Strategie per il progetto dei casi di test

Strategie

- **Finora abbiamo utilizzato strategie cosiddette esplorative, basate sulla conoscenza del dominio del problema e sulla scelta intuitiva delle prove da effettuare**
- **Ora vedremo alcune strategie un po' più sistematiche per trovare i casi di test**

Classi di equivalenza

Input domain partitioning

- **I dati di input ed output possono essere in genere suddivisi in classi dove tutti i membri di una stessa classe sono in qualche modo correlati.**

- **Ognuna delle classi costituisce una classe di equivalenza (una partizione) ed il programma si comporterà (verosimilmente) nello stesso modo per ciascun membro della classe.**

  •Matematicamente, una partizione in classi di equivalenza produce insiemi disgiunti la cui unione è l'insieme totale di partenza. Inoltre la relazione di equivalenza tra partizioni gode delle proprietà riflessiva, simmetrica e transitiva

- **I casi di Test dovrebbero essere scelti all'interno di ciascuna partizione.**

Esempio : somma

- **Se stiamo provando la funzione somma, allora è interessante vedere se il nostro programma funziona bene quando con i numeri negativi oltre che con i positivi**
- **Per questo motivo, per il primo addendo consideriamo due classi di equivalenza:**
  - Numeri >= 0
  - Numeri negativi
- **Facciamo lo stesso anche per il secondo addendo**

Esempio : classi di equivalenza per la somma

- **Abbiamo due input indipendenti (gli addendi) con due classi di equivalenza ognuno**
- **Una buona idea sarebbe quella di fare almeno quattro casi di test:**
  - Un numero >=0 sommato ad un altro numero >=0
  - Un numero >=0 sommato ad un negativo
  - Un negativo sommato ad numero >=0
  - Un numero negativo sommato ad un altro numero negativo

Esempio : casi di test per la somma

- **I casi di test si ottengono scegliendo un elemento per ogni insieme (a piacere)**

