TVVS - *Software Testing, Verification and Validation*

Mutation Testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Agenda

- What is quality?
- Some concepts
- Mutation Testing

What is quality?

ISO/IEC 25010:2011 defines:

- A **product quality** model composed of eight characteristics (which are further subdivided into sub characteristics) that relate to static properties of software and dynamic properties of the computer system. The model is applicable to both computer systems and software products.
- A **quality in use** model composed of five characteristics (some of which are further subdivided into sub characteristics) that relate to the outcome of interaction when a product is used in a particular context of use. This system model is applicable to the complete human-computer system, including both computer systems in use and software products in use.

ISO/IEC 25010:2011- Product quality model

The product quality model defined in ISO/IEC 25010 comprises the eight quality characteristics shown in the following figure:

Functional Suitability

- This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following sub characteristics:
  - **Functional completeness.** Degree to which the set of functions covers all the specified tasks and user objectives.
  - **Functional correctness.** Degree to which a product or system provides the correct results with the needed degree of precision.
  - **Functional appropriateness.** Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

Performance efficiency

- This characteristic represents the performance relative to the amount of resources used under stated conditions. This characteristic is composed of the following sub characteristics:
  - **Time behaviour.** Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
  - **Resource utilization.** Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
  - **Capacity.** Degree to which the maximum limits of a product or system parameter meet requirements.

Compatibility

- Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment. This characteristic is composed of the following sub characteristics:
  - **Co-existence.** Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
  - **Interoperability.** Degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.

Usability

- Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. This characteristic is composed of the following sub characteristics:
  - **Appropriateness recognizability.** Degree to which users can recognize whether a product or system is appropriate for their needs.
  - **Learnability.** degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
  - **Operability.** Degree to which a product or system has attributes that make it easy to operate and control.
  - **User error protection.** Degree to which a system protects users against making errors.
  - **User interface aesthetics.** Degree to which a user interface enables pleasing and satisfying interaction for the user.
  - **Accessibility.** Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

Reliability

- Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time. This characteristic is composed of the following sub characteristics:
  - **Maturity.** Degree to which a system, product or component meets needs for reliability under normal operation.
  - **Availability**. Degree to which a system, product or component is operational and accessible when required for use.
  - **Fault tolerance.** Degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
  - **Recoverability.** Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

Security

- degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. This characteristic is composed of the following sub characteristics:
  - **Confidentiality.** Degree to which a product or system ensures that data are accessible only to those authorized to have access.
  - **Integrity.** Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
  - **Non-repudiation.** Degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
  - **Accountability.** Degree to which the actions of an entity can be traced uniquely to the entity.
  - **Authenticity.** Degree to which the identity of a subject or resource can be proved to be the one claimed.

Maintainability

- This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. This characteristic is composed of the following sub characteristics:
  - **Modularity.** Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
  - **Reusability.** Degree to which an asset can be used in more than one system, or in building other assets.
  - **Analyzability.** Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
  - **Modifiability.** Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
  - **Testability.** Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

Portability

- Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another. This characteristic is composed of the following sub characteristics:
  - **Adaptability.** Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
  - **Installability.** Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment.
  - **Replaceability.** Degree to which a product can replace another specified software product for the same purpose in the same environment.

ISO/IEC 25010:2011- Quality in use

- **Effectiveness** - *The accuracy and completeness with which users achieve specified goals*
- **Efficiency**- *The resources expended in relation to the accuracy and completeness with which users achieve goals*
- **Satisfaction**- *The degree to which user needs are satisfied when a product or system is used in a specified context of use*
  - Usefulness
  - Trust
  - Pleasure
  - Comfort

ISO/IEC 25010:2011- Quality in use

- **Freedom from risk** - degree to which a product or system mitigates the potential risk to economic status, human life, health, or the environment
  - Economic risk mitigation
  - Health and safety risk mitigation
  - Environmental risk mitigation
- **Context coverage** - degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in both specified contexts of use and in contexts beyond those initially explicitly identified
  - Context completeness
  - Flexibility

Agenda

- What is quality?
- **Some concepts**
- Mutation Testing

Some concepts

- Error / Defect / Failure – What is the difference?

Analyst/Designer/Programmer makes a mistake/error.

Defect appears in the program.

Defect remains undetected during testing.

The program fails during execution i.e. it behaves unexpectedly.

Error / Defect / Failure

- **error**: A human action that produces an incorrect result.
- **defect:** A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.
- **failure:** Deviation of the component or system from its expected delivery, service or result..

Verification vs Validation (V&V)

- **Verification** - Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled. [ISTQB]
  - ensures that you **"build it right"**, i.e., that the work products and the final product meet **specified requirements**

"Are we building the product right?"

- **Validation** - Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled. [ISTQB]
  - ensures that you **"build the right thing"**, i.e., that the product will fulfill its **intended use** in its intended environment

"Are we building the right product?"

