Shape const * const me);
(4)               void (*draw)(Shape const * const me);
        };
```

# Polymorphism

## Declaration of the Shape base class (file shape.h)

```
        /* Shape's operations (Shape's interface)... */
        void Shape_ctor(Shape * const me, int16_t x, int16_t y);
        void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy);

        static inline uint32_t Shape_area(Shape const * const me) {
(5)             return (*me->vptr->area)(me);
        }

        static inline void Shape_draw(Shape const * const me) {
(6)             (*me->vptr->draw)(me);
        }

        /* generic operations on collections of Shapes */
(7)     Shape const *largestShape(Shape const *shapes[], uint32_t nShapes);
(8)     void drawAllShapes(Shape const *shapes[], uint32_t nShapes);
        #endif /* SHAPE_H */
```

# Polymorphism



# Design Principles

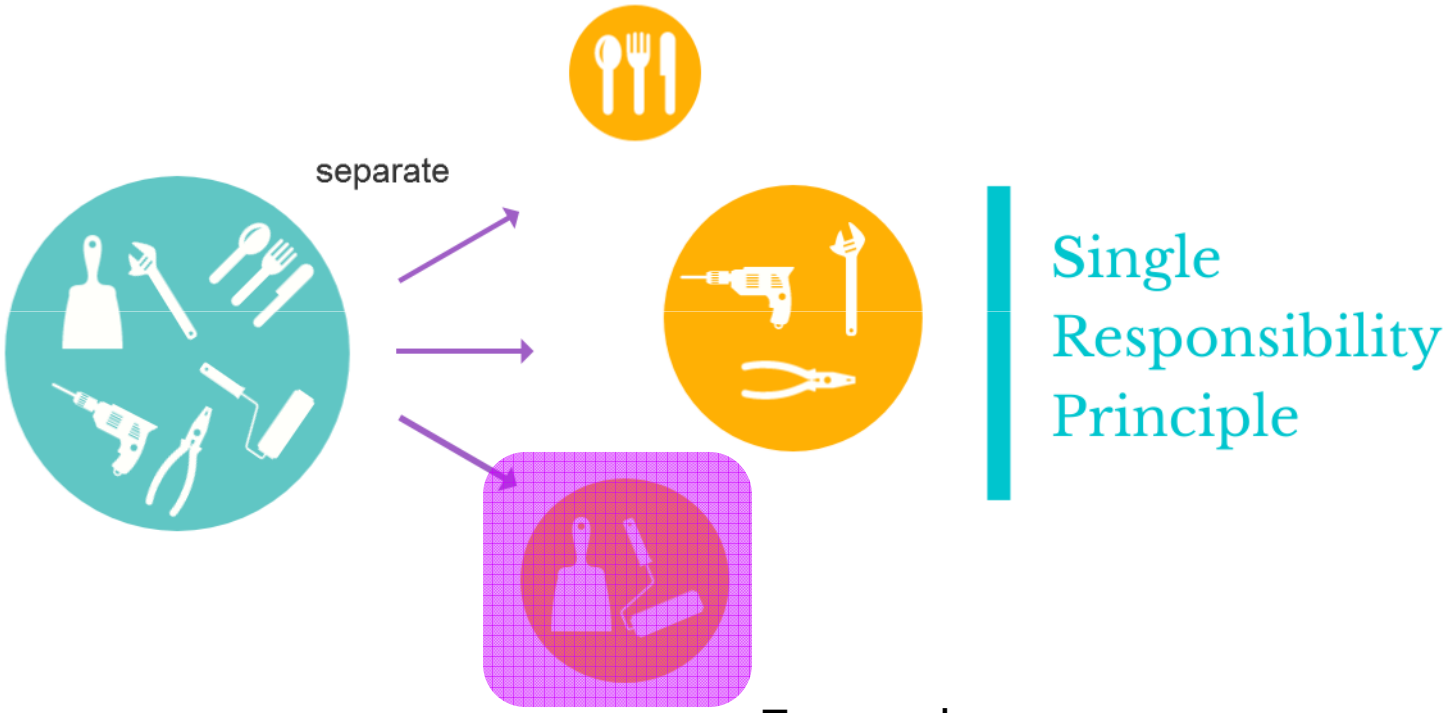
# TOPICS

- SOLID
- DRY
- KISS
- YAGNI

# SOLID

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

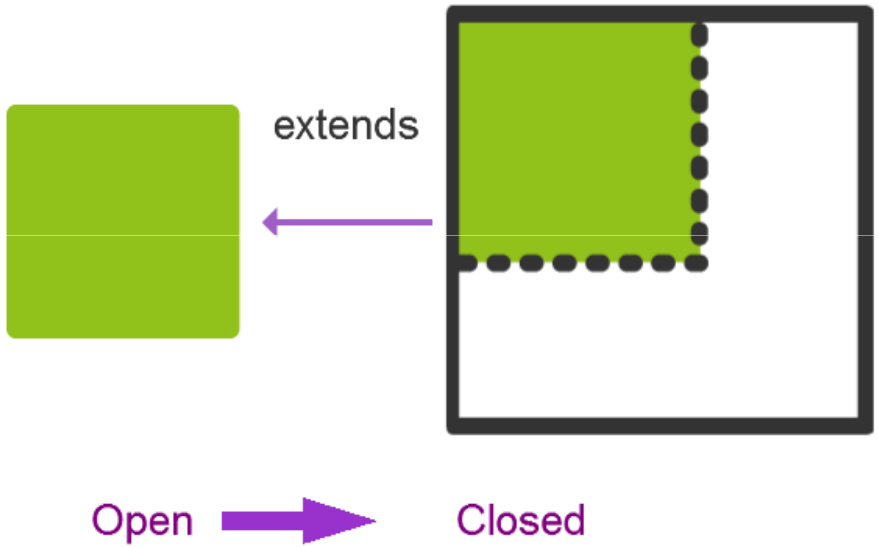
# Single Responsibility Principle




Focused on
Responsibility of class

- Each class in your system should have only one responsibility
- There should never be more than one reason for a class to change.

