



# Software Testing in 1/2 hour

## Overview of concepts

Food for thought

Suggestions for further learning

Topics for discussion

Maria Grazia Pia

*INFN Genova, Italy*

Maria.Grazia.Pia@cern.ch

<http://www.ge.infn.it/geant4/training/APC2016/>



# Why testing software?

To show that it does  
what it is intended to do

To discover defects  
before it is put into use

Testing cannot demonstrate  
that the software is free of defects  
or that it will behave as specified in every circumstance

“Program testing can be used to show the presence of bugs,  
but never to show their absence!”

*E. W. Dijkstra et al., Structured programming, 1972*

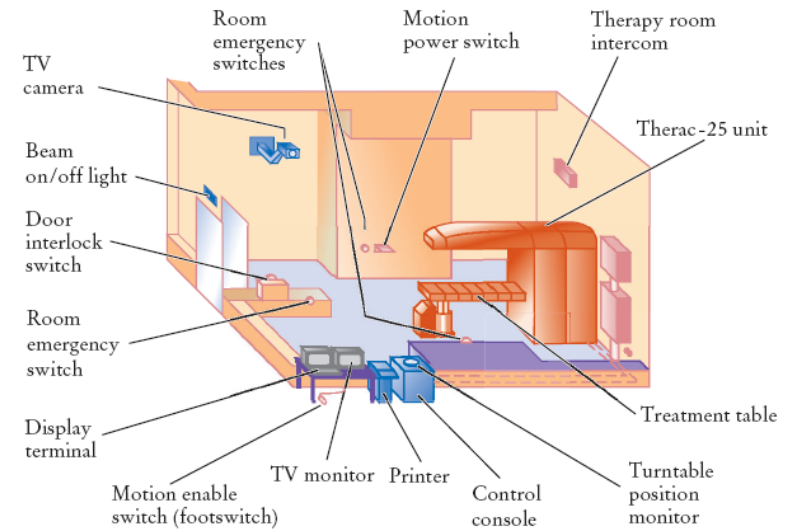


# Therac 25

Radiation therapy machine

6 massive overdose cases,  
1985-1987

Caused by  
**poor software engineering practices**



A commission concluded that the primary reason should be attributed to **bad software design** and **development practices**, and not explicitly to several coding errors that were found

The software was designed so that it was realistically impossible to test it in a clean automated way

**Safety analysis of the system excluded software**

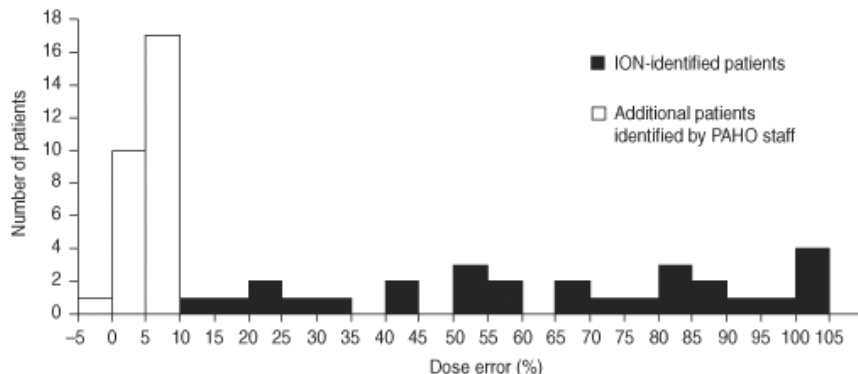
N. Leveson, C. S. Turner, **An Investigation of the Therac-25 Accidents**, *IEEE Computer*, vol. 26, no. 7, pp. 18-41, 1993.



# The Panama accidents

- 28 radiation therapy patients were overexposed in 2000-2001
- 23 of them had died by September 2005
- 18 of the deaths were from radiation effects

A modified protocol was used  
without verification and validation



The software permitted incorrect forms of data entry, which in turn led to miscalculation of treatment time

Therapy planning software from Multidata delivered **different doses** depending on the order in which data were entered

- IAEA, Investigation of an **Accidental Exposure of Radiotherapy Patients** in Panama: Report of a Team of Experts, 2001
- C. Borrás et al., Clinical effects in a cohort of **cancer patients overexposed** during external beam pelvic radiotherapy, *Int. J. Radiation Oncology Biol. Phys.*, vol.59, pp. 538-550, 2004
- C. Borrás et al., **Overexposure of radiation therapy patients in Panama**: problem recognition and follow-up measures, *Rev. Panam. Salud Publica*, vol.20, n.2-3, pp. 173-187, 2006



