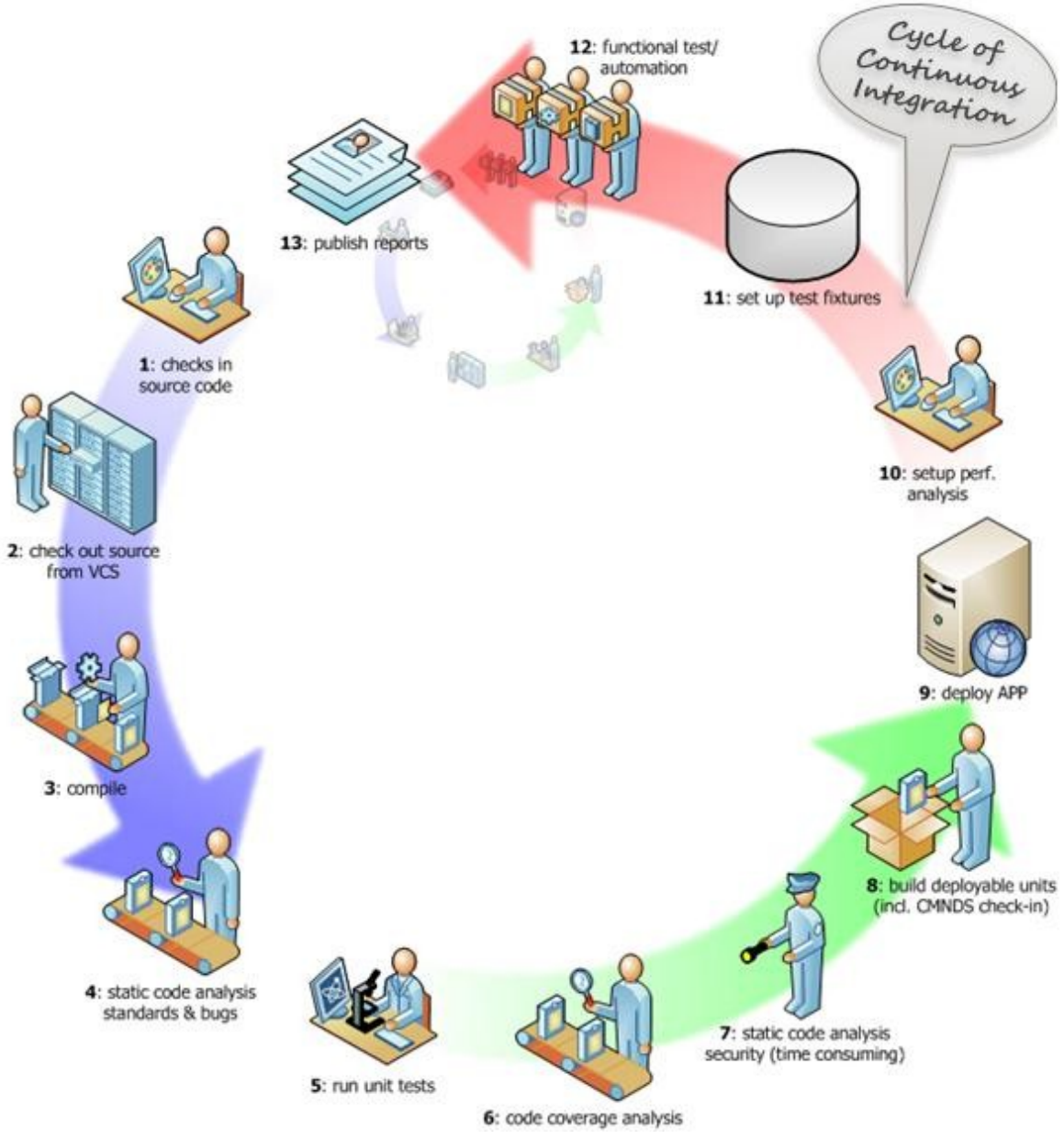
Continuous Integration &
Unit Test

Why continuous delivery

- build new products and services faster than other companies
- usage of Agile and DevOps is increasing
- The potential for more bugs

Original and © by Scott Adams, 1995

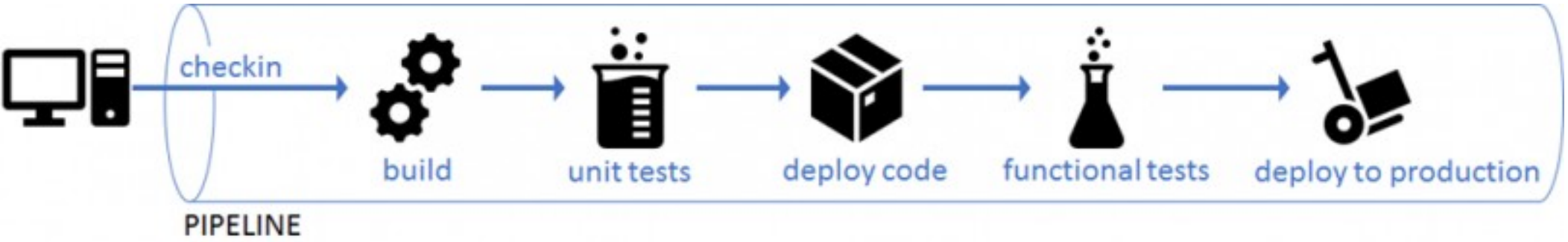
Automation Testing is important for CI/CD