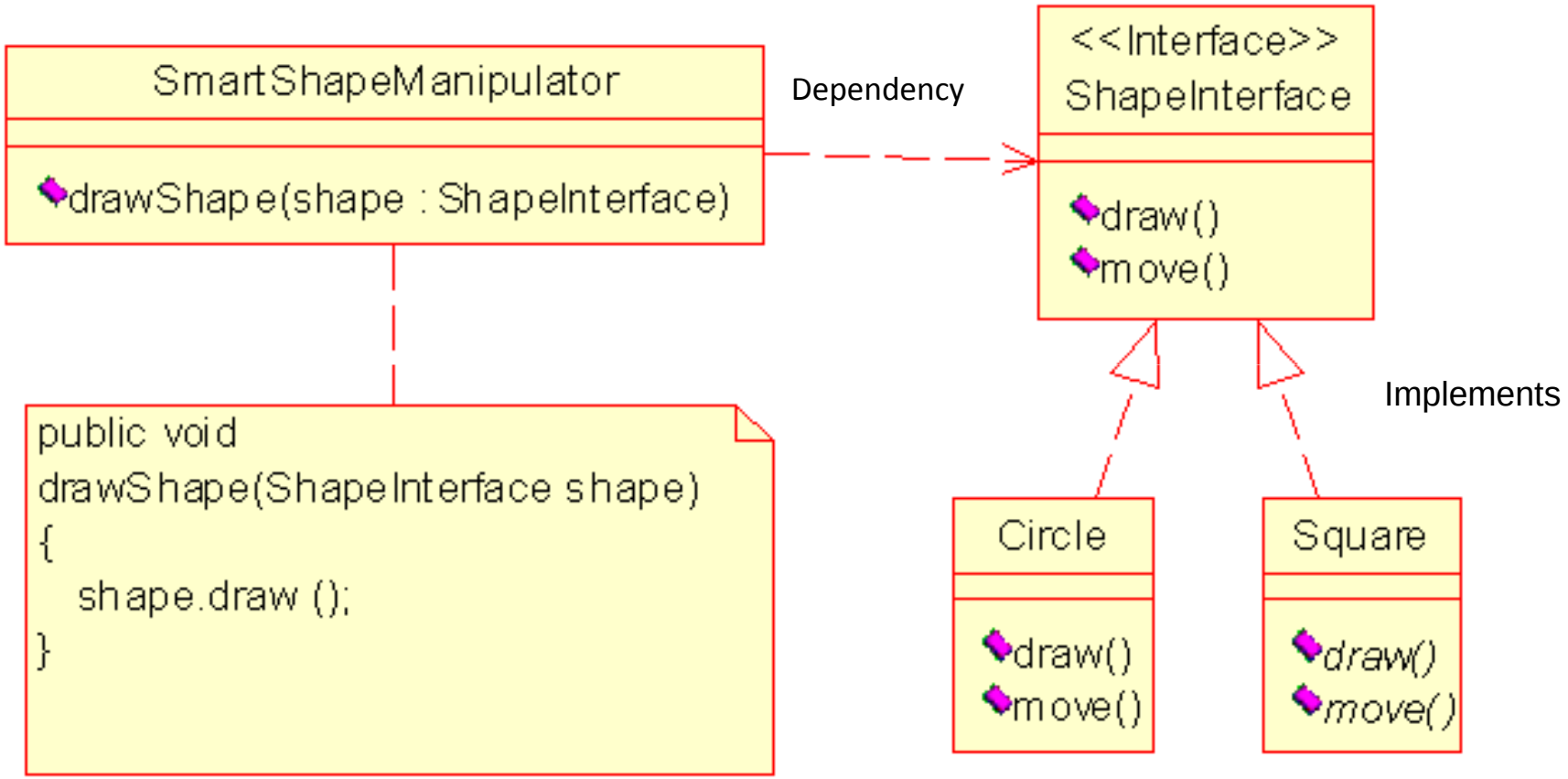
# Open Closed Principle




Open/Close Principle

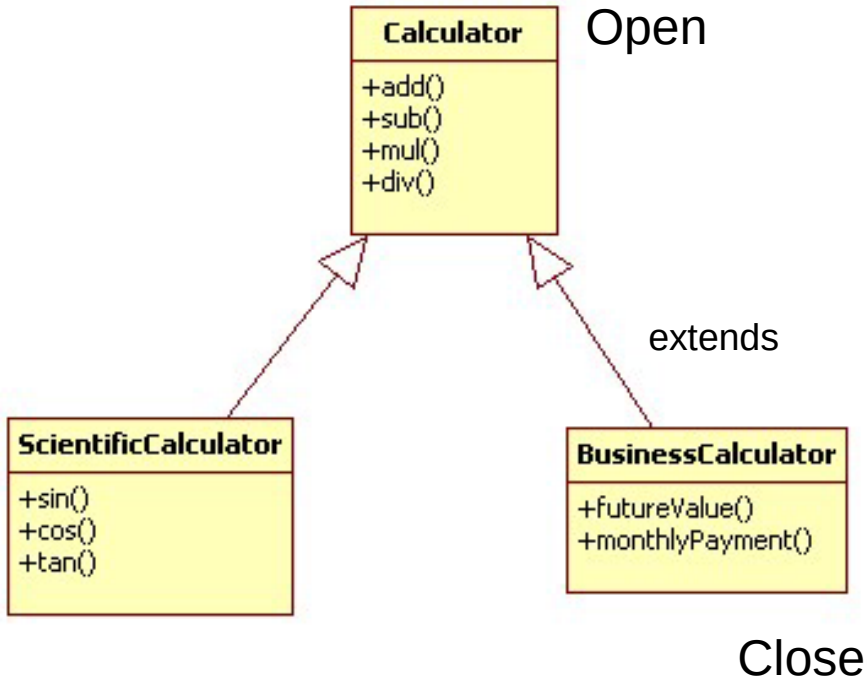
Software entities like classes, modules and functions should be open for extension but closed for modifications.

# Open Closed Principle

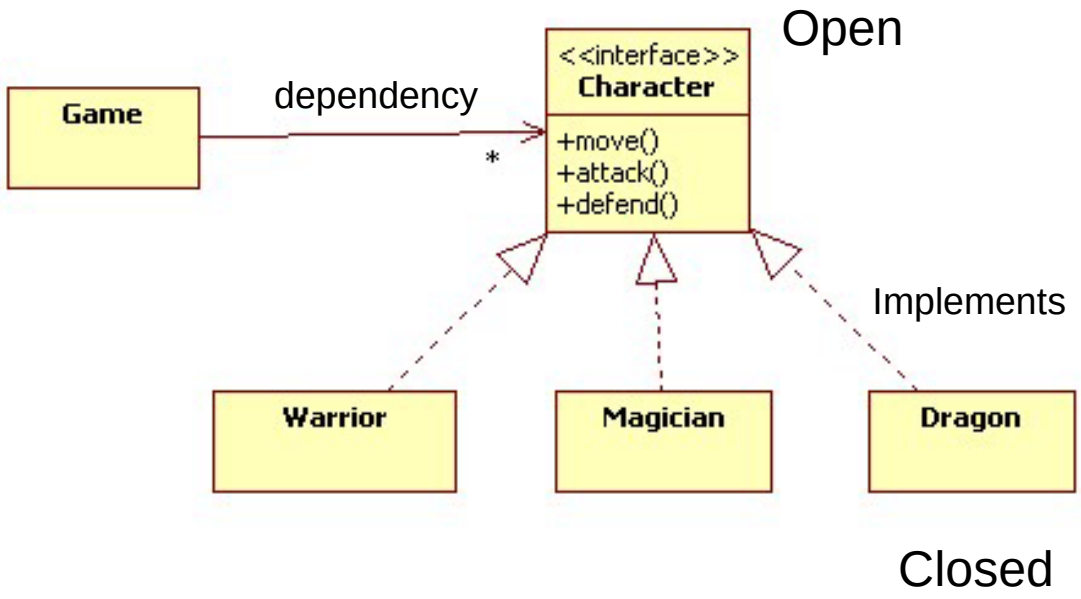


# Open Closed Principle

## Inheritance

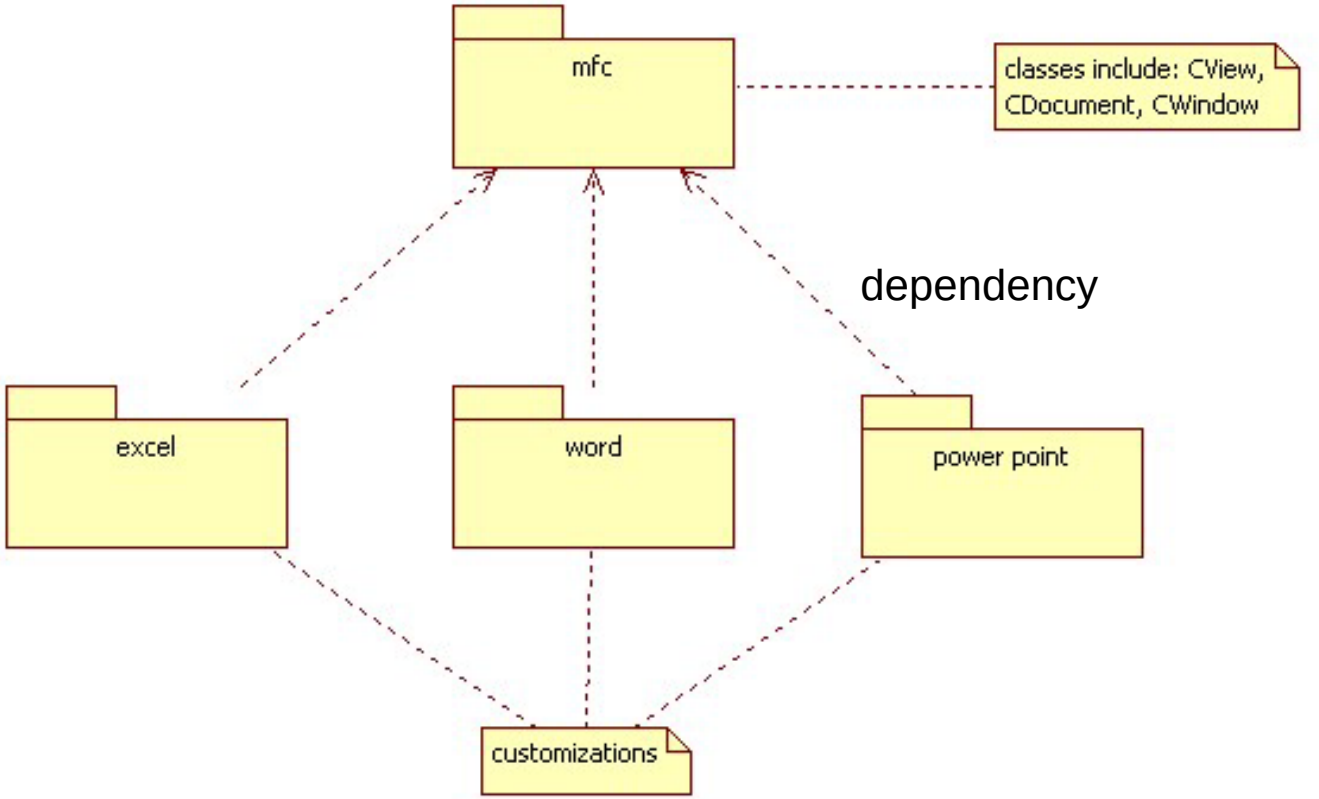