# Ariane 5 maiden flight

~40 seconds after initiation of the flight sequence, the launcher veered off its flight path, broke up and exploded

4 June 1996

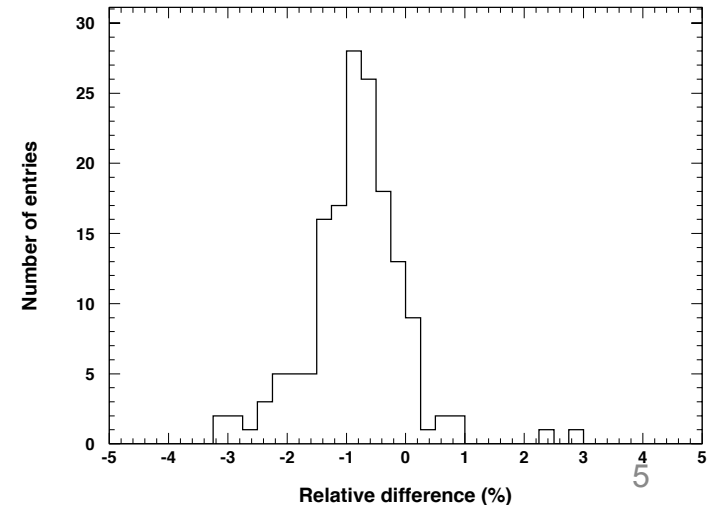
Credit: ESA

The primary cause was found to be a piece of software retained from the previous launcher systems, which **was not required** during the flight of Ariane 5. The software contained implicit assumptions about the parameters, which were **safe for Ariane 4, but not for Ariane 5.**

Relative difference of the energy deposited by  
~60 MeV protons in water, when Geant4  
Precompound model was used **standalone** or  
**through the Binary Cascade model**

M. G. Pia et al., Physics-related epistemic uncertainties of proton depth dose simulation, *IEEE Trans. Nucl. Sci.*, vol. 57, no. 5, pp. 2805-2830, 2010

Maria Grazia Pia, *INFN Genova*





# V&V

## Testing is part of the process of Software Verification and Validation

### Verification

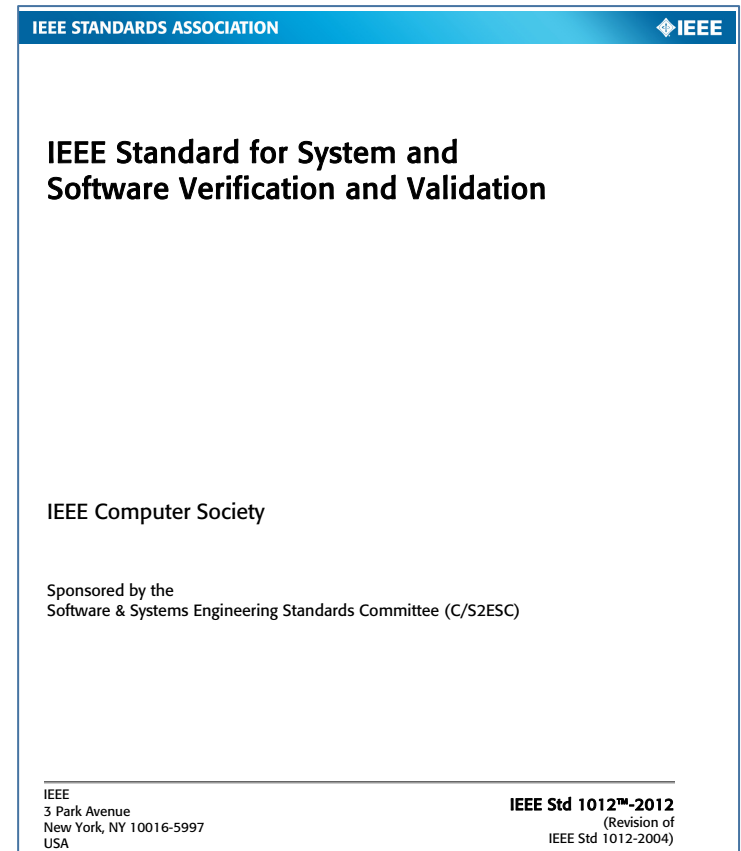
The process of providing objective evidence that the software and its associated products **conform to requirements**.