- **Esempio:**

  a =5, b = 37, expected = 42

  a =50, b = -8, expected = 42

  a = -15, b = 57, expected = 42

  a = -20, b= -22, expected = -42

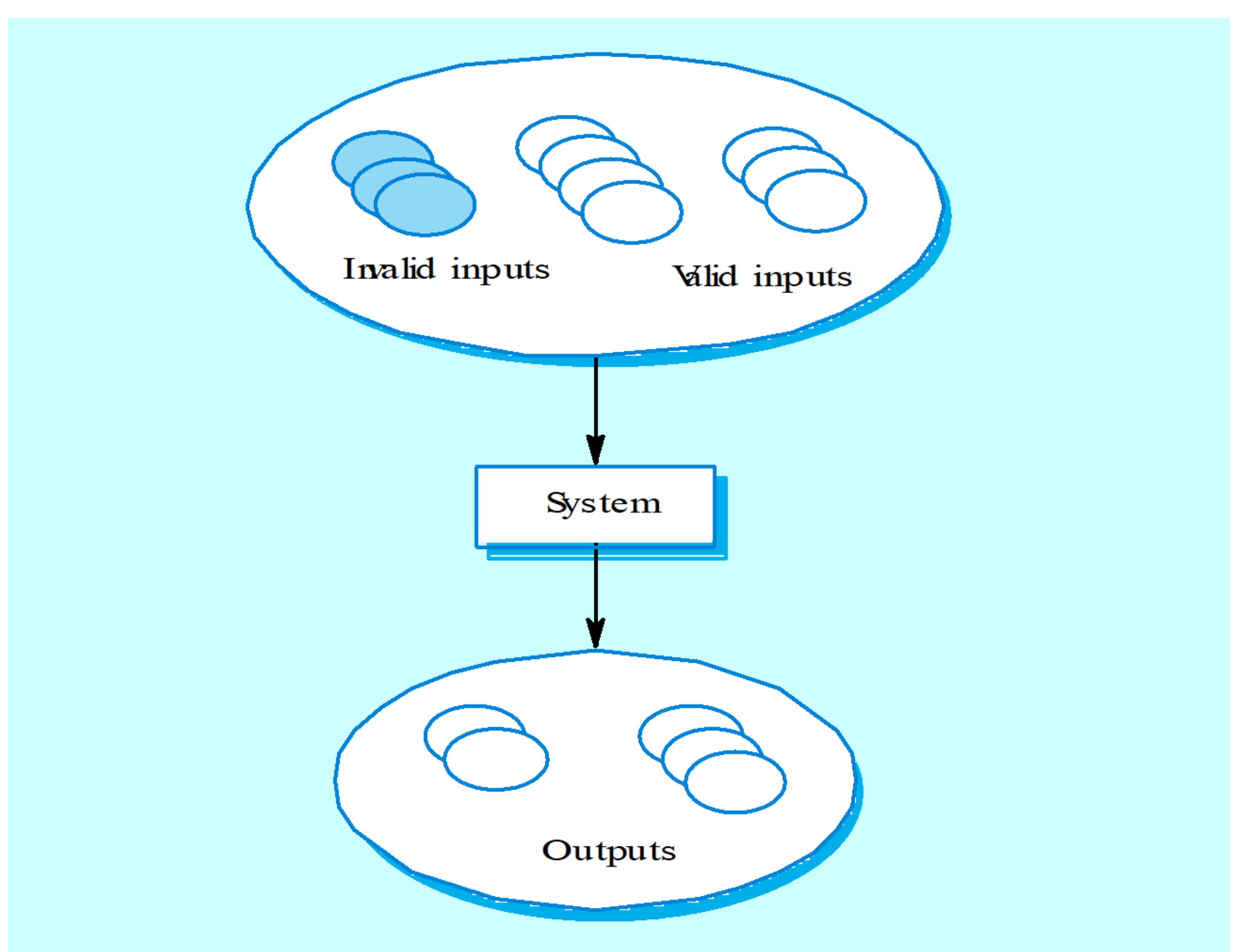
Input Domain Partitioning techniques

- **Input fields value can be divided into classes where all members of the same class are somehow related.**
  - **Each of the classes constitutes an equivalence class**
- **The rationale is that we hypothesize that the program should behave in the same way for each member of the class.**
- **Test cases consists of a selection of values belonging to equivalence classes**
  - **Coverage criteria can be designed based on the coverage of the equivalence classes for each input**

Equivance class partitioning

- **The equivalence classes for an input represent a partition of the input domain:**
  - **The union of all the equivalence classes corresponds to the input domain**
  - **There are no intersections between the equivalence classes**

Equivalence partitioning



Example : calendar

- **In a method you have to enter a date, consisting of day (integer), month (string), year (integer)**
- **The method must correctly recognize between valid dates (corresponding to days that actually existed) and invalid dates**
- **The method returns a boolean (valid date if true) and a string (the day of the week or "Error" in case of invalid date)**

Domain information

- **Source: https://it.wikipedia.org/wiki/Calendario_gregoriano**
- **In particular:**
  - **Months have different durations**
  - **February 29th exists only in leap years**
  - **Leap years are divisible by 4, but not by 100**
  - **The calendar came into force on October 15, 1582**

More details

- **Input:**
  - Day – integer in a limited range below and above
  - Month – string in a set including 12 valid values
  - Year – integer in a bounded range below
- **Output**
  - Validity – boolean
  - Day of the week – string in a set including 7 valid values and an invalid value (Error)

General cases

- If the input is a:
  - range of values
    - *a valid class for values within the range, an invalid class for values below the minimum, and an invalid class for values above the maximum*
  - specific value
    - *a valid class for the specified value, an invalid class for lower values, and an invalid class for higher values*
  - element of a discrete set
    - *a valid class corresponding to the collection (minimal testing) or a valid class for each element of the collection (detailed technique), an invalid class for an element not belonging to that set*
  - A boolean value
    - *As in the previous case, but for a discrete set of two values (true, false)*
- In some cases, an extra invalid class including all the remaining values that could be set as input should be considered

Valid and invalid classes: an example

- **Input:**
  - Day – integer in a limited range below and above
    - {day≤0}, {1≤day≤31}, {day>31}
  - Month – string in a set of 12 values
    - {month ∈ {January, February, March, April, May, June, July, August, September, October, November, December}}, {month ∉ {January, February, March, April, May, June, July, August, September, October, November, December}}
  - Year – integer in a bounded range below
    - {year<1582}, {year>=1582}

Warning

- **Equivalence classes are sets of possible test values**
- **Test cases consists of the specification of one value for each input**
    - The equivalence class in which the selected input value is, has been covered by the test case

Equivalence classes Coverage criteria

- **Given the equivalence classes, we may fix testing criteria aiming at the coverage of them, in different ways:**
  - Each Choice Coverage (ECC)
  - Base Choice Coverage (BCC)
  - Pair-Wise Coverage (PWC)
  - T-Wise Coverage (TWC)
  - All Combinations Coverage (ACC)

Each Choice Coverage

- Minimum coverage of equivalence classes
  - Each equivalence class is covered by at least one test case
    - *Minimum number of test cases equal to the maximum number of input classes with multiple equivalence classes*
      - **Based on the interface-based example**
      - **We omit tc4 in case of compiled languages because the type check is done by the compiler so we cannot run tests with incorrect type values**

Test case	**TC1**	**TC2**	**TC3**	**TC4**
Day	1	0	35	first
Month	January	Brumaire	January	January
Year	1980	1492	1980	two thousand