Agenda

- What is quality?
- Some concepts
- **Mutation Testing**

Mutation Testing

What tests the tests?

Mutation Testing

- **Mutation Testing**: Testing in which two or more variants of a component or system are executed with the same inputs, the outputs compared, and analyzed in cases of discrepancies [ISTQB]
- **Mutation Analysis**: A method to determine test suite thoroughness by measuring the extent to which a test suite can discriminate the program from slight variants (mutants) of the program. [ISTQB]
- **Mutation Score:** The calculation to measure the quality of a test suite detecting the introduced faults in the mutants. **It is defined as the quotient between the number of dead mutants and the number of non-equivalent mutants.** It is given as a percentage.

Mutation Testing

- Mutation testing is a **fault-based testing technique** that is based on the assumption that a program is well tested if
- all simple faults are predicted and removed;
- complex faults are combinations of simple faults and are therefore detected by tests that detect single faults.

Mutation Testing

- Mutation testing is **used to test the quality of your test suite**. This is done by mutating certain statements in your source code and checking if your test code is able to find the errors.
  - How do you know that you can trust your unit tests?
  - How do you know that they're really telling you the truth?
  - If they don't find a bug, does that really mean that there aren't any?
  - What if you could test your tests?

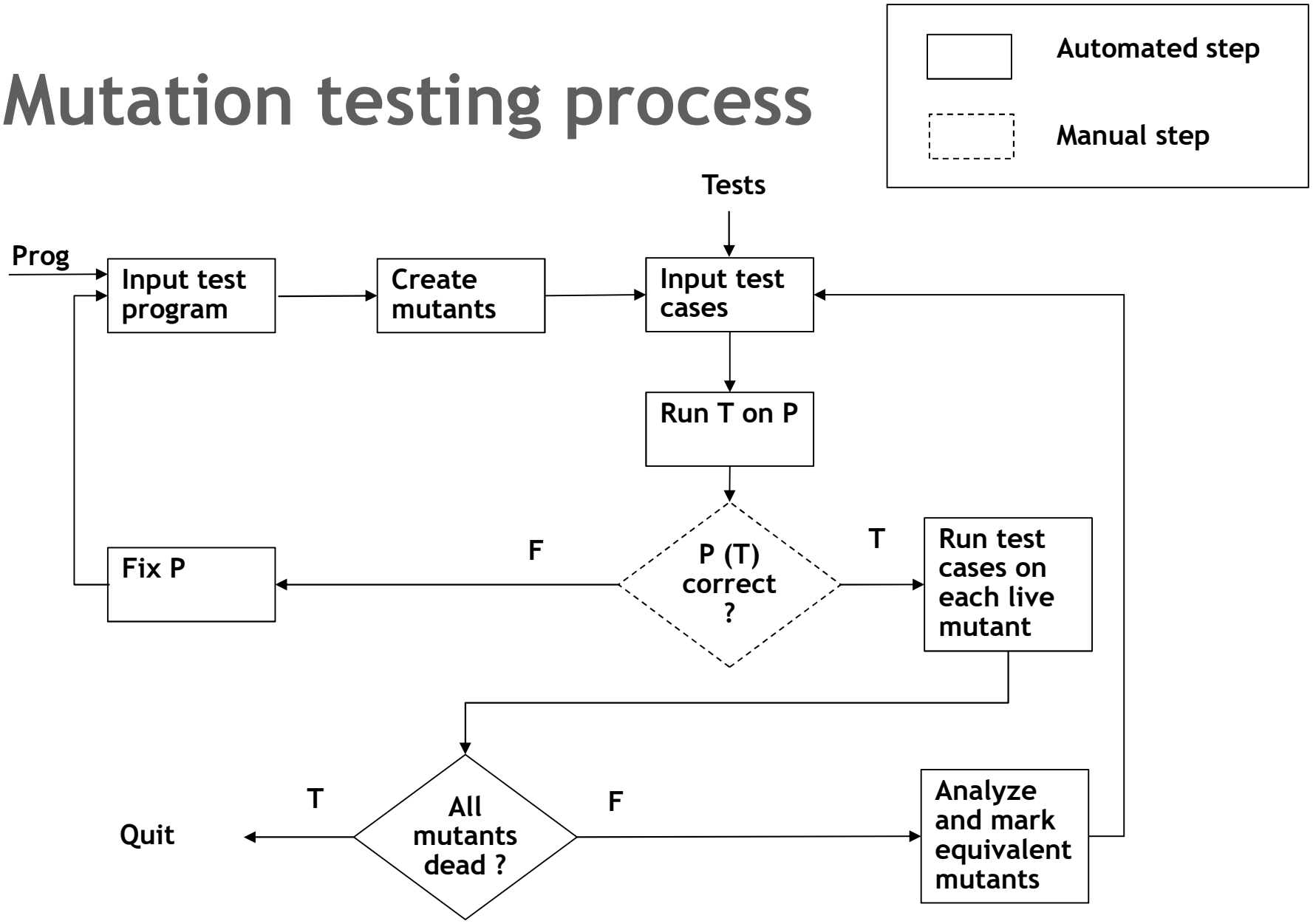
Mutation testing

- **Mutation testing** is a method of software testing, which involves modifying program's source code or byte code in small ways.
  - In short, any tests which pass after code has been mutated are defective.
- These, so-called ***mutations***, are based on well-defined ***mutation operators*** that either mimic typical programming errors
  - (such as using the wrong operator or variable name) or force the creation of valuable tests (such as driving each expression to zero).
- The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Mutation Testing

- To assess the quality of the tests by performing them on mutated code
- Help construct *more adequate* tests
- Produce a suite of *valid tests* which can be used on real programs

Mutation testing process



Mutation testing process

- Starts with a code component and its associated test cases (in a state such that the code passes all test cases)
- The original code component is **modified** in a simple way (replace operators, constants, etc.) to provide a set of similar components that are called mutants, based on typical errors
- The original test cases are **run** with each mutant
- **Live mutants** cannot be distinguished from the original program (parent)
- Distinguishing a mutant from its parent is referred to as **killing** such mutant

Mutation testing process

- If a mutant remains **live** (passes all the test cases), then either
  - the mutant is **equivalent** to the parent (and is ignored), or
  - it is not equivalent,
- in which case additional test cases should be developed in order to kill such mutant
- The rate of mutants "killed" (after removing mutants that are equivalent to the original code) gives an indication of the rate of undetected defects that may exist in the original code

Mutation testing

- A **mutation operator** is a rule that is applied to a program to create mutants. Typical mutation operators, for example, replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements.

Traditional Mutation Operators

Examples of traditional mutation operators