Are we building the product right?

### Validation

The process of providing evidence that the software and its associated products **solve the right problem** (*e.g., correctly model physical laws and use the proper system assumptions*), and **satisfy intended use** and user needs.

Are we building the right product?





# How do you trust the software you use?



"I'm just doing what the other ones are doing"



# Multiple perspectives

## Levels of testing

- Unit
- Integration
- System
- Acceptance

## Static testing

## Functional/non-functional testing

## Black/white-box testing

## Regression testing

## Performance testing

## Stress testing

## Configuration testing

## Security testing

## Test coverage

## Test harness

## Test automation

## Test cases

## Test planning

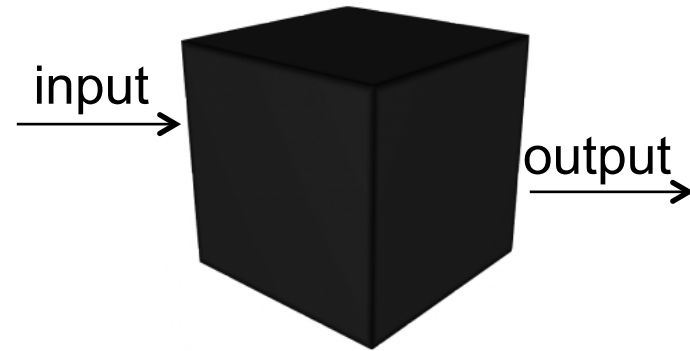
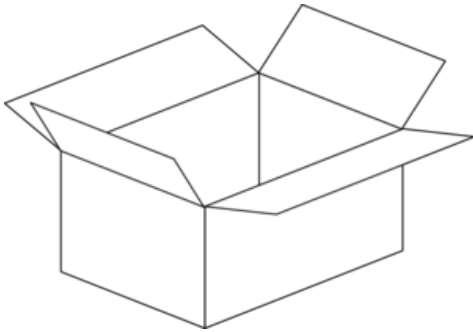
## Test frameworks

**...etc.**

*No time to cover all topics!*



# Testing strategies



- Knowledge of the **source code** is used to design defect tests
- **Systematic** approach
- Focus on program **coverage**

- Only considers software **behaviour**
- Based on **functional** specifications

**Good practice:** a combination of both strategies



# Epistemology

## How to test it? Calibrated?

...more about “magic numbers” in Refactoring

### G4HadronElastic

```
G4double dd = 10.;
G4Pow* g4pow = G4Pow::GetInstance();
if (A <= 62) {
  bb = 14.5*g4pow->Z23(A);
  aa = g4pow->powZ(A, 1.63)/bb;
  cc = 1.4*g4pow->Z13(A)/dd;
} else {
  bb = 60.*g4pow->Z13(A);
  aa = g4pow->powZ(A, 1.33)/bb;
  cc = 0.4*g4pow->powZ(A, 0.4);
}
```

### G4ChipsAntiBaryonElasticXS

```
lastPAR[43]=920.+03*a8*a3;
lastPAR[44]=93.+0023*a12;
```

### G4GoudsmitSaundersonMscModel

```
if(i>=19)ws=cos(sqrtA);
```

### G4EmCorrections

```
if(15 >= iz) {
  if(3 > j) { tet = 0.25*Z2*(1.0 + 5*Z2*alpha2/16.); }
  else      { tet = 0.25*Z2*(1.0 + Z2*alpha2/16.); }
}
```

### G4UrbanMscModel

```
coeffc1 = 2.3785 - Z13*(4.1981e-1 - Z13*6.3100e-2);
```



# Test cases

## ● Require **domain knowledge**

- Guidance from use cases, user stories, scenarios...
- Requirements, specifications and designs provide guidance at a higher level of abstraction and generality

## ● Require **source code knowledge**

- Code-based control
- Logic and sequence defects
- Initialization and data flow defects

Software testing requires studying both the **problem domain** and the **source code** in depth



# Levels of testing

## Unit testing

Test **individual pieces** of the system as they are created

## Integration testing

Test **sets** of interoperating or communicating **units or components**

## System testing

Test the **entire system**

## Acceptance testing

Demonstrate that the product is ready for release  
Usually performed by **users**



# Regression testing

A **regression** is a feature (*function, attribute*) that used to work and no longer does

It is usually related to some **change**

1. A change or bug fix creates a new bug
2. A change or bug fix reveals an existing bug
3. A change or bug fix in one area breaks something in another area

## Risk mitigation strategies

Run all tests



requires automation

Run part of the tests

based on

- traceability
- change analysis
- quality risk analysis



# Not only running code...

## Inspections and reviews

Analyze and check:

- Requirements
- Design models
- Source code
- Proposed tests
- ...