Discussion

- It is the technique that guarantees the coverage of the equivalence classes with the minimum number of test cases
  - Maximum efficiency
- But, if a defect is found, it can be difficult to individuate the cause.
  - If TC1 fails and TC2 does not fail, the failure can be caused by any of the input values or by their interaction
    - *Lacks in fault localization*

Base Choice Coverage

- Coverage of adjacent equivalence classes
  - Each equivalence class is covered by at least one test case
  - For each test case there is at least another one that differs for only one equivalence class
    - *The number of test cases is in the order of the total amount of equivalence classes*

- *Q : number of inputs*
- *B_i : number of classes for input number i*
- *The number of test cases is :*

$$1 + \sum_{i=1}^{Q} (B_i - 1)$$

Base Choice Coverage

- The three equivalence classes have cardinality 3, 2, 2
- The first test (base test) covers one class for each input
- The other tests each cover only one class not yet covered.

- Number of tests = sum of classes – number of inputs +1
- Example:
  - (3+2+2) – 3 + 1 = 5

Test case	TC1	TC2	TC3	TC4	TC5
Day	1	0	35	1	1
Mont	January	January	January	Brumaire	January
Year	1980	1980	1980	1980	1492

Discussion

- Less efficient technique than the previous one
  - The number of test cases grows linearly with the number of inputs and equivalence classes
    - *The technique is inspired by Hamming's distance theory*
      - **The chosen test suite has a Hamming distance of 1**
- More useful for debugging purposes
  - If only one test fails, then there is another one that differs by only one input value that does not fail: there is a good chance that the failure was triggered by the only different value between the two tests
    - *It may happen, however, that the failures caused by a pair of values are not individuated*

All combinations coverage (ACC)

- Coverage of all equivalence class combinations
  - *The number of test cases equals to the product of the cardinalities of the quantities of equivalence classes of each class*
  - *Corresponds to the t-wise case when t=n =number of inputs*

Test case	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
Day	**1**	**0**	**35**	**1**	**0**	**35**
Month	**January**	**January**	**January**	**Brumaire**	**Brumaire**	**Brumaire**
Year	**1980**	**1980**	**1980**	**1980**	**1980**	**1980**
Test case	**TC7**	**TC8**	**TC9**	**TC10**	**TC11**	**TC12**
Day	**1**	**0**	**35**	**1**	**0**	**35**
Month	**January**	**January**	**January**	**Brumaire**	**Brumaire**	**Brumaire**
Year	**1492**	**1492**	**1492**	**1492**	**1492**	**1492**

Discussion

- Testing of all combinations is "exhaustive" testing of equivalence classes
  - *May find any defect that arises from a specific combination of equivalence classes*
- Suitable for critical systems where effectiveness drives the selection of the testing strategy

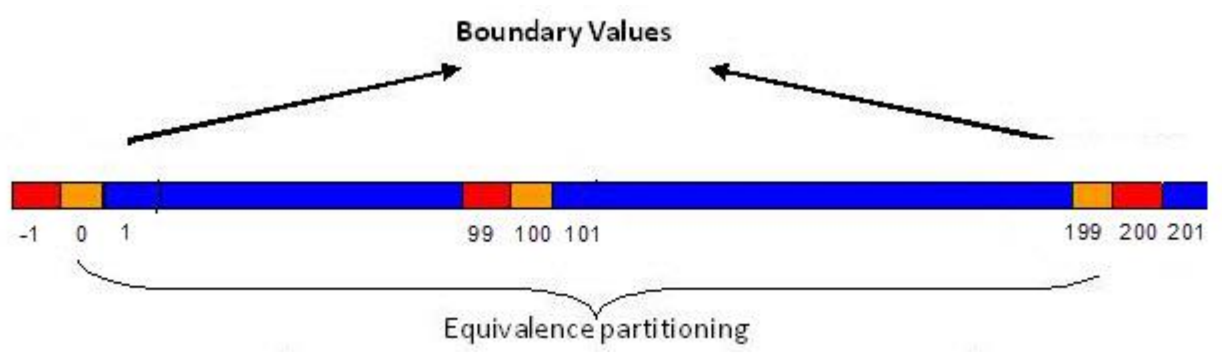
Analisi dei valori limite

Boundary value analysis

- A variant to the equivalence class technique consider limit values (boundaries)
- Boundary values form additional valid and invalid equivalence classes corresponding to the limit values of the data validity sets
- Usually, boundary values regard intervals
- Boundary values are also values for which it is assumed that there may be a particular behavior with respect to some operation
  - for example the zero value for an integer that might fit into a division or for a pointer