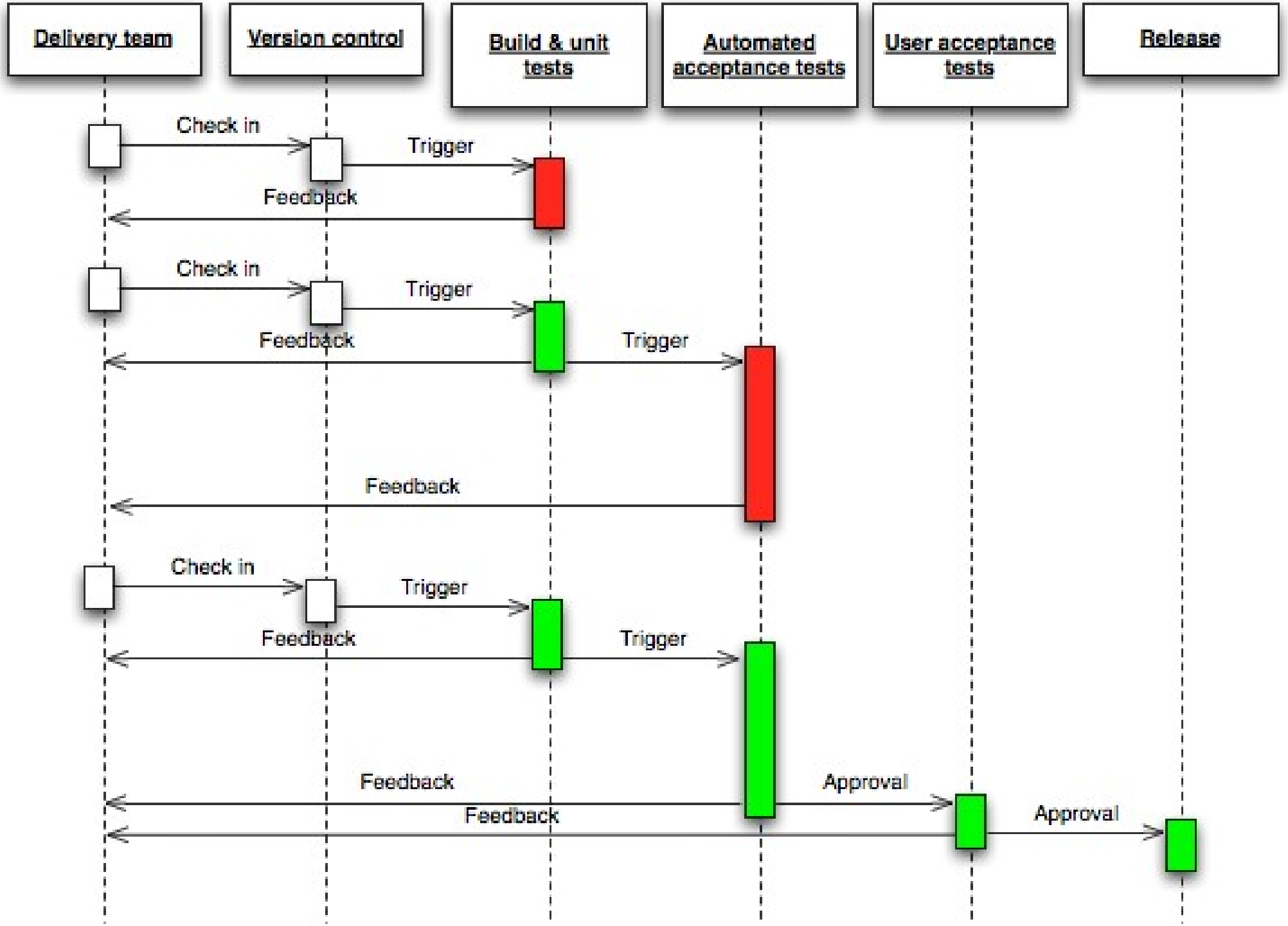


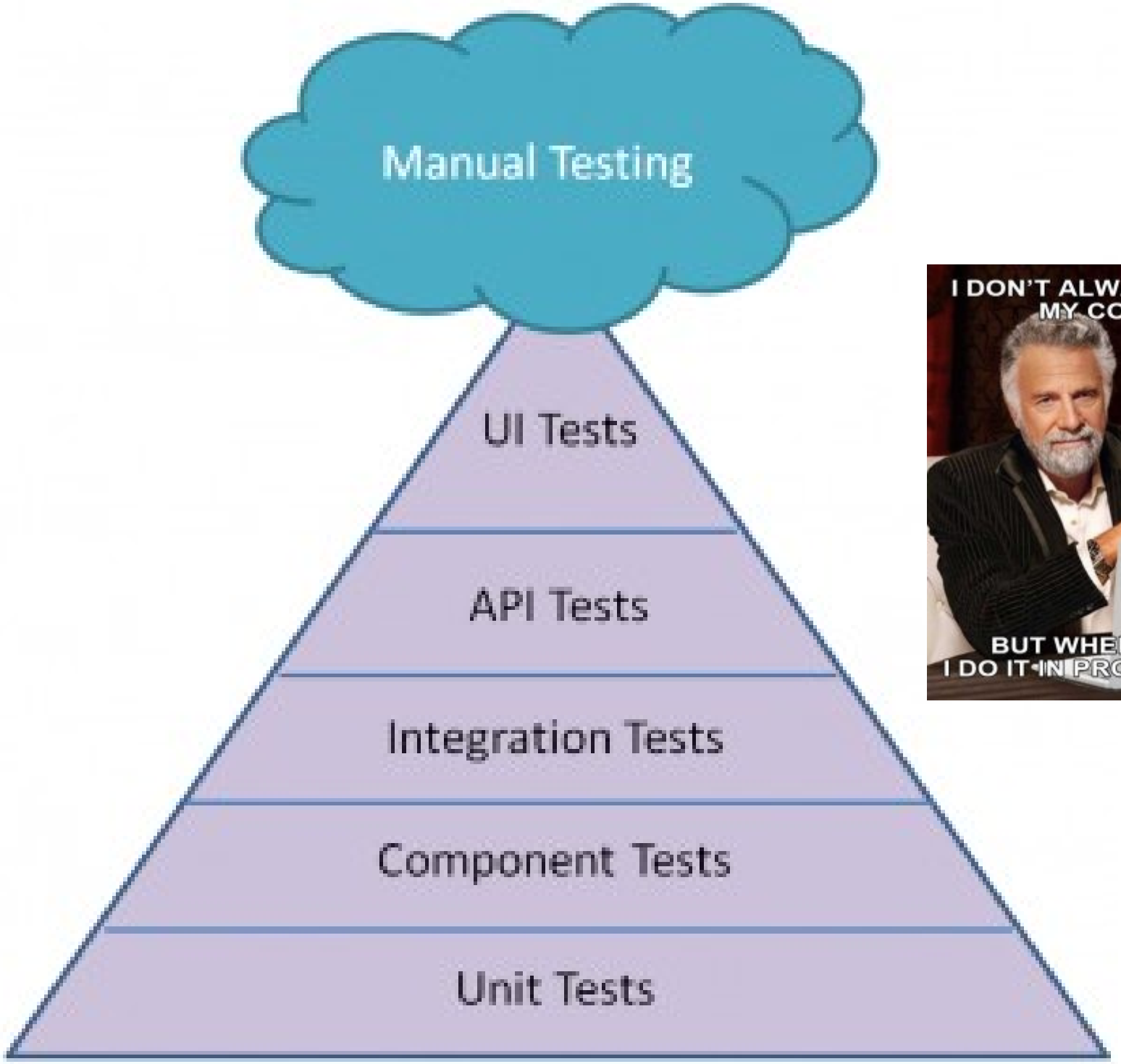
Automation testing is important for a successful Continuous Integration/Continuous Delivery

