Enterprise information systems in practise SW TESTING – part 2

Ing. Daniel Mika, Ph.D. (daniel.mika@atos.net)

- Two hours in the course
- 7 years of praxis in IT (ANF Data, SIS, Atos)
- Area of interest: test and acceptance criteria, quality
- Projects: IMS, WiMAX, ChargingSpot, sLIM
- ISTQB certified tester – foundation level

Content (1)

- Purpose of testing
- Basic test principles
- Test process
- Multilevel testing
- Static techniques
- Blackbox vs. Whitebox testing
- Test management (Test Plan)

Content (2)

- Risk-based testing strategy
- Test exit criteria
- Test-driven development
- Combinatorial testing
- Test automation and regression testing
- Test tools in praxis

Risk-based testing strategy

- What is a risk?
  - Unwanted event that threatens project objectives with negative consequencies
- Three aspects related to risks
  - Impact (loss, cost)
  - Likelihood of occuerence
  - Degree to which its outcome can be influenced
- Categories of risks
  - Project (hard deadlines, external dependencies, skill missing)
  - Process (planning risks, underestimation, bad progress control)
  - Business and product (bad/unstable requirements, bad usability, product complexity, fault proneness, bad quality/stability/perfomance)

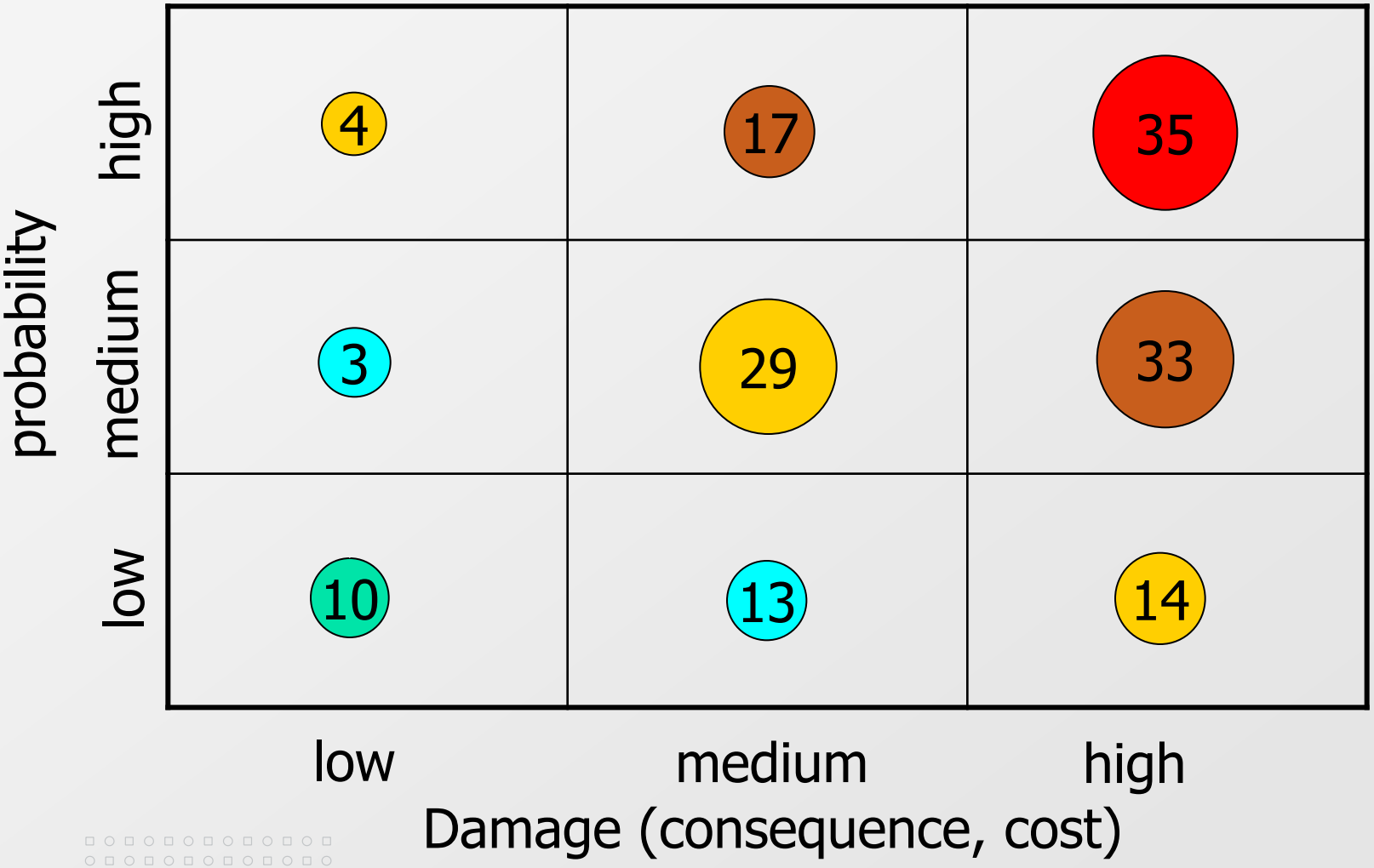
Risk-based testing strategy

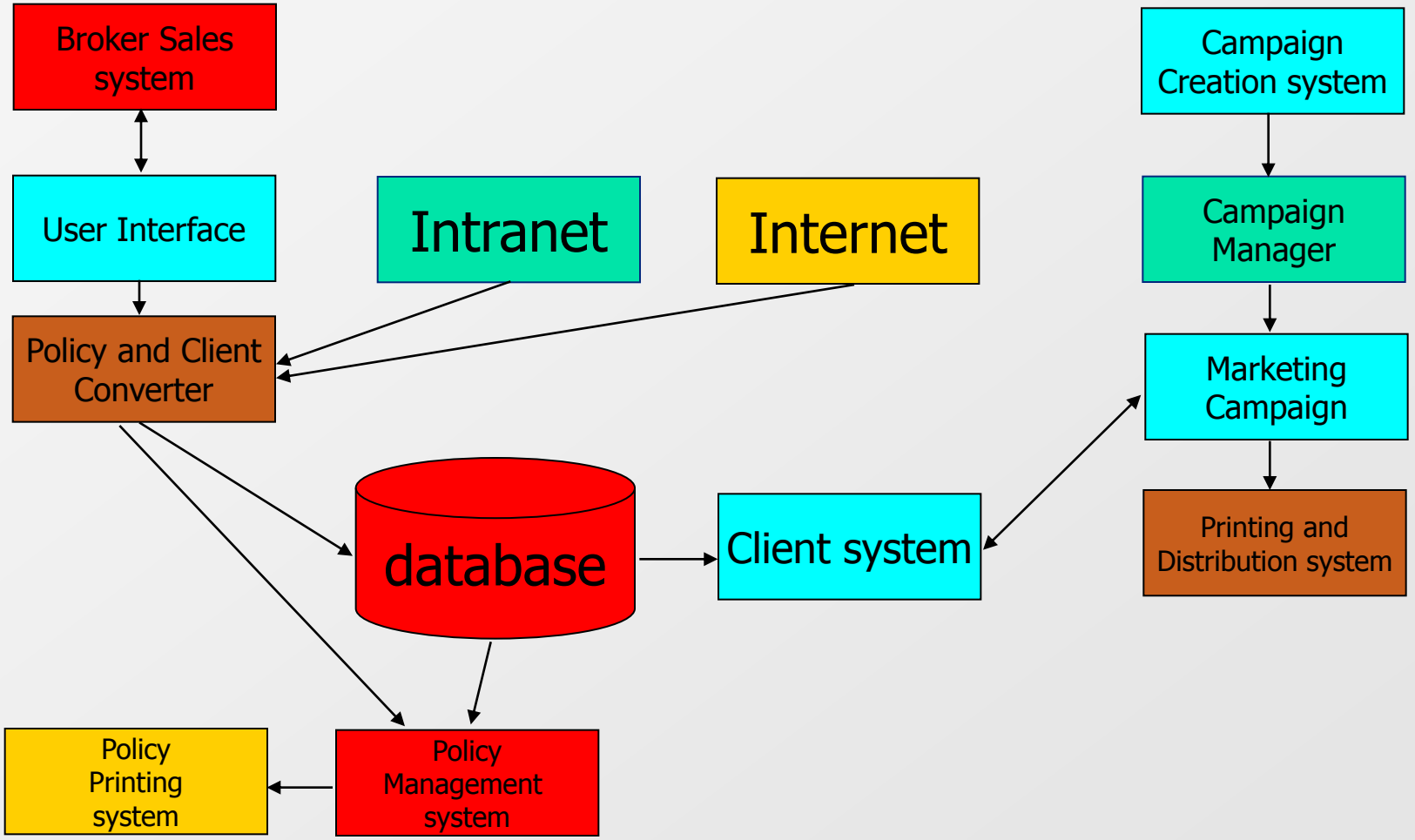
- Base the testing strategy on business goals and priorities => Risk-based testing strategy