Examples of boundary values

- If the condition on input variables specifies:
  - (Closed) range of values
    - *–Boundary classes: minimum of the range, maximum of the range (valid classes), slightly below minimum, slightly above maximum (invalid classes)*
    - *–If there are more ranges, the reasoning is iterated for each of them*
  - Integer values
    - *•A boundary class, regardless of specification, is the set {0}; another, unless otherwise considered, is the class of negative numbers, and so on*



Examples of boundary classes

- For the input Day:
  - **{Day <<0}: lower value of the lower bound of the range**
  - {0}: slightly lower value of the lower bound of the range and also null value
  - {1}: lower end
  - {2}: value slightly higher than the lower bound
  - **{Day >>2 && Day << 27}: valid value away from boundaries**
  - {27}: value slightly lower than the upper bound
  - {28}: upper bound in some cases
  - {29}: well-known critical value
  - {30}: well-known critical value
  - {31}: well-known critical value
  - {32}: value slightly greater than the upper bound
  - **{Day >> 31}: value greater than the upper bound of the range**

- For the input input Year
  - **{Day << 1582}: lower value of the lower bound of the range**
  - *{1581}: slightly lower value of the lower bound of the range*
  - {1582}: lower bound and critical value
  - {1583}: value slightly higher than the lower bound
  - **{Year >> 1583}: value higher than the upper bound**

(non-boundary values in bold)

Examples of boundary classes

- For the input Month:
  - **{Month $\notin$ mesi dell'Year}**
  - {January}
  - {february}
  - {march}
  - {april}
  - {may}
  - {june}
  - {july}
  - {august}
  - {september}
  - {october}
  - {november}
  - {december}
  -

Alternatively, classes could be considered:

- {Month $\notin$ year months}
- {Month with 31 days}
- {Month with 30 days}
- {february}

(non-boundary values in bold)

Examples of boundary classes

- The three classes have now cardinality (12, 4, 5)
-
  - ECC: 12 test cases
  - BCC: 12 + 4 + 5 – 3 + 1 = 19 test cases
  - Pair wise: 60 test cases

  - ACoC: 12 * 4 * 5 = 240 test cases

- To cover cases such as April 31, pair wise testing is now sufficient
- We probably do not cover February 29 of a Leap Year, because we have not specified the existence of leap years

Ulteriore suddivisione

- For the input Year
  - **{Day << 1582}: lower value of the lower bound of the range**
  - {1581}: slightly lower value of the lower bound of the range
  - {1582}: lower bound and critical case
  - {1583}: value slightly higher than the lower bound
  - **{Year >> 1583, Leap year}**

- The three classes have now cardinality (12, 4, 5)

  - ECC: 12 test cases
  - BCC: 12 + 4 + 5 – 3 + 1 = 19 test cases
  - Pair-wise: test cases
  - ACoC: 12 * 4 * 5 = 240 test cases

To cover february 29 of a leap year we need a 3-way testing (all combinations)

Tabelle di decisione

Functionality-based approaches

- **Let us now focus on the behaviors expected by the System**
  - In the case of the calendar method we must imagine specific situations to want to test
    - E.g. February 29 of a Leap Year, Day 31 of a Month with 30 days, ...
      - Design tests that cover these requirements
      - Decision Table Technique

Decision tables

- Decision tables are a tool for specifying test cases in which;
  - Different combinations of inputs correspond to different outputs/actions;
  - The various combinations can be represented as mutually exclusive Boolean expressions;
  - The result should not depend on previous inputs or outputs, nor on the order in which the inputs are provided.
- Decision Tables are primarily a design technique, but they are also useful to support testing

Building the Decision Table

- The columns in the Table represent the combinations of inputs to which the different actions correspond.
- The rows in the table show the values of the input variables (in the Conditions section) and the actions that can be performed (in the Actions Section)
- Each distinct combination of inputs is called an Instance.

Exercise

- Write the decision table about the validity of a date that takes into account the following constraints:
  - April, June, September, November have 30 days
  - February has 28 days in non-leap years, 29 otherwise
  - Leap years are all years divisible by 4 and not divisible by 100
  - Years divisible by 400 are leap years
  - The calendar is valid from 15 October 1582

Example: Validity of the date

		Varianti																
Conditions	Day	[1,28]	29	29	29	30	30	31	31	>31	<1	[1,31]	[1,31]	[1,31]	[1,30]	[1,31]	[1,14]	[15,31]
	Month	[1,12]	≠2	2	2	≠2	2	(2,4,6,9,11)	(1,3,5,7,8,10,12)	[1,12]	[1,12]	∉[1,12]	[1,12]	[1,9]	[11]	[12]	10	10,
	Year	>1583	>1583	>1583 bisest	>1583 Non bis	>1583	>1583	>1583	>1583	>1583	>1583	>1583	<1582	1582	1582	1582	1582	1582
Actions	Valid	True	True	True	False	True	False	False	True	False	False	False	False	False	True	True	False	True