## Polymorphism

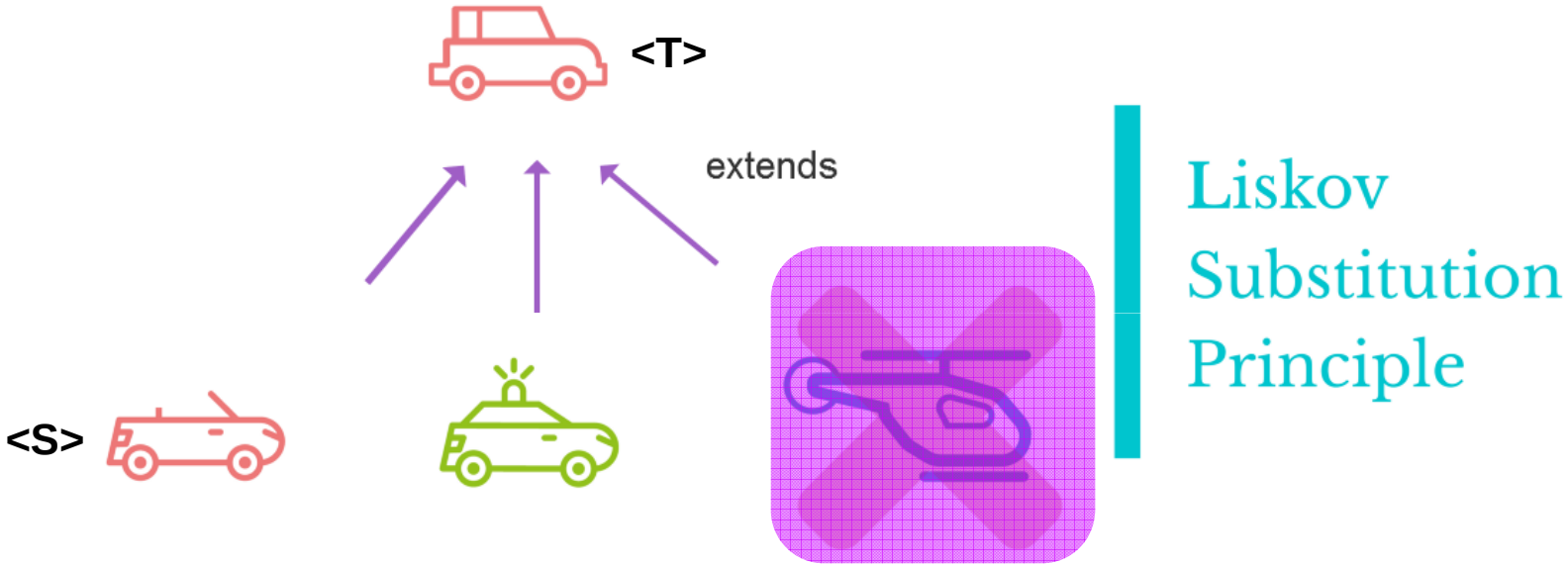



open for extension, but closed for modification

# Horizontal Framework (Microsoft Foundation Classes)



# Liskov Substitution Principle

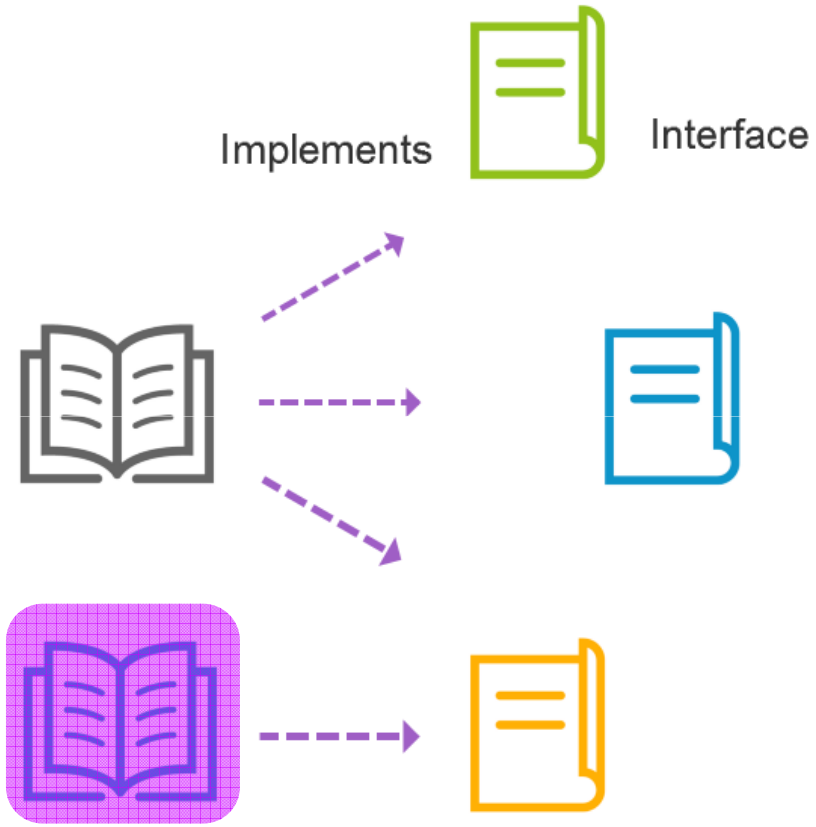



Focused on Implemented Class

If **S** is a subtype of **T**, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program (correctness, task performed, etc.)