- **Deletion of a statement**
- **Boolean**
  - Replacement of a **statement** with another, e.g., `==` and `>=`, `<` and `>`
  - Replacement of **Boolean expressions** with true and false, e.g., `a||b` with `True`
- Replacement of **arithmetic**, e.g., `*` and `+`, `/` and `-`
- Replacement of **variable** (ensuring same scope/type)

Method-level Mutation Operators

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Deletion
COD	Conditional Operator Insertion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Mutation operators

- Beside this, there are mutation operators for **object-oriented** languages, for concurrent constructions, complex objects like containers etc. They are called class-level mutation operators.
- For example, *muJava* classifies class mutation operators into four groups: Encapsulation, Inheritance, Polymorphism and Java-Specific Features.

Inter-Class Mutation Operators

Language Features	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHM	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	Super keyword insertion
	ISD	Super keyword deletion
	IPC	Explicit call to a parent's constructor type

Inter-Class Mutation Operators

Language Features	Operator	Description
Polymorphism	PNC	New method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Class type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Argument overloading method call change

Inter-Class Mutation Operators

Language Features	Operator	Description
Java-specific features	JTI	*this* keyword insertion
	JTD	*this* keyword deletion
	JSI	*static* modifier insertion
	JSD	*static* modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Some examples

ENCAPSULATION

Example: AMC – Access modifier change:

Original Code	AMC Mutants
Public Stack s;	private Stack s; protected Stack s; Stack s;

INHERITANCE

Example: IHD – Hiding variable deletion

Original Code	IHD Mutan

class List {
  int size;
  ...
}
class Stack extends List {
  int size;
  ...
}
	class List {
	  int size;
	  ...
	}
	class Stack extends List {
	  // int size;
	  ...
	}

More examples: https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf

Types of Mutants

- Based on the types of faults seeded, mutants can be classified as
  - **First Order Mutants** (FOMs) and
  - **Higher Order Mutants** (HOMs).
- First order mutants seed only simple faults, generated by a single syntactic change to the original program.
- Higher order mutants combine simple first order faults to simulate more complex faults.

Weak/Strong mutation testing

Imagine the following statement of a program:

if (a && b) c=1; else c=0;

- The condition mutation operator would replace “&&” with ‘||’ and produce the following mutant:

if (a || b) c=1; else c=0;

Weak/Strong mutation testing

Now, for the test to kill this mutant, the following condition should be met:

1) Test input data should cause different program states for the mutant and the original program. For example, a test with a=1 and b=0 would do this (it goes through **else** in the original and through **then** in the mutated).

2) The value of ‘c’ should be propagated to the program’s output and checked by the testing.

Weak/Strong mutation testing

- ***Weak mutation testing*** (or *weak mutation coverage*) requires that only the first condition is satisfied. ***Strong mutation testing*** requires that both conditions are satisfied.
  - Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.
- **Weak Mutation Testing** is a coverage measure - i.e. tells you about the code that is run by your tests
- **Strong Mutation Testing** measures whether your code needs to be like it is to pass the tests.