Example: Validity of the date

		Varianti															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Conditiont	Day	[1,28]	29	29	{29,30}	31	30	31	Any	Any	<15	[15,31]	[1,31]	[1,30]	<1	>31	Any
	Month	Any	2	2	$\neq 2$	$\in\{1,3,5,7,8,10,12\}$	2	$\in\{2,4,6,9,11\}$	Any	<10	10	10	{10,12}	{11}	Any	Any	$\neq [1,12]$
	Year	>1582	>1582 Bisest	>1582 Non bisest	>1582	>1582	Any	Any	<1582	1582	1582	1582	1582	1582	Any	Any	Any
Actions	Valid	True	True	False	True	True	False	False	False	False	False	True	True	True	False	False	False

Explicit and Implicit Variants

- In the table, the logical operator between the conditions is And;
- In the previous example we have 23 conditions on inputs and 15 significant variants, but in general there are more possible combinations.
  - How many combinations of conditions are generally possible?
    - *For n conditions, 2^n variants (but not all are plausible) are called implicit variants.*
  - The number of explicit (significant) variants is usually smaller!

Discussion

- Test cases can be automatically generated from decision tables
  - Oracles can be also automatically generated, if we have filled actions
  - There are frameworks that support writing decision tables, generating, executing, and evaluating test cases, especially to support acceptance testing activities.
    - *E,g, Fitnesse: http://www.fitnesse.org/*
- Decision tables require effort for their design but provide optimal effectiveness and efficiency (in the context of black box techniques)
  - Taking into account the additional constraints defined, their effectiveness is equal to that of a combinatorial testing with 8 * 8 * 4 = 256 test cases

Testing strutturale (white box)

Testing strutturale

- **E' un tipo di testing più preciso (potenzialmente più efficace) rispetto a quello funzionale (black box)**
- **Guardiamo il codice e ci concentriamo su ciò che non è stato ancora eseguito**
- **Fino ad ottenere una copertura «ottimale»**
  – Ci piacerebbe ottenere il 100% ma potrebbe esistere qualche parte del codice non eseguibile
    - Ad esempio obsoleta

Debugging

Debugging

- Research and correction of faults that are the cause of failures.
- It consists of two phases:
  - Finding a defect
  - Correction of the defect

- Debugging is far from formalized
- Debugging methodologies and techniques are above all an element of the programmer/tester's experience

Fault Localization

- The problem to find the fault on the basis of the failure
- There are many techniques to reduce the distance between defect fault and failure:
  - Probes, to monitor the value of variables during the execution
    - Probes may be simple output operations, log writing operations
  - Assertions, to highlight when anomalies occurs
    - Assertion may throw exceptions or break the execution of the program
  - Conditional breakpoints, to stop the execution when specific conditions occur

Debugging features

- **Breakpoints**
  - IDE user-defined artificial program interruptions
    - The IDE communicates the existence of breakpoints to the JVM. In debugging execution mode the occurrence of a breakpoint is observed by the JVM that stops the execution of the process (or only one of its threads)
  - Conditional breakpoints:
    - Breakpoints that are triggered only when you switch to a certain line and a certain condition occurs
  - Breakpoints dependent on the Hit Count:
  - Breakpoints that fire only after a certain number of steps on that line of code
    - Conditional breakpoints can be set in Eclipse by selecting "Breakpoint properties“ in the breakpoint context menu