Can spot defects that  
would not be seen  
through running tests

**Pair programming** can be seen as a static testing method  
(continuous code review)

Pill of advice: train yourself to do peer reviews!

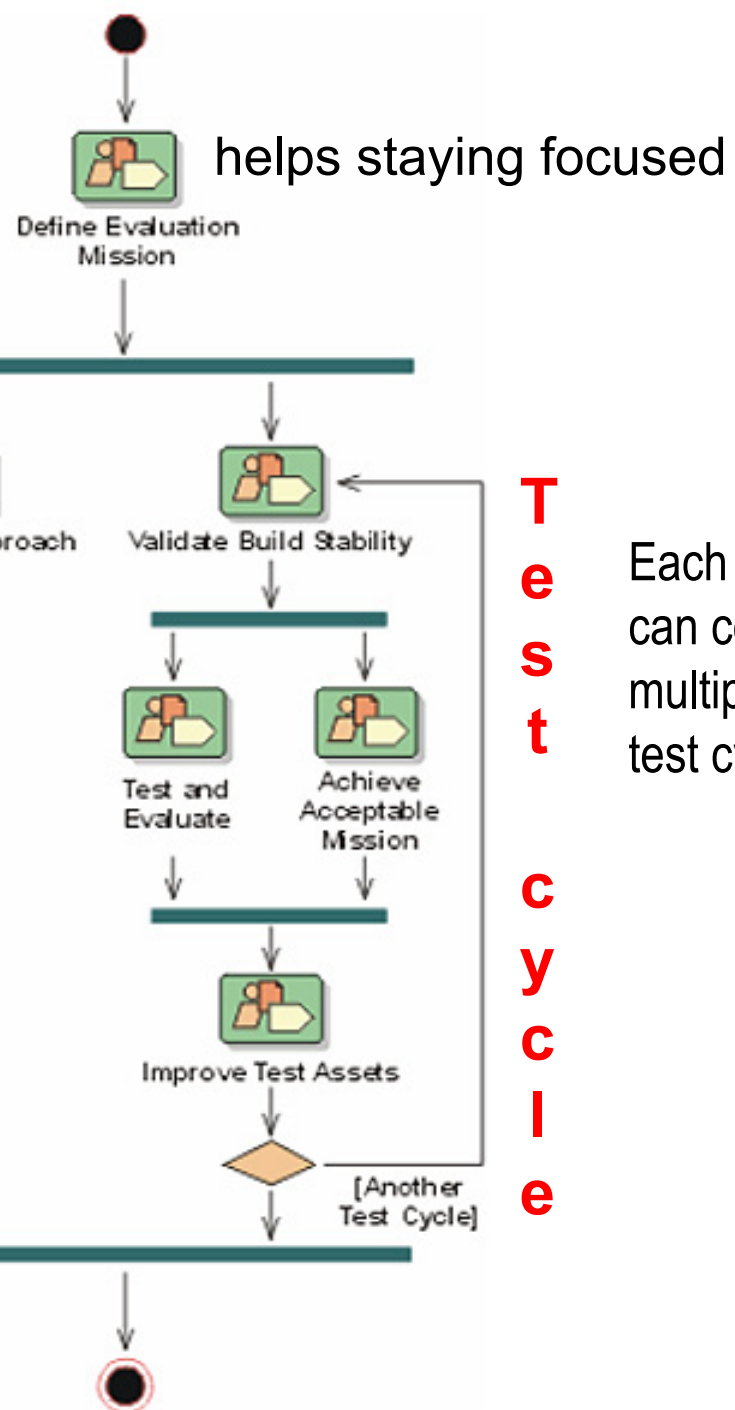


# Test process

## Workflow in the RUP

- Test Plan
- Test Ideas
- Test Cases
- Tests
- Test Suites
- Test Evaluation Summary
- Defect and Defect List
- Workload Model

Focus on **concepts**,  
not on formalities or paperwork!