- Easy and efficient assessment of minor changes
- Faster regression tests
- More consistent results
- Better agility

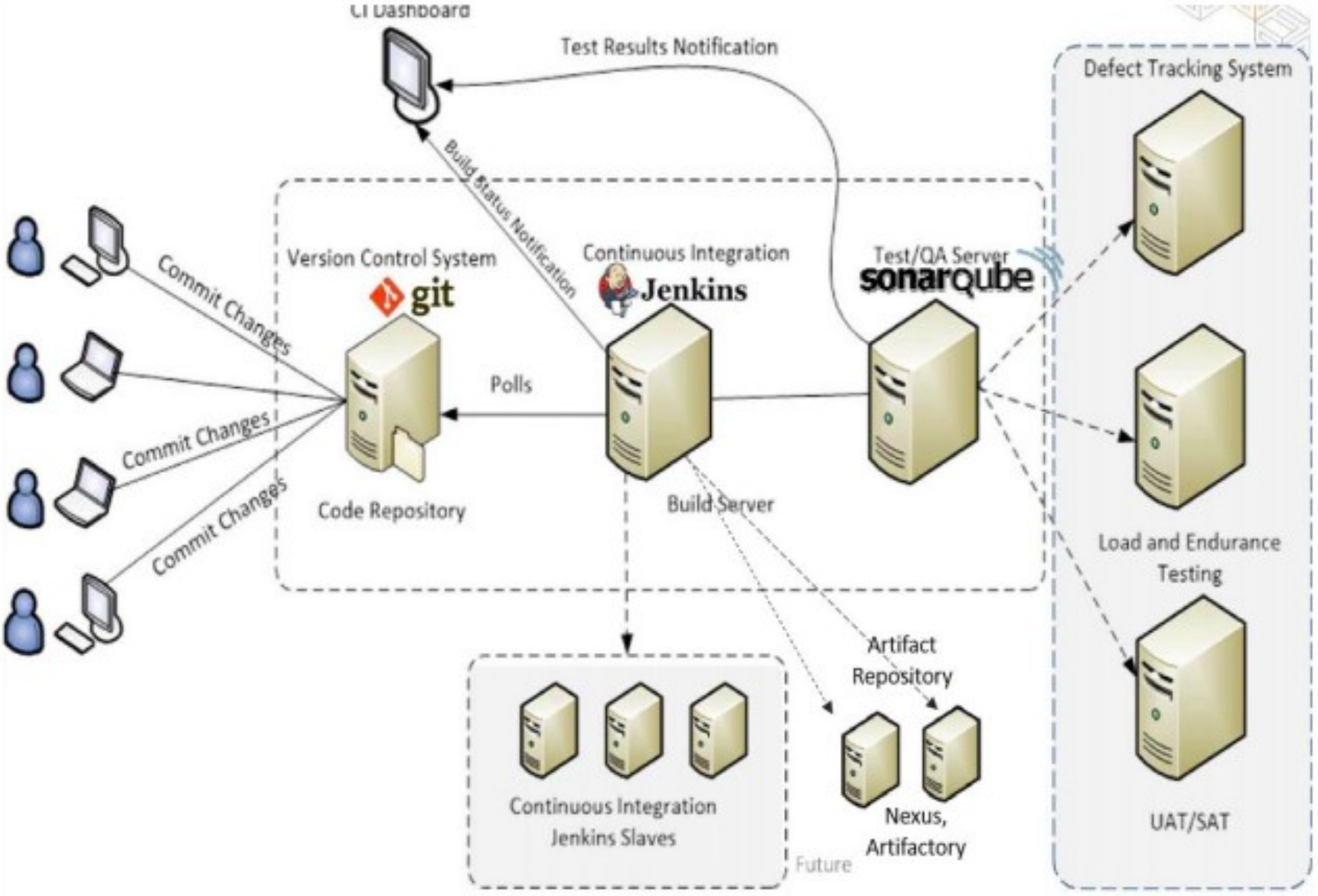
Deployment pipeline




Delivery team
Version control
Build & unit tests
Automated acceptance tests
User acceptance tests
Release
Check in
Trigger
Feedback
Check in
Trigger
Feedback
Trigger
Feedback
Check in
Trigger
Feedback
Trigger
Feedback
Approval
Feedback
Approval


Manual Testing
UI Tests
API Tests
Integration Tests
Component Tests
Unit Tests

I DON'T ALWAYS TEST MY CODE
BUT WHEN I DO, I DO IT IN PRODUCTION


CI Dashboard
Test Results Notification
Build Status Notification
Defect Tracking System
Version Control System
git
Continuous Integration
Jenkins
Test/QA Server
sonarqube
Commit Changes
Commit Changes
Commit Changes
Polls
Code Repository
Build Server
Load and Endurance Testing
Artifact Repository
Continuous Integration Jenkins Slaves
Nexus, Artifactory
UAT/SAT
Future