- No risk = No test
- Risk = P x D
  - P … probability of failure
  - D … damage (consequnce & cost for business & test & usage)

Risk-based testing strategy

- A testing strategy should answer:
  - What to test?
  - Where to test?
  - Why?
  - When?
  - Who tests?
  - How to test?
  - How much to test?


probability
high
medium
low
4
17
35
3
29
33
10
13
14
low
medium
high
Damage (consequence, cost)


Broker Sales system
User Interface
Intranet
Internet
Policy and Client Converter
database
Client system
Campaign Creation system
Campaign Manager
Marketing Campaign
Printing and Distribution system
Policy Printing system
Policy Management system

Test exit criteria

- Time is over
- Budget is used up
- The boss says “ship it!”
- Testing is never finished, it’s stopped!
- Software products are never released, they escape!

Test exit criteria – unit and integration tests

- All unit and integration tests and results are documented
- There can be no High severity bugs
- There must be 100% statement coverage
- There must be 100% coverage of all programming specifications
- The results of a code walkthrough and they are documented and acceptable

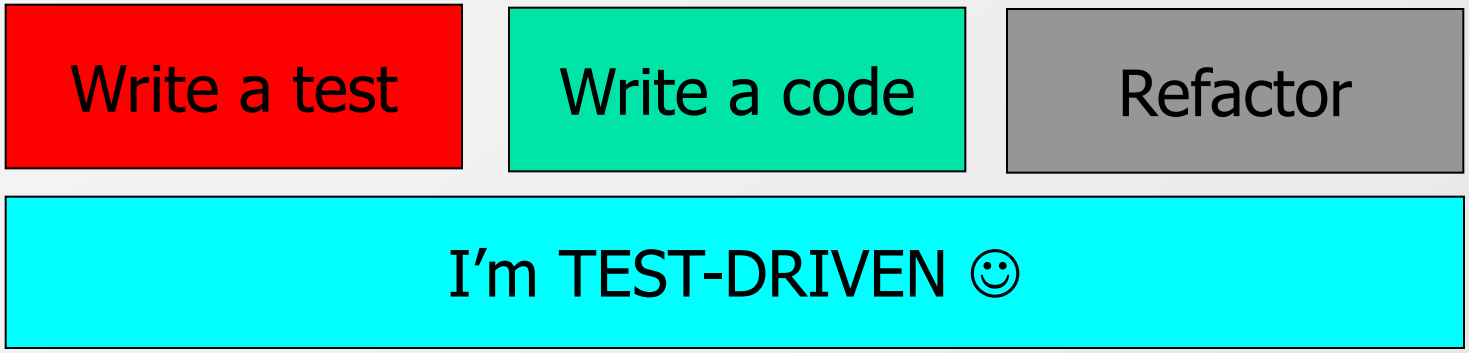
Test exit criteria – system tests

- All test cases must be documented and run
- 90% of all test cases must pass
- All test cases high risk must pass
- All medium and high defects must be fixed
- Code coverage must be at least 90% (including unit and integration tests)

Test exit criteria – acceptance tests

- There can be no medium or high severity bugs
- There can be no more than 2 bugs in any one feature or 50 bugs total
- At least one test case must pass for every requirement
- Test cases 23, 25 and 38-72 must all pass
- 8 out of 10 experienced bank clerks must be able to process a loan document in 1 hour or less using only the on-line help system
- The system must be able to process 1000 loan applications/hour
- The system must be able to provide an average response time of under 1 second per screen shift with up to 100 users on the system
- The users must sign off on the results

Test-Driven Development (TDD)



Efficiency

The fine granularity of test-then-code cycle gives continuous feedback to the developer. With TDD, faults and/or defects are identified very early and quickly as new code is added to the system, and the source of the problem is more easily determined. We contend that the efficiency of fault/defect removal and the corresponding reduction in the debug time compensates for the additional time spent writing and executing test cases. In net, TDD does not have a detrimental effect on the productivity of the software developer.