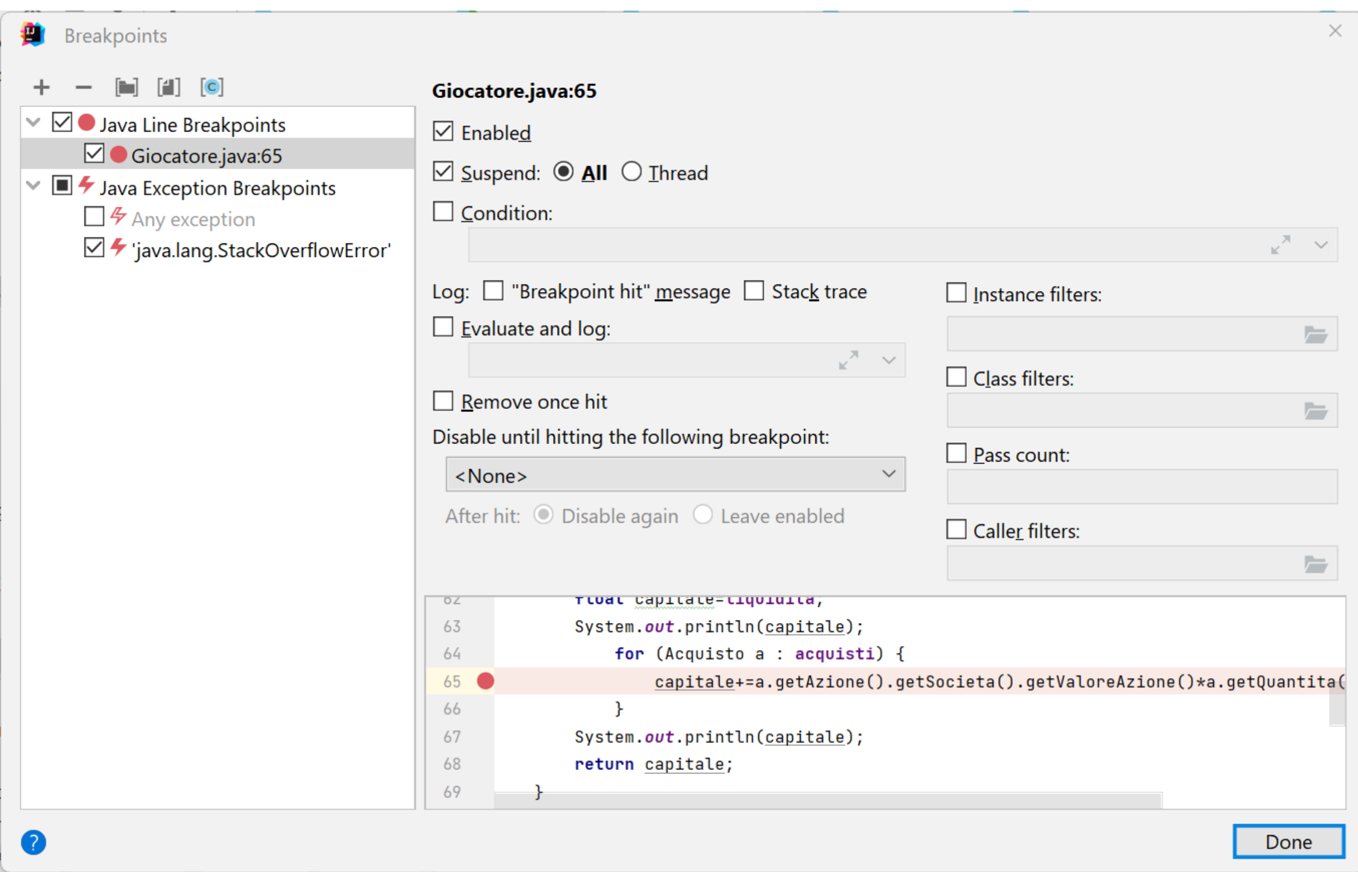
Debugging features

- **Step-by-step code execution**
  - Step Over : execute the current line of code
    - If the line of code contains one or more function calls they are all executed and the execution breaks at the completion of the execution of the line of code
  - Step Into : execute the next (bytecode) statement
    - In case there are function calls in the line of code with the breakpoint, the next step will consist in the start of the first function called into the statement
  - Step Out : the execution continues until it is reached the exiting from the current executed function
  - Resume : the execution continues until another breakpoint is encountered

- **Between the execution of two steps the program is suspended and the programmer can:**
  - Read variables values
  - Change variables values (not in all languages)
  - Evaluate the result of logical expressions
  - Call functions (not in all languages)

- **Beware: the execution in step by step mode is not faithful to the real one from a temporal point of view!**

Advanced breakpoint properties (Intellij)



Advanced breakpoint properties (IntelliJ)

- **Conditional breakpoint**
  - The breakpoint stops the execution only if a certain Boolean expression is satisfied
  - Conditional breakpoint are the most common feature supporting fault localization, thus they help the developer in finding a point into the code from which we start the step-by-step execution
- **Log**
  - Write a log line when the breakpoint is activated
    - It allows us to have a probe without changing the source code
- **Instance filters / Class filters / Caller filters**
  - Additional filters to limit the cases in which the breakpoint is activated
- **Pass count**
  - The breakpoint is activated only when it is executed a certain number of times
    - Very useful with long loops
- **Exception Breakpoint**
  - The breakpoint is executed after an exception occurrence

Watchpoint

- **A watchpoint is a breakpoint linked not to a line of code but to an attribute of a class**
  - It causes breaking of the execution each time the attribute is read (Field Access) or written (Field Modification)
  - Through its properties it is possible to customize when to suspend, with the same options that there are for breakpoints

- **Watchpoints can be used to manually perform data flow analysis**
  - They can be used to reconstruct the history of the variation of the values of a variable which final value is not correct

Test dei difetti : Mutation Testing

Mutanti

- **L’unico caso nel quale veramente possiamo misurare quanti difetti sono stato trovati è quello nel quale i difetti ce li abbiamo inseriti direttamente noi!**
  - Oppure li ha inseriti un apposito programma al posto nostro