Unit test

- is a level of the software testing process where individual units/components of a software/system are tested.
- The purpose is to validate that each unit of the software performs as designed

Unit test

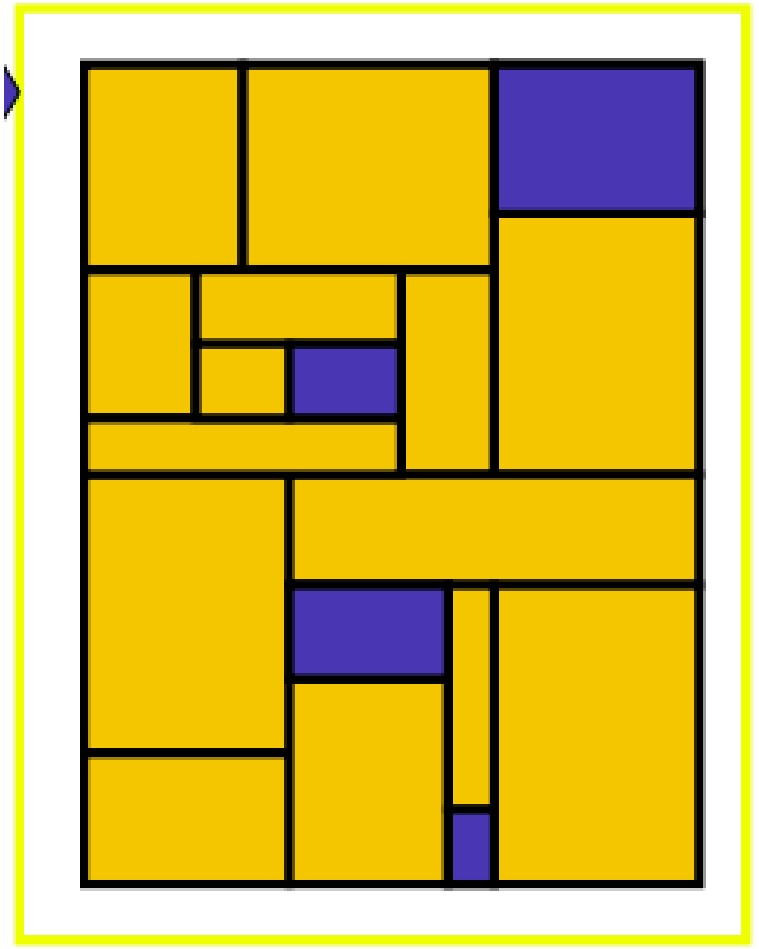
- concerned with functional correctness and completeness of individual program units.
- typically written and run by software developers to ensure that code meets its design and behaves as intended.
- Its goal is to isolate each part of the program and show that the individual parts are correct.

What is Unit Testing

- Concerned with Functional correctness and completeness
- Error handling
- Checking input values (parameter)
- Correctness of output data (return values)
- Optimizing algorithm and performance

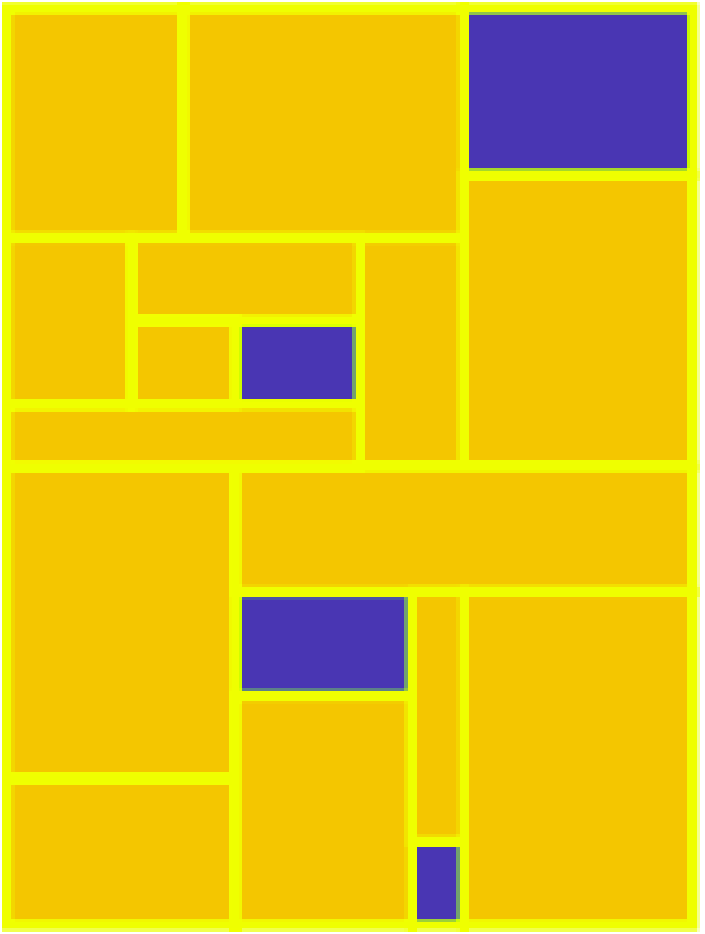
Traditional Testing

- Test the system as a whole
- Individual components rarely tested
- Errors go undetected
- Isolation of errors difficult to track down



Unit Testing

- Each part tested individually
- All components tested at least once
- Errors picked up earlier
- Scope is smaller, easier to fix errors



Traditional Testing vs Unit Testing

- Print Statements
- Use of Debugger
- Debugger Expressions
- Test Scripts

- Isolatable
- Repeatable
- Automatable
- Easy to Write

Types of testing

- Black box testing – (application interface, internal module interface and input/output description)
- White box testing- function executed and checked
- Gray box testing - test cases, risks assessments and test methods