*Subtypes must be substitutable for their base types*

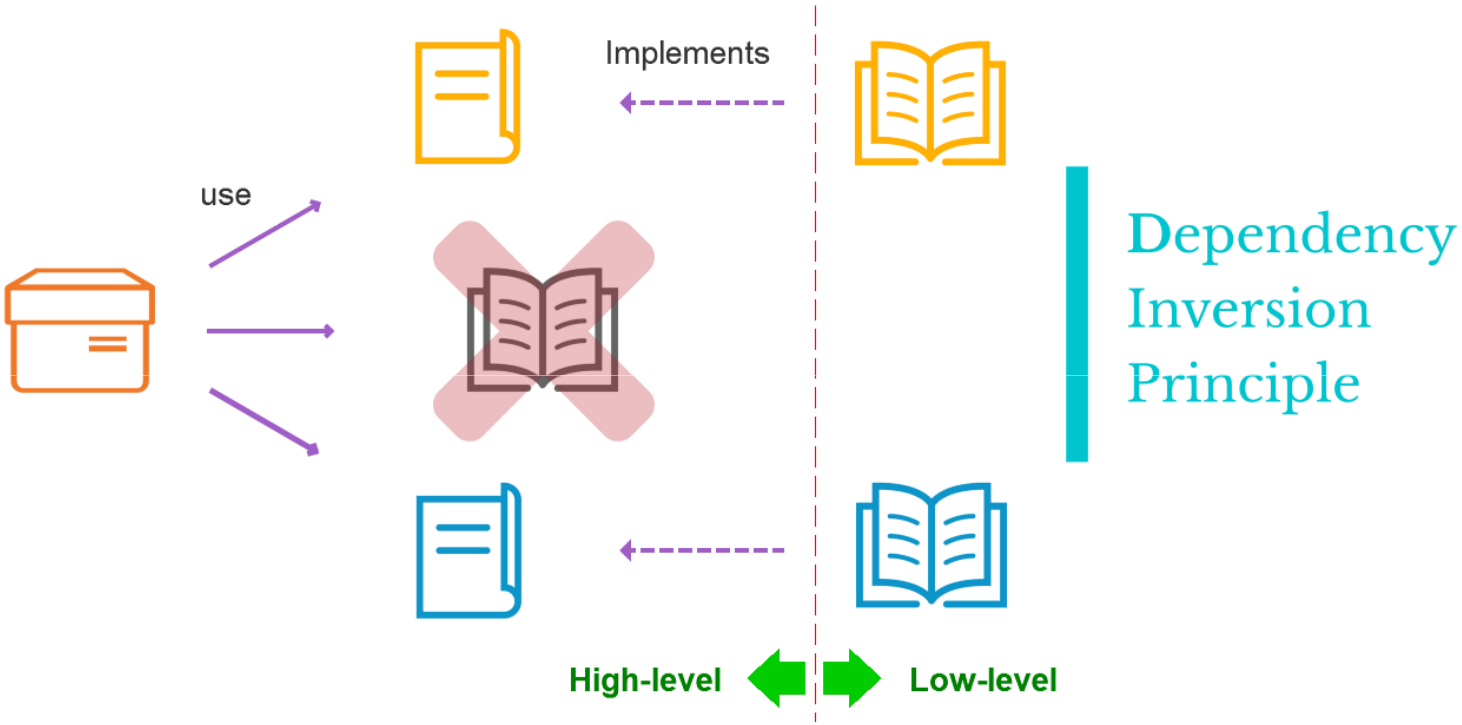
# Interface Segregation Principle




Interface Segregation Principle

The interface-segregation principle states that no client should be forced to depend on methods it does not use.