- **Perché inserire difetti nel codice?**
  - Per sapere se i nostri test sarebbero in grado di trovare veri difetti se ci fossero

- **A mutant is killed (or retired) when:**
  - there is at least one test case whose result differs when run on the original program and on the mutant program
  - The result is generally given by assertion value, but it could also be a regular termination vs crash
- **A mutant survives a test suite if:**
  - For all test cases in the test suite, no distinction is made on the result of any test whether it is run on the original program or on the mutant program

Examples of mutation operators

- **Depending on the code element on which they apply:**
  - Changes on variables and types
  - Changes on arrays (e.g. index) and lists
  - Changes on operators (assignment, arithmetic, logical)
  - Changes to the prototype of a function/method/service
  - Changes of modifiers (eg static, transient, synchronized, final, ...)
  - Changes in inheritance or polymorphism (e.g. casting, super, override, ...)
  - https://pitest.org/quickstart/mutators/

Come ritirare un mutante

- **Un mutante viene ritirato da un test quando l'esito del test è diverso se eseguito sul programma originale e su quello mutante**

- **Il caso più semplice: se eseguo il test sul programma normale non causa un fallimento, mentre se lo eseguo sul programma mutante causa fallimento**
  - Per trovare i mutanti dobbiamo essere anche bravi a scrivere le asserzioni

Mutanti equivalenti

- **Alcuni mutanti sono troppo semplici da trovare**
  - Se il programma mutante non compila allora non c'è bisogno di test per scoprirli → INUTILI !
  - Se il programma mutante viene scoperto da ogni test allora non ci aiuta a capire quali sono i test migliori → INUTILI
- **Alcuni mutanti sono impossibili da trovare**
  - Ad esempio modificano una parte di programma nella quale è impossibile andare → INUTILI
    - Si chiamano Mutanti Equivalenti

Example

Program

```
int add(int x, int y) {
  return x + y;
}
```

Test suite

t_1: `assertEquals(0, add(0, 0))`

t_2: `assertEquals(1, add(1, 0))`

t_3: `assertEquals(2, add(2, 0))`

Mutants

```
int add(int x, int y) {
  return x - y;
}
```
m_1

```
int add(int x, int y) {
  return x * y;
}
```
m_2

```
int add(int x, int y) {
  return x++ + y;
}
```
m_3

Example

Program

```
int add(int x, int y) {
  return x + y;
}
```

Test suite

```
t1: assertEquals(0, add(0, 0))
t2: assertEquals(1, add(1, 0))
t3: assertEquals(2, add(2, 0))
```

Mutants

```
int add(int x, int y) {
  return x - y;
}
```
m_1

```
int add(int x, int y) {
  return x * y;
}
```
m_2

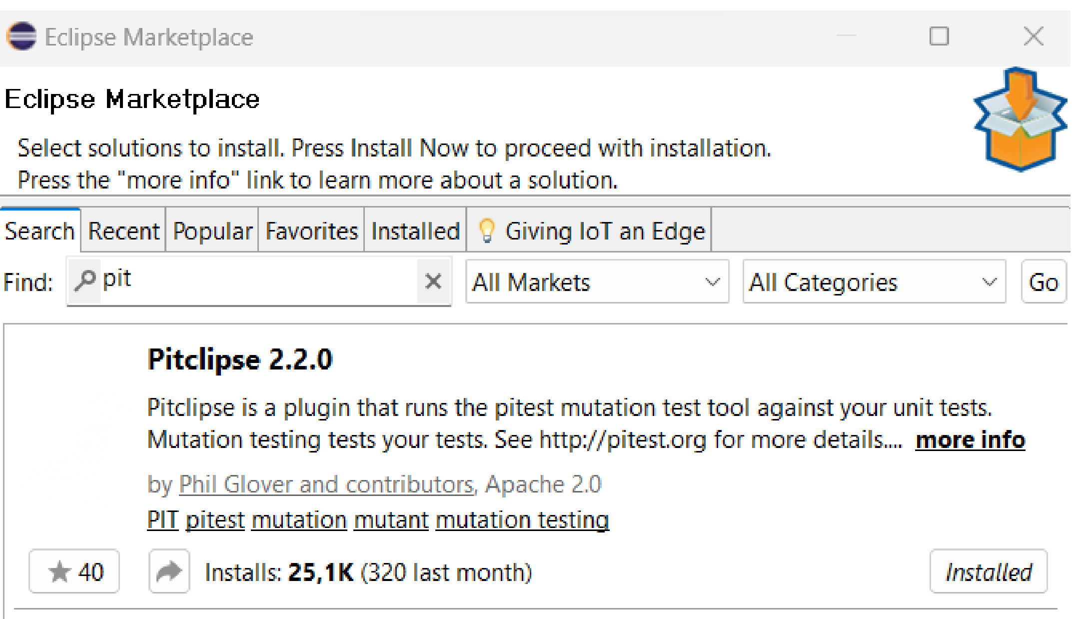
```
int add(int x, int y) {
  return x++ + y;
}
```
m_3