Black-, Gray-, & White-box Testing

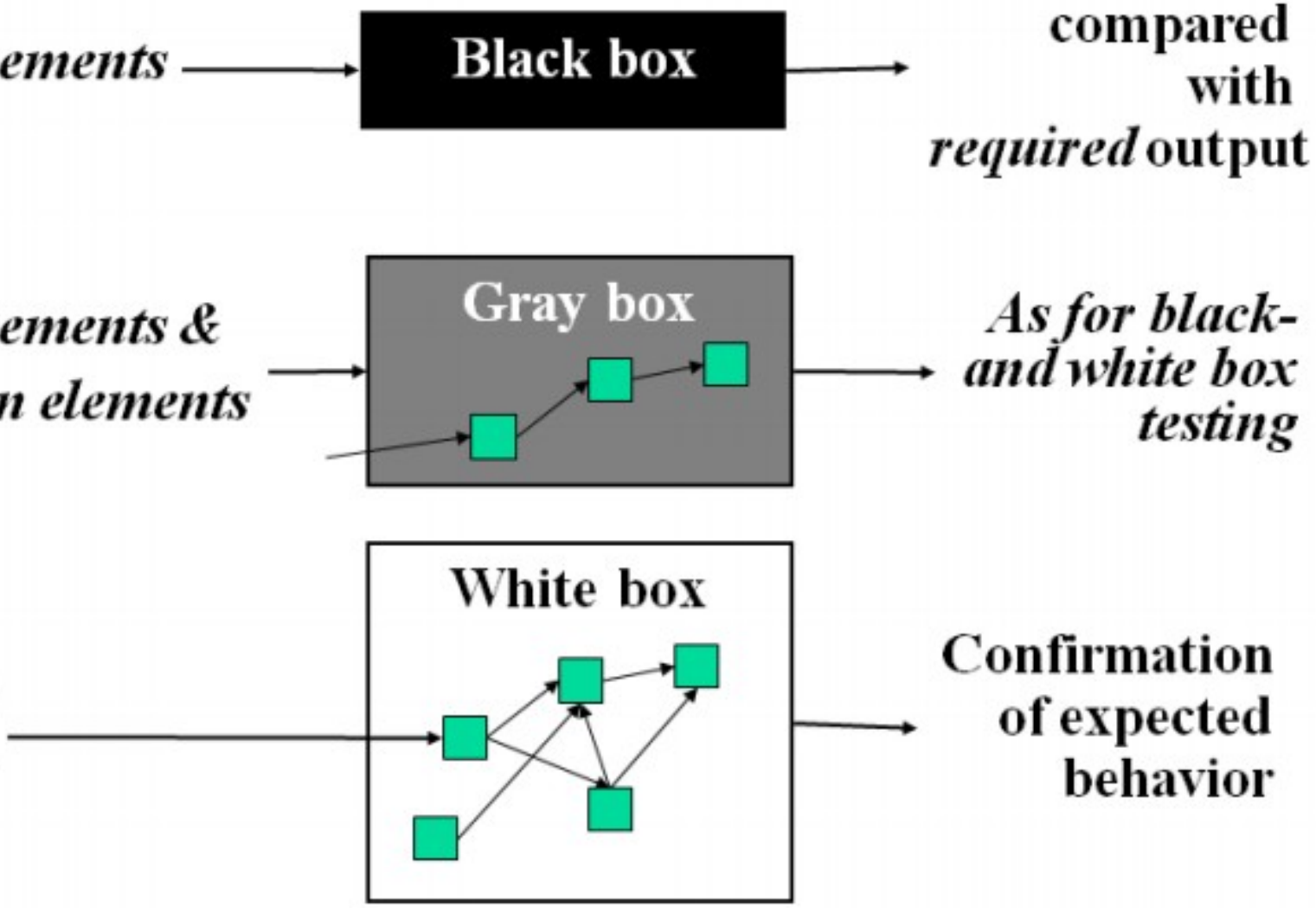
Input determined by...

Result

... requirements → **Black box** → *Actual* output compared with *required* output

... requirements & key design elements → **Gray box** → *As for black- and white box testing*

... design elements → **White box** → Confirmation of expected behavior



白盒测试与黑盒测试

白盒测试概述

- 白盒测试也称结构测试或逻辑驱动测试，是针对被测单元内部是如何进行工作的测试。它根据程序的控制结构设计测试用例，主要用于软件或程序验证。
- 白盒测试法检查程序内部逻辑结构，对所有逻辑路径进行测试，是一种穷举路径的测试方法。但即使每条路径都测试过了，仍然可能存在错误。因为:
  - 穷举路径测试无法检查出程序本身是否违反了设计规范，即程序是否是一个错误的程序。
  - 穷举路径测试不可能查出程序因为遗漏路径而出错。
  - 穷举路径测试发现不了一些与数据相关的错误。

白盒测试概述

- 采用白盒测试方法必须遵循以下几条原则，才能达到测试的目的:
  - 保证一个模块中的所有独立路径至少被测试一次。
  - 所有逻辑值均需测试真 (true) 和假 (false) 两种情况。
  - 检查程序的内部数据结构，保证其结构的有效性。
  - 在上下边界及可操作范围内运行所有循环。
  - 白盒测试主要是检查程序的内部结构、逻辑、循环和路径。

1 测试覆盖率

- 测试覆盖率：用于确定测试所执行到的覆盖项的百分比。其中的覆盖项是指作为测试基础的一个入口或属性，比如语句、分支、条件等。
- 测试覆盖率可以表示出测试的充分性，在测试分析报告中可以作为量化指标的依据，测试覆盖率越高效果越好。但覆盖率不是目标，只是一种手段。
- 测试覆盖率包括功能点覆盖率和结构覆盖率：
  - 功能点覆盖率大致用于表示软件已经实现的功能与软件需要实现的功能之间的比例关系。
  - 结构覆盖率包括语句覆盖率、分支覆盖率、循环覆盖率、路径覆盖率等等。