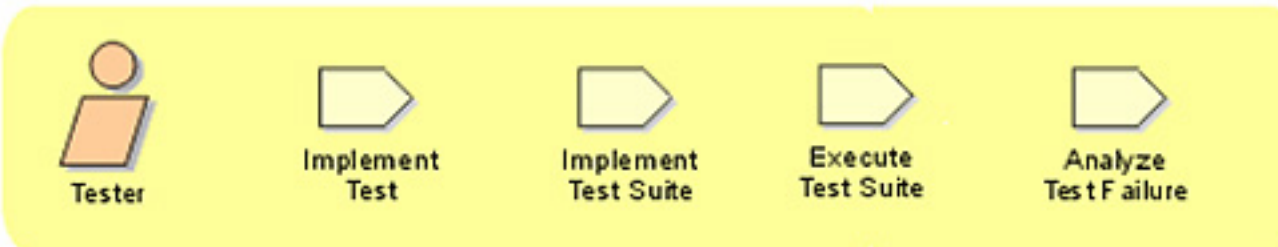
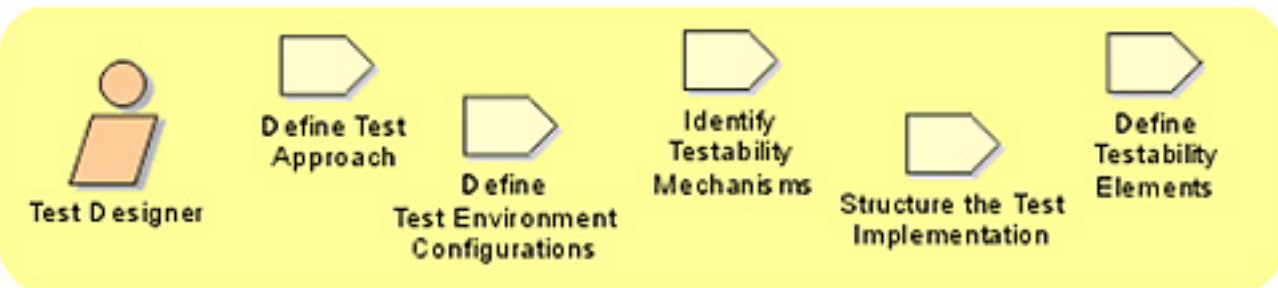
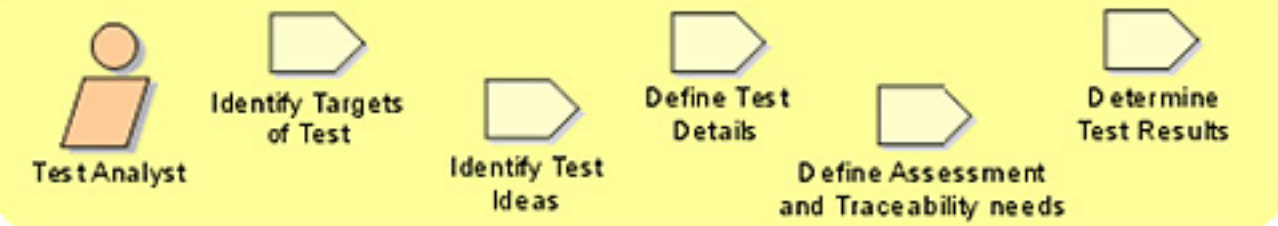