- **T2 kills the mutant M2 because the result of the test is positive on the original program, failure on M2**
- **T3 kills the mutant M2 because the result of the test is positive on the original program, failure on M2**

- **In total, only one of the three mutants was "killed", while the other two survived.**
- **Our test suite is unable to recognize potential faults of m1 and m3**
- **The m3 mutant is called an equivalent mutant because it can be shown that there is no test that can detect it.**
  - In this case x (local variable) is used in addition, then incremented but never reused
- **Our test suite was able to kill half of the mutants that could be killed.**

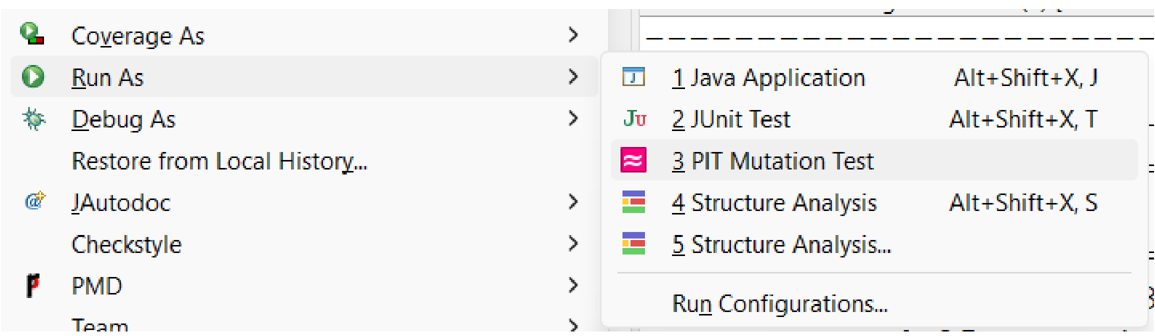
Mutation based testing with pitclipse

- **Pitclipse is an eclipse extension**
- **It allows mutation-based testing in the context of the Eclipse IDE**



Using Pitclipse

- **PIT can be used in Eclipse by the Run as contextual menu**
- **It can be applied on a JUnit test suite without failing test cases**



Pitclipse Output

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	90%	9/10	88%	22/25	88%	22/25

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
default	1	90%	9/10	88%	22/25	88%	22/25

Report generated by PIT 1.6.8

```
================================================================================
- Statistics
================================================================================
>> Line Coverage: 11/13 (85%)
>> Generated 25 mutations Killed 22 (88%)
>> Mutations with no coverage 0. Test strength 88%
>> Ran 287 tests (11.48 tests per mutation)
```

Pitclipse Output

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	90%	9/10	88%	22/25	88%	22/25

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
TriangleType.java	90%	9/10	88%	22/25	88%	22/25

Report generated by PIT 1.6.8

Pitclipse Output

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:test3WiseACOC(int, int, int, java.lang.String)]/[test-template-invocation:#15] (8 ms)
- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:testBCCExtended(int, int, int, java.lang.String)]/[test-template-invocation:#6] (6 ms)
- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:test3WiseACOC(int, int, int, java.lang.String)]/[test-template-invocation:#19] (8 ms)
- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:testBCCExtended(int, int, int, java.lang.String)]/[test-template-invocation:#2] (9 ms)

Pitclipse Output

Mutations

Line	Mutations
16	1. changed conditional boundary → SURVIVED 2. changed conditional boundary → SURVIVED 3. changed conditional boundary → SURVIVED 4. negated conditional → KILLED 5. negated conditional → KILLED 6. negated conditional → KILLED
17	1. replaced return value with null for TriangleType::triangle → KILLED
20	1. changed conditional boundary → KILLED 2. changed conditional boundary → KILLED 3. changed conditional boundary → KILLED 4. Replaced integer addition with subtraction → KILLED 5. Replaced integer addition with subtraction → KILLED 6. Replaced integer addition with subtraction → KILLED 7. negated conditional → KILLED 8. negated conditional → KILLED 9. negated conditional → KILLED
21	1. replaced return value with null for TriangleType::triangle → KILLED
24	1. negated conditional → KILLED 2. negated conditional → KILLED
25	1. replaced return value with null for TriangleType::triangle → KILLED
28	1. negated conditional → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
29	1. replaced return value with null for TriangleType::triangle → KILLED
31	1. replaced return value with null for TriangleType::triangle → KILLED

Pitclipse Mutation Operators

- https://pitest.org/quickstart/mutators/