2 逻辑覆盖法

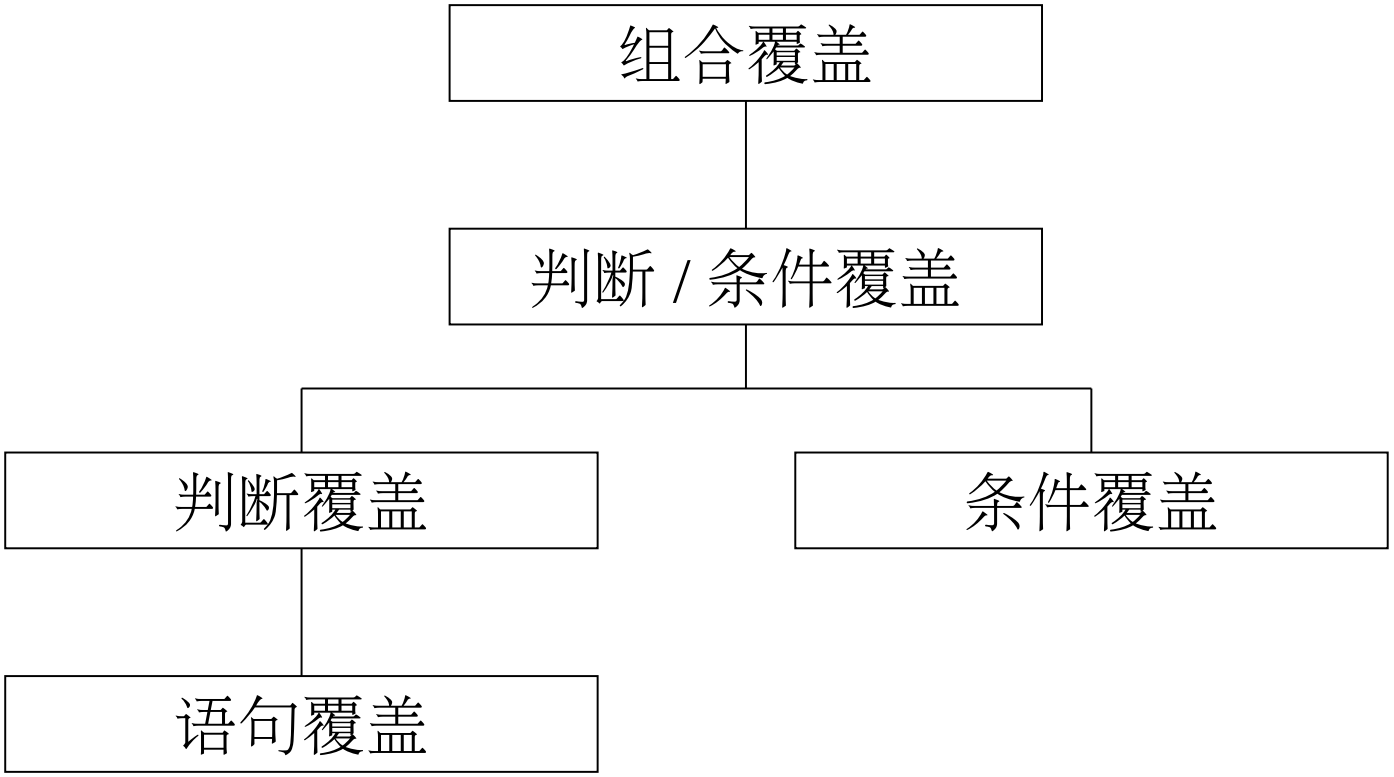
- 根据覆盖目标的不同，逻辑覆盖又可分为语句覆盖、判定覆盖、条件覆盖、判定 / 条件覆盖、组合覆盖和路径覆盖。
  - 语句覆盖：选择足够多的测试用例，使得程序中的每个可执行语句至少执行一次。
  - 判定覆盖：通过执行足够的测试用例，使得程序中的每个判定至少都获得一次“真”值和“假”值， 也就是使程序中的每个取“真”分支和取“假”分支至少均经历一次，也称为“分支覆盖”。
  - 条件覆盖：设计足够多的测试用例，使得程序中每个判定包含的每个条件的可能取值（真 / 假）都至少满足一次。

逻辑覆盖法

- 判定 / 条件覆盖：设计足够多的测试用例，使得程序中每个判定包含的每个条件的所有情况（真 / 假）至少出现一次，并且每个判定本身的判定结果（真 / 假）也至少出现一次。
  - 满足判定 / 条件覆盖的测试用例一定同时满足判定覆盖和条件覆盖。
- 组合覆盖：通过执行足够的测试用例，使得程序中每个判定的所有可能的条件取值组合都至少出现一次。
  - 满足组合覆盖的测试用例一定满足判定覆盖、条件覆盖和判定 / 条件覆盖。