# Test process

A person can play multiple roles

**Roles  
and activities  
in the RUP**

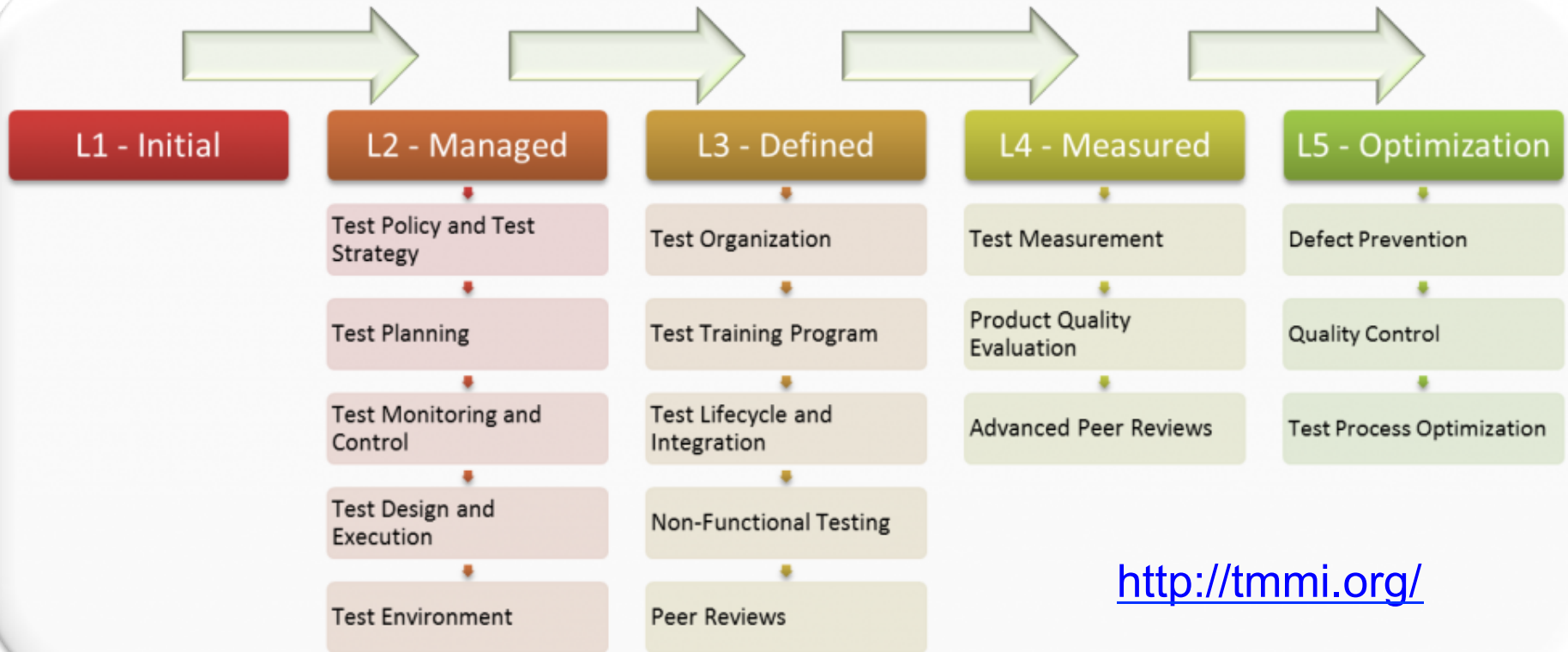


**Guidance**  
rather than  
prescription



# How good is my testing process?

## TMMi Maturity Levels



<http://tmimi.org/>

Inspiration and guidance for improvement



# Unit testing

**Unit** = smallest testable part

Procedural Programming: a **function**

Object Oriented Programming: a **class**



Unit testing contributes to **risk mitigation**:

- Issues are found at an early stage
- Can be resolved minimizing the impact on other parts of the code
- Helps in changing and maintaining the code
- Facilitates regression testing