Problems with mutation testing

- 1. High computational cost
- 2. Trivial mutants
- 3. Equivalent mutants (human effort)
- 4. Oracle (human effort)

1. Computational effort

- One of the barriers to the practical use of mutation testing is the unacceptable computational expense of generating and running vast numbers of mutant programs against the test cases.
- The number of mutants generated for a software unit is proportional to the product of the number of data references and the number of data objects. Typically, this is a large number for even small software units. Because each mutant program must be executed against at least one, and potentially many, test cases, mutation analysis requires large amounts of computation.

1. Computational effort

- Approaches to reduce this computational expense usually follow one of three strategies: **do fewer**, **do smarter**, or **do faster**.
  - The **do fewer** approaches seek ways of running fewer mutant programs without incurring intolerable information loss.
  - The **do smarter** approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution.
  - The **do faster** approaches focus on ways of generating and running each mutant program as quickly as possible.

2. Trivial Mutants

- Traditional mutation testing has only applied **first order mutants**.
- However, many of the first order mutants generated by these simple syntactic fault insertions are **readily killed by the simplest of test cases executed**, leading **to much wasted effort** killing rather trivial mutants.
- As a result, many mutation testers observe that even the **most trivial, small and unimaginative test suite will kill a very large proportion of the first order mutants**.

3. Equivalent mutants

- Many mutation operators can produce equivalent mutants. For example, Boolean relation mutation operator will replace "==" with ">=" and produce the following mutant:

original

```
int index=0;
while (…) { …;
  index++;
  if (index==10)
    break;
}
```

mutant

```
int index=0;
while (…) { …;
  index++;
  if (index>=10)
    break;
}
```

- However, it is not possible to find a test case which could kill this mutant. The resulting program is equivalent to the original one. Such mutants are called equivalent mutants.

3. Equivalent mutants

- Equivalent mutants detection is **one of biggest obstacles** for practical usage of mutation testing. The effort, needed to check if mutants are equivalent or not, can be very high even for small programs.

4. Oracle

- The human **oracle problem** refers to the process of **checking the original program's output** with each test case.
- Strictly speaking, this **is not a problem unique to mutation** testing. In all forms of testing, once a set of inputs has been arrived at, there remains the problem of checking the output.
- However, **mutating testing is effective** precisely because it is a demanding test, and this **can lead to an increase** in the number of test cases, thereby increasing the oracle cost.
- The **oracle cost** is often the **most expensive** part of the overall test activity

Some Tools

- Mothra
- Proteum
- muJava
- MuClipse
- Jumble
- JesTer (for Java)
- PesTer (for Python)
- SQLMutation
- Nester (for C#)
- PIT
- …

References

- http://www.mutationtest.net/
- http://en.wikipedia.org/wiki/Mutation_testing
- Black, P. E., V. Okun, et al. (2000). Mutation of Model Checker Specifications for Test Generation and Evaluation. Mutation 2000, Jan Jose, California, Kluwer Academic Publishers.
- Murnane, T. and K. Reed (2001). On the Effectiveness of Mutation Analysis as a Black Box Testing Technique. Software Engineering Conference, Australian.
- Ammann, P. E., P. E. Black, et al. (1998). Using Model Checking to Generate Tests from Specifications. 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia, IEEE Computer Society.
- http://cs.gmu.edu/~nli1/oomutation/oomuts.pdf
- https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf
- Higher Order Mutation Testing by Yue Jia; http://www0.cs.ucl.ac.uk/staff/M.Harman/yue-phd.pdf

Exercise

```
1:  bool A,B,C; int V;
2:  read(A);read(B);read(C);read(V);
3:  while (V<20)  {
4:        if ( A /\ ( B \/ C ) )
5:            V:= V+5;
6:        if (A /\ ~B)
7:            V:=V+10;
8:  }
9:  print V;
```

Id	Mutation
1	3: while (V<=20)
2	4: if (A \/ (B \/ C))
3	6: if (A \/ ~B)

Conceive test cases (with the 100% degree of coverage) for the program considering the following mutants. Indicate that the value returned by the program in each of these test cases.

Exercise

```
1: read(A); read(B); read(C);
2: W := 0;
3: while (C>B AND C>A) {
4:         C := C-5;
5:         if (C>10)
6:             W:=5;
7:         if (C==A+B OR C>A-B)
8:             W:=10;
9:         else if (C==10)
10:                W:=15;
13: }
14: print(C);print(W);
```

Id	Mutation
1	3: while (C>B OR C>A)
2	5: if (C<10)
3	9: else if (C<>10)

Conceive test cases (with the 100% degree of coverage) for the program considering the following mutants. Indicate that the value returned by the program in each of these test cases.