Test-Driven Development

Test Assets (benefits)

TDD entices programmers to write code that is automatically testable, such as having functions/methods returning a value which can be checked against expected results. Benefits of automated testing, such as TDD testing, include:

- production of a more reliable system
- improvement of the quality of the testing effort
- reduction of the testing effort
- minimization of the schedule

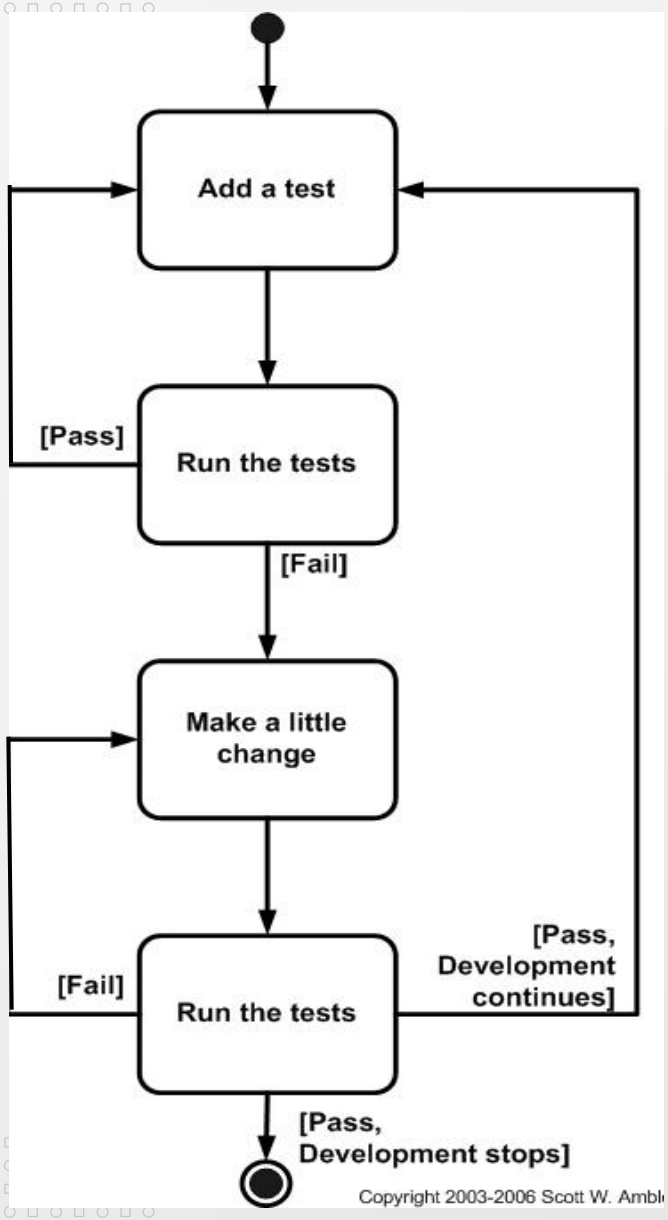
The automated unit test cases written with TDD are valuable assets to the project. Subsequently, when the code is enhanced or maintained, running the automated unit tests may be used for the identification of newly introduced defects, i.e., for regression testing.

Test-Driven Development

Reducing Defect Injection

Debugging and software maintenance is often viewed as a low-cost activity in which working code defect is “patched” to alter its properties, and specifications and designs are neither examined nor updated.

Unfortunately, such fixes and “small” code changes may be nearly 40 times more error prone than new development, and often new faults are injected during the debugging and maintenance. The TDD test cases are a high granularity low-level regression test. By continuously running these automated test cases, one can find out whether a change breaks the existing system. The ease of running the automated test cases after changes are made should allow smooth integration of new functionality into the code base and reduce the likelihood that fixes and maintenance introduce new permanent defects.


Add a test
[Pass]
Run the tests
[Fail]
Make a little change
[Fail]
Run the tests
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambl


Add Test
Watch Test Fail
Write Code
Run Tests
Refactor

TDD - calculator example

x	y	add()	subtract()	multiply()	divide()
0	0	0	0	0	error
1	1	2	0	1	1
4	2	6	2	8	2
9	3	12	6	27	3
35	5	40	30	175	7

```
public class CalculatorFixture extends ColumnFixture {
  public int x;
  public int y;
  public int add() { return 0; }
  public int subtract() { return 0; }
  public int multiply() { return 0; }
  public int divide() { return 0; }
}
```