逻辑覆盖法

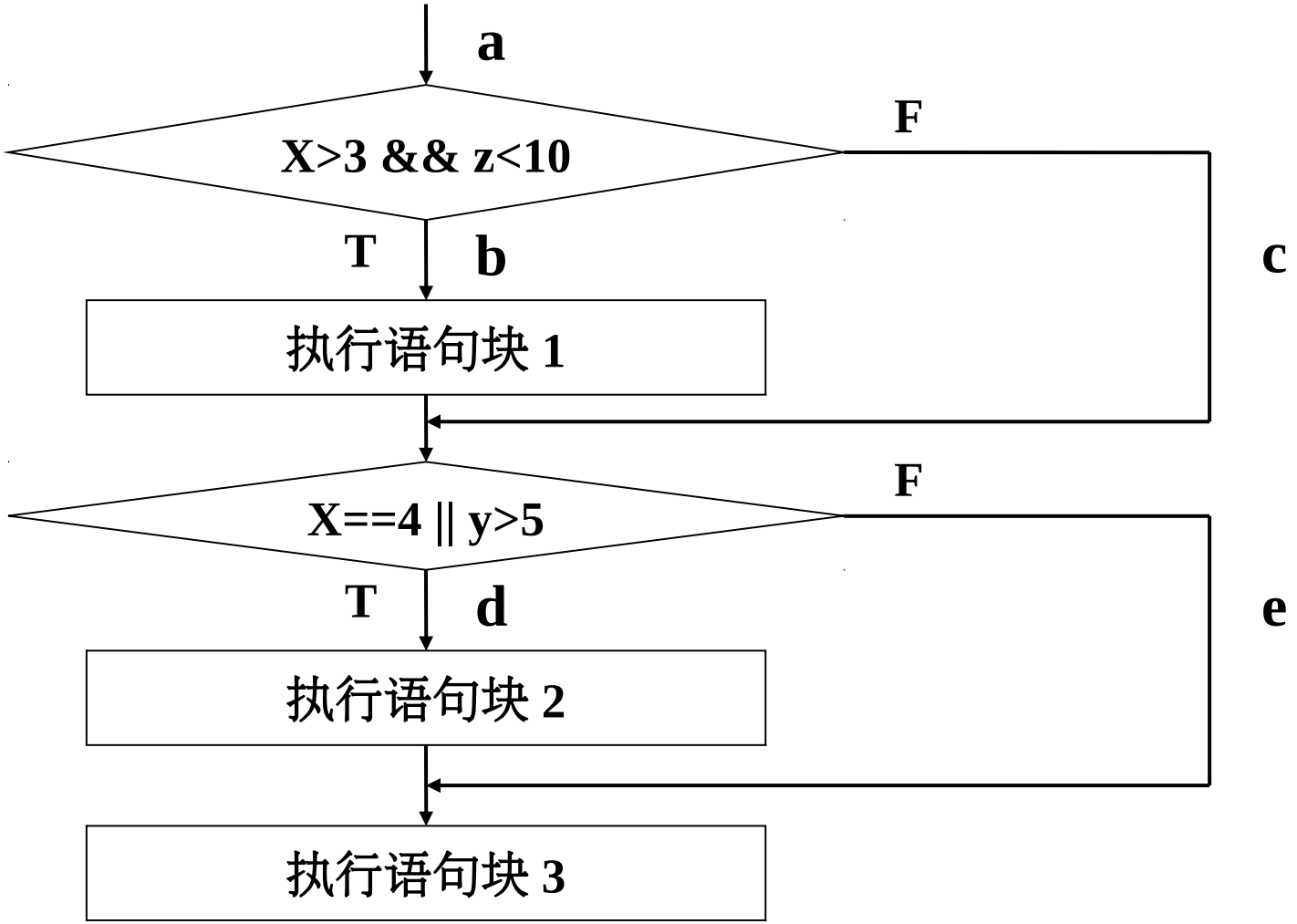
逻辑覆盖主要考察使用测试数据运行被测程序时对程序逻辑的覆盖程度。通常希望选择最少的测试用例来满足所需的覆盖标准。主要的覆盖标准有：



逻辑覆盖法

```
void  DoWork (int x,int y,int z)
{
   int  k=0,j=0;
   if ( (x>3)&&(z<10) )
   {   k=x*y-1;
       j=sqrt(k);
   }                                  // 语句块 1
   if ( (x==4)||(y>5) )
   {  j=x*y+10;   }                    // 语句块 2
   j=j%3;                             // 语句块 3
}
```

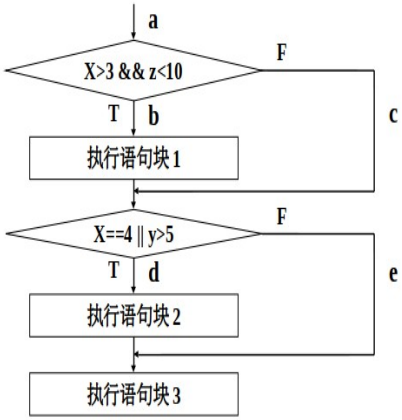
逻辑覆盖法



3 语句覆盖

- 要实现 DoWork 函数的语句覆盖，只需设计一个测试用例就可以覆盖程序中的所有可执行语句。
  - 测试用例输入为：{ x=4 、 y=5 、 z=5 }
  - 程序执行的路径是： abd
- 分析：

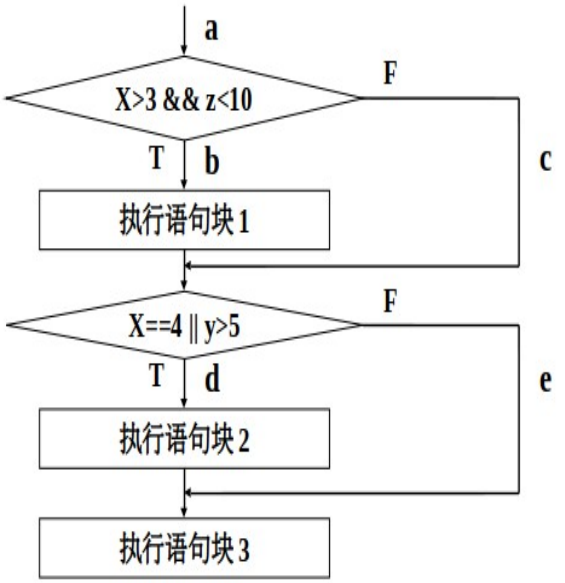
  语句覆盖可以保证程序中的每个语句都得到执行，但发现不了判定中逻辑运算的错误，即它并不是一种充分的检验方法。例如在第一个判定 ((x>3)&&(z<10)) 中把“ &&” 错误的写成了“ ||”，这时仍使用该测试用例，则程序仍会按照流程图上的路径 abd 执行。可以说语句覆盖是最弱的逻辑覆盖准则。



4 判定覆盖

- 要实现 DoWork 函数的判定覆盖，需要设计两个测试用例。
  - 测试用例的输入为： {x=4 、 y=5 、 z=5}； {x=2 、 y=5 、 z=5}
  - 程序执行的路径分别是： abd ； ace
- 分析：

  上述两个测试用例不仅满足了判定覆盖，同时还做到语句覆盖。从这点看似乎判定覆盖比语句覆盖更强一些，但仍然无法确定判定内部条件的错误。例如把第二个判定中的条件 y>5 错误写为 y<5 ，使用上述测试用例，照样能按原路径执行而不影响结果。因此，需要有更强的逻辑覆盖准则去检验判定内的条件。