# Dependency Inversion Principle




- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

program to an interface,  not an implementation

# Dependency Injection(DI)

**Non DI**

```
class Programmer {
  Computer computer;
  Programmer() {
    computer = new Computer();//non
di
  }

  void code(){
    computer.program();
  }
}
```

**DI using Constructor**

```
class Programmer {
  Computer computer;
  Programmer(Computer c) {
    computer = c;//is the DI


  }

  void code(){
    computer.program();
  }
}
```

# Inversion of Control(IoC)

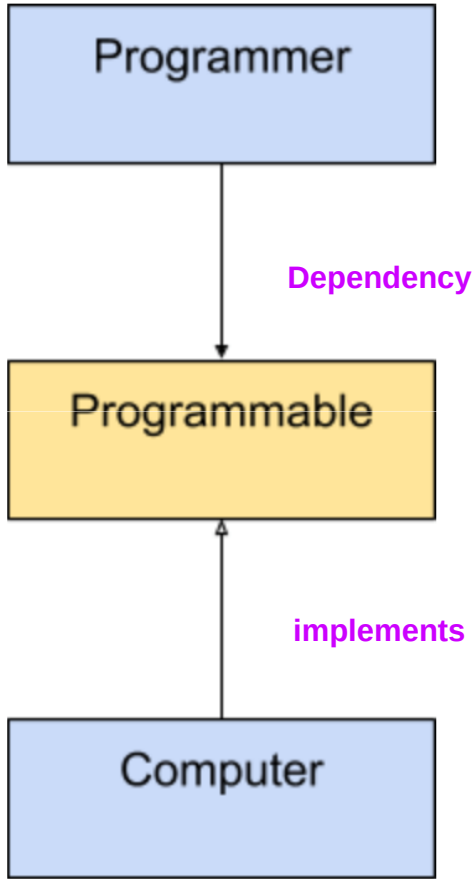
```
class Work { // like a IoC container
  public static void main(String[] args) {
    Computer computer = new Computer();
    Programmer programmer = new
Programmer(computer);
  }
}
```

Decoupling programer and computer using IoC container like work.

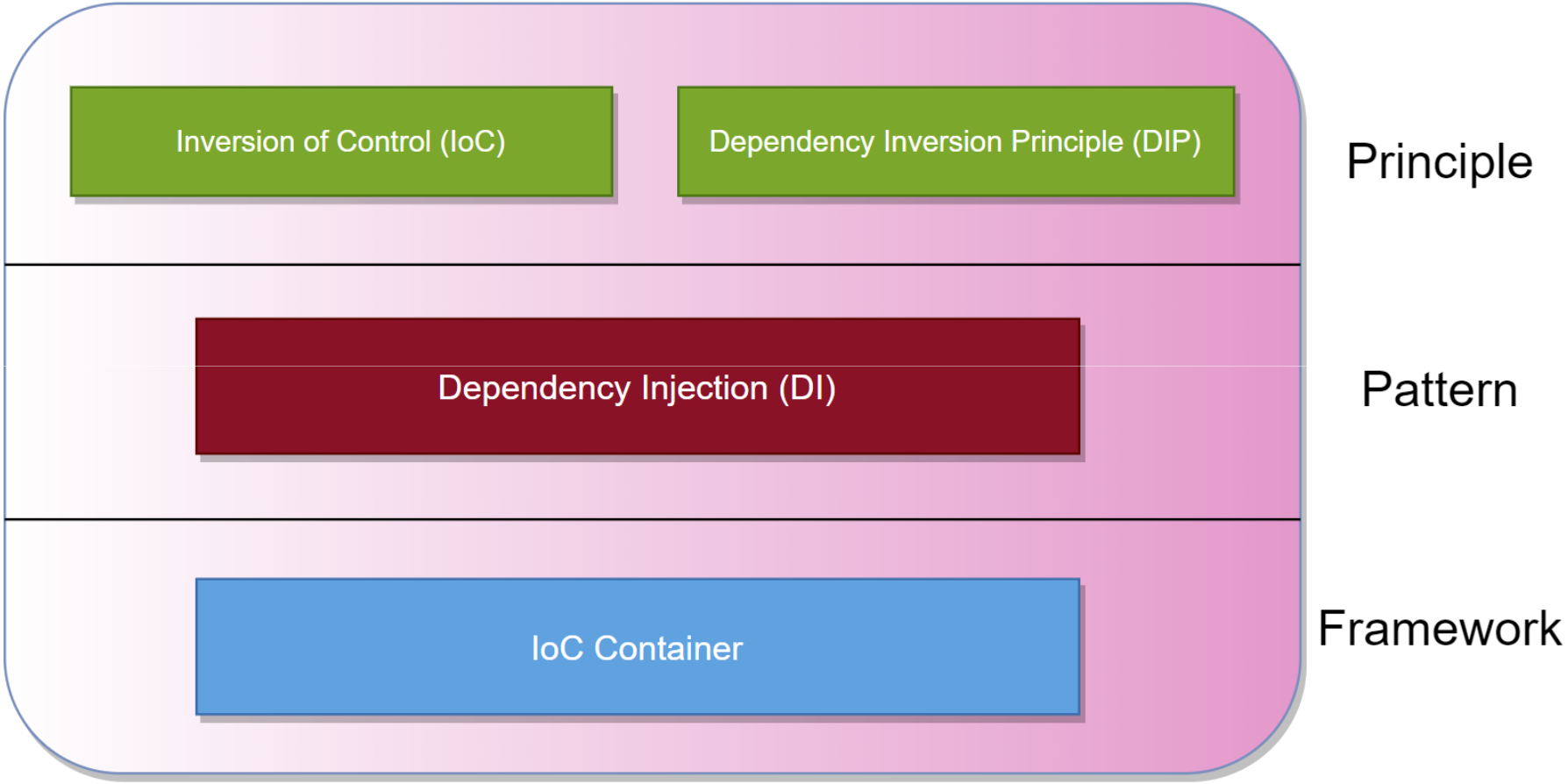