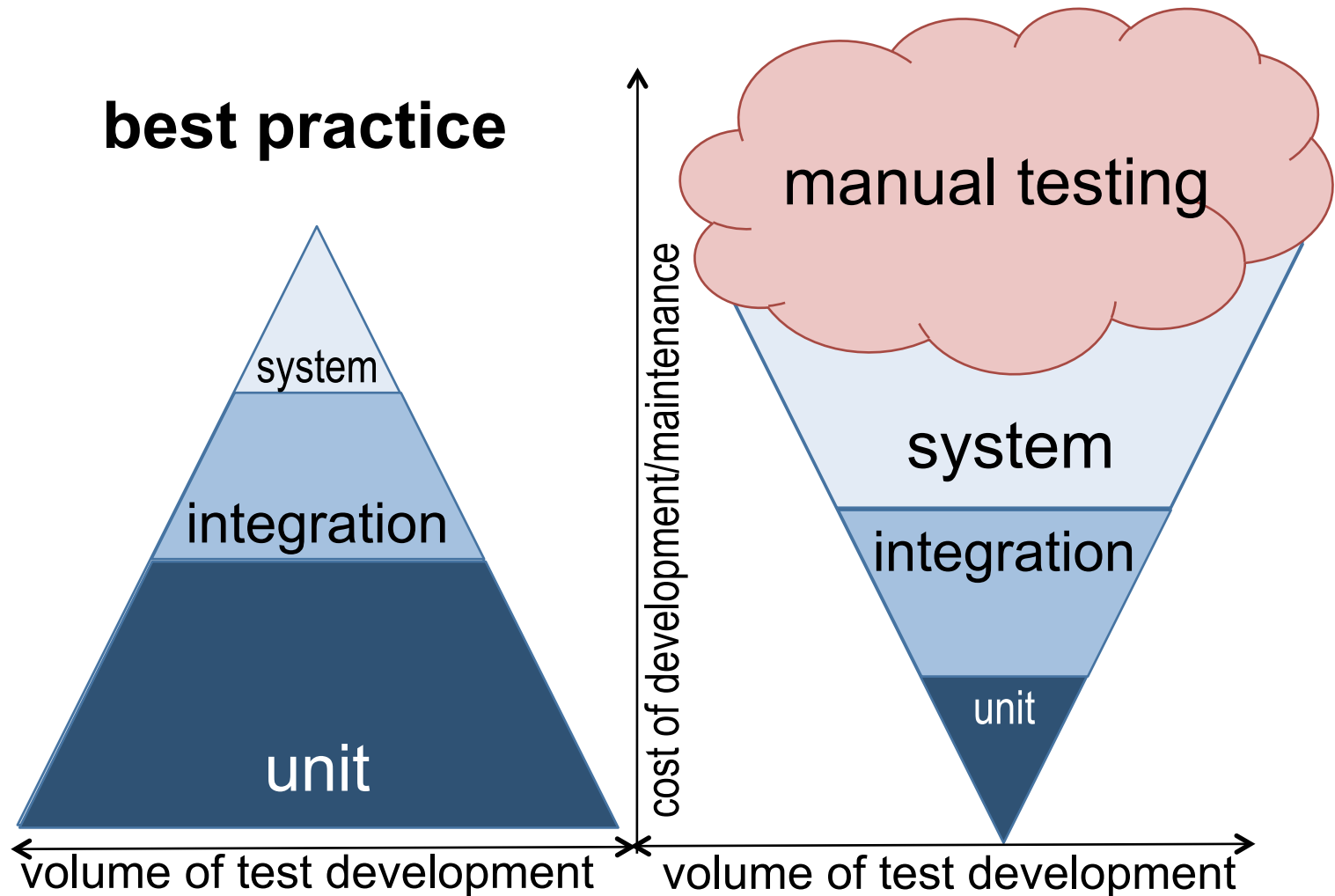


# Good characteristics of unit tests

- It should be **automated** and **repeatable**
  - Unit test frameworks
- Once a test is written, it should remain for future use
  - Unit tests should be **documented**
  - Planning and infrastructure for running
- Anyone should be able to run existing unit tests
  - Domain knowledge required to design a test
  - Not to execute it: **self-explaining** result



# Pyramid and ice-cream testing





# Unit test frameworks

Tools enabling programmers to:

- describe every test as a simple script
  - specify the **test configuration**, the **input** and an assertion describing the expected pass/fail criteria (*oracle*)
  - run a set of tests as an automated process
- 
- Pioneers: xUnit family
    - JUnit, CppUnit...
  - Modern tools
    - Boost Test Library, Google Test, catch, cgreen...



# Google unit test framework

AKA Google Test, googletest, gtest

- Unit testing framework for C++ code
- Follows xUnit concept
- Open source

## Assertion

Check whether a condition is true  
*Success, non-fatal failure, fatal failure*

## Tests

Use assertions

## Test case

Contains one or more tests

## Test program

Can contain multiple test cases

## Test fixture

A class for common objects and  
functionality shared by multiple tests



# Example: unit test for a square root function

```
double square-root (const double);
```

```
#include "gtest/gtest.h"
```

Creates a hierarchy named SquareRootTest

```
TEST(SquareRootTest, PositiveNos) {
```