5 条件覆盖

- 在实际程序代码中，一个判定中通常都包含若干条件。 条件覆盖的目的是设计若干测试用例，在执行被测程序后，要使每个判定中每个条件的可能值至少满足一次。
- 对 DoWork 函数的各个判定的各种条件取值加以标记。
  - 对于第一个判定 ((x>3)&&(z<10)) :

    条件 x>3　取真值记为 T1，取假值记为 -T1

    条件 z<10　取真值记为 T2，取假值记为 -T2
  - 对于第二个判定 ((x==4)||(y>5)) :

    条件 x==4　取真值记为 T3，取假值记为 -T3

    条件 y>5　取真值记为 T4，取假值记为 -T4

条件覆盖

- 根据条件覆盖的基本思想，要使上述 4 个条件可能产生的 8 种情况至少满足一次，设计测试用例如下：

测试用例	执行路径	覆盖条件	覆盖分支
x=4 、 y=6 、 z=5	abd	T1 、 T2 、 T3 、 T4	bd
x=2 、 y=5 、 z=15	ace	-T1 、 -T2 、 -T3 、 -T4	ce

条件覆盖

- 说明：虽然前面的一组测试用例同时达到了条件覆盖和判定覆盖，但是，并不是说满足条件覆盖就一定能满足判定覆盖。如果设计了下表中的这组测试用例，则虽然满足了条件覆盖，但只是覆盖了程序中第一个判定的取假分支 c 和第二个判定的取真分支 d，不满足判定覆盖的要求。

测试用例	执行路径	覆盖条件	覆盖分支
x=2 、 y=6 、 z=5	acd	-T1 、 T2 、 -T3 、 T4	cd
x=4 、 y=5 、 z=15	acd	T1 、 -T2 、 T3 、 -T4	cd

6 判定 / 条件覆盖

- 判定 / 条件覆盖实际上是将判定覆盖和条件覆盖结合起来的一种方法，即：设计足够的测试用例，使得判定中每个条件的所有可能取值至少满足一次，同时每个判定的可能结果也至少出现一次。
- 根据判定 / 条件覆盖的基本思想，只需设计以下两个测试用例便可以覆盖 4 个条件的 8 种取值以及 4 个判定分支。

测试用例	执行路径	覆盖条件	覆盖分支
x=4 、 y=6 、 z=5	abd	T1 、 T2 、 T3 、 T4	bd
x=2 、 y=5 、 z=15	ace	-T1 、 -T2 、 -T3 、 -T4	ce

判定 / 条件覆盖

- 分析：从表面上看，判定 / 条件覆盖测试了各个判定中的所有条件的取值，但实际上，编译器在检查含有多个条件的逻辑表达式时，某些情况下的某些条件将会被其它条件所掩盖。因此，判定 / 条件覆盖也不一定能够完全检查出逻辑表达式中的错误。
  - 例如：对于第一个判定 (x>3)&&(z<10) 来说，必须 x>3 和 z<10 这两个条件同时满足才能确定该判定为真。如果 x>3 为假，则编译器将不再检查 z<10 这个条件，那么即使这个条件有错也无法被发现。对于第二个判定 (x==4)||(y>5) 来说，若条件 x==4 满足，就认为该判定为真，这时将不会再检查 y>5 ，那么同样也无法发现这个条件中的错误。

7 组合覆盖

- 组合覆盖的目的是要使设计的测试用例能覆盖每一个判定的所有可能的条件取值组合。
- 对 DoWork 函数中的各个判定的条件取值组合加以标记：

1 、 x>3, z<10　　记做 T1 T2 ， 第一个判定的取真分支
2 、 x>3, z>=10　　记做 T1 -T2 ， 第一个判定的取假分支
3 、 x<=3, z<10　　记做 -T1 T2 ， 第一个判定的取假分支
4 、 x<=3, z>=10　　记做 -T1 -T2 ， 第一个判定的取假分支
5 、 x==4, y>5　　记做 T3 T4 ， 第二个判定的取真分支
6 、 x==4, y<=5　　记做 T3 -T4 ， 第二个判定的取真分支
7 、 x!=4, y>5　　记做 -T3 T4 ， 第二个判定的取真分支
8 、 x!=4, y<=5　　记做 -T3 -T4 ， 第二个判定的取假分支

1 黑盒测试法的概念

- 黑盒测试被称为功能测试或数据驱动测试。在测试时，把被测程序视为一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下进行。
- 采用黑盒测试的目的主要是在已知软件产品所应具有的功能的基础上，进行：
  - 检查程序功能能否按需求规格说明书的规定正常使用，测试各个功能是否有遗漏，检测性能等特性要求是否满足。
  - 检测人机交互是否错误，检测数据结构或外部数据库访问是否错误，程序是否能适当地接收输入数据而产生正确的输出结果，并保持外部信息（如数据库或文件）的完整性。
  - 检测程序初始化和终止方面的错误。

黑盒测试法的概念

- **黑盒测试**（又称行为测试）把测试对象看做一个黑盒子，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能需求。
- 黑盒测试可用于各种测试，它试图发现以下类型的错误：
  - 不正确或遗漏的功能
  - 接口错误，如输入 / 输出参数的个数、类型等
  - 数据结构错误或外部信息（如外部数据库）访问错误
  - 性能错误
  - 初始化和终止错误

黑盒测试法的概念

- 黑盒测试是依据软件的需求规约，检查程序的功能是否符合需求规约的要求。
- 主要的黑盒测试方法有：
  - **等价类划分**
  - **边界值分析**

实例　三角形问题

1 、三角形问题

输入三个整数 a 、 b 、 c ，分别作为三角形的三条边，现通过程序判断由三条边构成的三角形的类型为等边三角形、等腰三角形、一般三角形（特殊的还有直角三角形），以及构不成三角形。

现在要求输入三个整数 a 、 b 、 c ，必须满足以下条件：

条件 1　1≤a≤100

条件 2　1≤b≤100

条件 3　1≤c≤100

条件 4　a<b+ c

条件 5　b<a+ c

条件 6　c<a+ b