# Dependency Inversion Principle

```
interface Programmable {
   void program();
}

class Computer implements
Programmable {
   @Override
   void program(){
    //computer program
   }
}
```

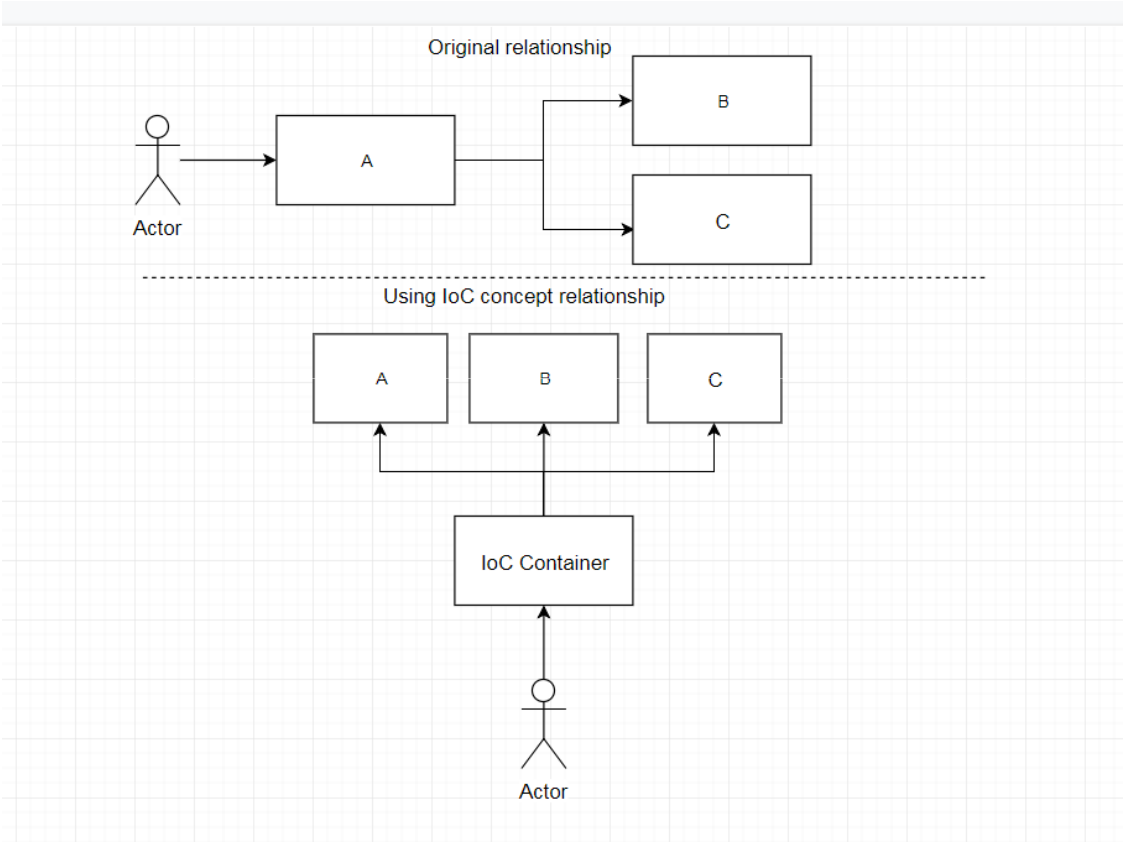
```
class Programmer {
 Programmable programmable;
 Programmer(Programmable p) {
  programmable = p;
 }

 void code(){
  programmable.program();
 }
}
```



# DIP , DI , IoC


Inversion of Control (IoC)
Dependency Inversion Principle (DIP)
Principle
Dependency Injection (DI)
Pattern
IoC Container
Framework

# Applications of DIP




**Design Pattern**

- Strategy
- Dependency Injection
- Template Method
- Abstract Factory
- Factory Method
- Service Locator
- ...

**Design Principle**

- Inversion of Control (IoC)

- **DRY - Don’t Repeat Yourself**
- **KISS - Keep it simple, stupid!**
- **YAGNI - You aren't gonna need it (Need)**