← PositiveNos is a test

```
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}
```

execution continues even if there is a failure

```
TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}
```

aborts if there is a failure

```
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

initializes the framework  
automatically detects and runs all the tests  
defined using the TEST macro



# A test fixture class

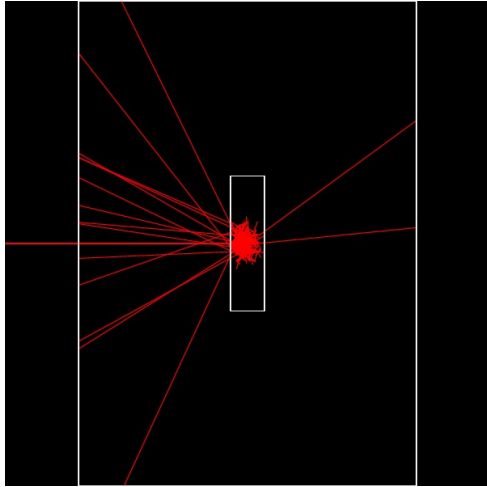
```
class myTestFixture1: public ::testing::test {  
public:                                     derived from the ::testing::test class in gtest.h  
    myTestFixture1( ) {  
        // initialization code here  
    }  
  
    void SetUp( ) {  
        // code here will execute just before the test ensues  
    }  
  
    void TearDown( ) {  
        // code here will be called just after the test completes  
        // ok to through exceptions from here if need be  
    }  
  
    ~myTestFixture1( ) {  
        // cleanup any pending stuff, but no exceptions allowed  
    }  
  
    // put in any custom data members that you need  
};
```



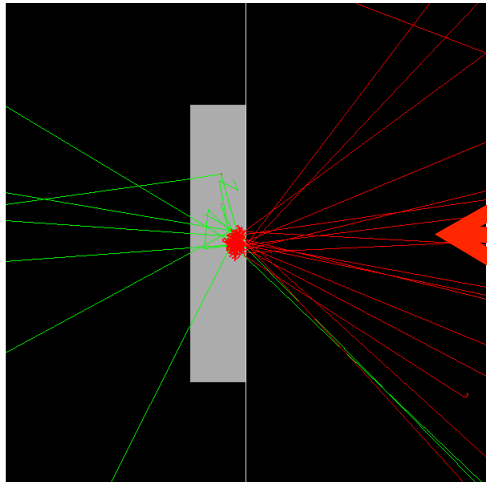
# Passed $\neq$ exempt from flaws

“Testing shows the presence, not the absence of bugs.” E.W. Dijkstra,  
*Software Engineering Techniques*, J. N. Buxton and B. Randell eds., 1970

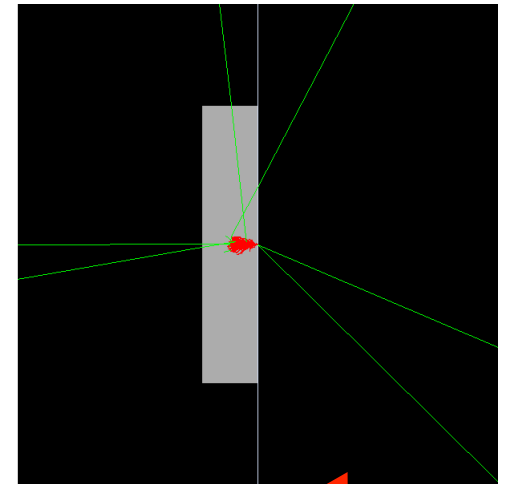
G4eSingleCoulombScatteringModel  
Geant4 10.2



← Developer's test



Physical expectation: a  
fraction of electrons is  
backscattered  
(consistent with experiment)



↗ No backscattered  
electrons are observed



# Testable software

- Detector design, experimental strategies, physics results depend critically on software
- ...which is often untested (*partially tested*) because it is untestable
  - or became untestable in the course of its evolution
- **Making software testable**
  - Improving software design (*refactoring*)
  - Breaking dependencies (*techniques à la Feathers*)
  - **Embedding testability in the software design**
- Testability must be **maintained**
  - through the evolution of the whole software system
- **Epistemological issues**
  - domain knowledge and implementation details



# Discipline of software engineering

- Most of these problems can be easily solved if we simply write tests as we develop our code
  - ...and we **maintain** the tests
  - ...and we **regularly execute** them
  - ...and we **investigate** the reasons for failure

**If a test is hard to write, that means that we have to find a different design which is testable**

- It is always possible
- **Software design reviews:** care about testability



# Hands-on exercise

- Refactoring is intertwined with unit testing
- In the refactoring exercise get acquainted with:
  - Associating software development and change with unit testing
  - A unit test framework: **GoogleTest**
  - Test **automation**
  - **Regression** testing

**Apply what you learn at the APC school to your own software development environment!**