TDD - calculator example

x	y	add()	subtract()	multiply()	divide()
0	0	0	0	0	expected: error
1	1	expected: 2 actual: 0	0	expected: 1 actual: 0	expected: 1 actual: 0
4	2	expected: 6 actual: 0	expected: 2 actual: 0	expected: 8 actual: 0	expected: 2 actual: 0
9	3	expected: 12 actual: 0	expected: 6 actual: 0	expected: 27 actual: 0	expected: 3 actual: 0
35	5	expected: 40 actual: 0	expected: 30 actual: 0	expected: 175 actual: 0	expected: 7 actual: 0

```
public class CalculatorFixture extends ColumnFixture {
  public int x;
  public int y;
  public int add() { return 0; }
  public int subtract() { return 0; }
  public int multiply() { return 0; }
  public int divide() { return 0; }
}
```

TDD - calculator example

x	y	add()	subtract()	multiply()	divide()
0	0	0	0	0	expected: error
1	1	2	0	expected: 1 actual: 0	expected: 1 actual: 0
4	2	6	2	expected: 8 actual: 0	expected: 2 actual: 0
9	3	12	6	expected: 27 actual: 0	expected: 3 actual: 0
35	5	40	30	expected: 175 actual: 0	expected: 7 actual: 0

```
public class CalculatorFixture extends ColumnFixture {
  public int x;
  public int y;
  public int add() { return x+y; }
  public int subtract() { return x-y; }
  public int multiply() { return 0; }
  public int divide() { return 0; }
}
```

Result of the refactoring

```
public class Calculator {
public int plus(x, y) { return x + y; }
public int minus(x, y) { return x - y; }
public int times(x, y) { return x * y; }
public int divide(x, y) { return x / y; }
}
public class CalculatorFixture extends ColumnFixture {
  public int x;
  public int y;
  private Calculator calc;
  public CalculatorFixture() { calc = new Calculator(); }
  public int add() { return calc.plus(x,y); }
  public int subtract() { return calc.minus(x,y); }
  public int multiply() { return calc.times(x,y); }
  public int divide() { return calc.divide(x,y); }
}
```

Combinatorial testing

- Electronic bookstore (5 parameters with different number of values)
- 1200 = 4 * 3 *5 * 5 * 4 possible parameter combinations exist
- Which parameter combinations shall be selected?

Type of credit card	Credit card number	Expiration date	Product type purchased	Quantity purchased
Amex	Correct	50	Book	1
Discover	Incorrect Length	Invalid year	Video	0
Visa	Invalid digits	Today	Software	-1
Master Card		Yesterday	Book,Software, Video	10
		Invalid Character	Book,Sofware	

Combinatorial testing

- System under test with 4 components, each of which has 3 possible elements
- Overall number of possible configurations: 81 = 3 * 3 * 3 * 3
- Which configurations shall be selected?

Calling phone	Call type	Access	Called phone
Regular	Local	ISDN	Regular
Mobile	Long distance	PBX	Mobile
VOIP	Toll free	Loop	pager

Combinatorial testing

- n independent parameters P_1, P_2, … , P_n
- with m_i different values each with i = 1,2,…,n
- number of possible combinations: $m_1 * m_2 * … * m_n$

- Testing all possible combinations results in an astronomical number of test cases which is infeasible and inefficient

- Wanted: an adequate test case design method to reduce the number of test cases while enhancing coverage and quality of tests

Combinatorial testing

- Best guess
  - Intuition and hope and luck
- Random choice
  - Expert know-how
- All combinations
  - Every combination used in test cases
  - Suitable for trivial cases
- Each choice
  - Each value of each parameter to be included in at least one test case

All Combinations for Three Variables of Three Levels Each			
	A	B	C
1	Red	Red	Red
2	Red	Red	Green
3	Red	Red	Blue
4	Red	Green	Red
5	Red	Green	Green
6	Red	Green	Blue
7	Red	Blue	Red
8	Red	Blue	Green
9	Red	Blue	Blue
10	Blue	Red	Red
11	Blue	Red	Green
12	Blue	Red	Blue
13	Blue	Green	Red
14	Blue	Green	Green
15	Blue	Green	Blue
16	Blue	Blue	Red
17	Blue	Blue	Green
18	Blue	Blue	Blue
19	Green	Red	Red
20	Green	Red	Green
21	Green	Red	Blue
22	Green	Green	Red
23	Green	Green	Green
24	Green	Green	Blue
25	Green	Blue	Red
26	Green	Blue	Green
27	Green	Blue	Blue

All-Pairs Array, Three Variables of Three Levels Each			
	A	B	C
2	Red	Red	Green
4	Red	Green	Red
9	Red	Blue	Blue
12	Blue	Red	Blue
14	Blue	Green	Green
16	Blue	Blue	Red
19	Green	Red	Red
24	Green	Green	Blue
26	Green	Blue	Green

Test Automation

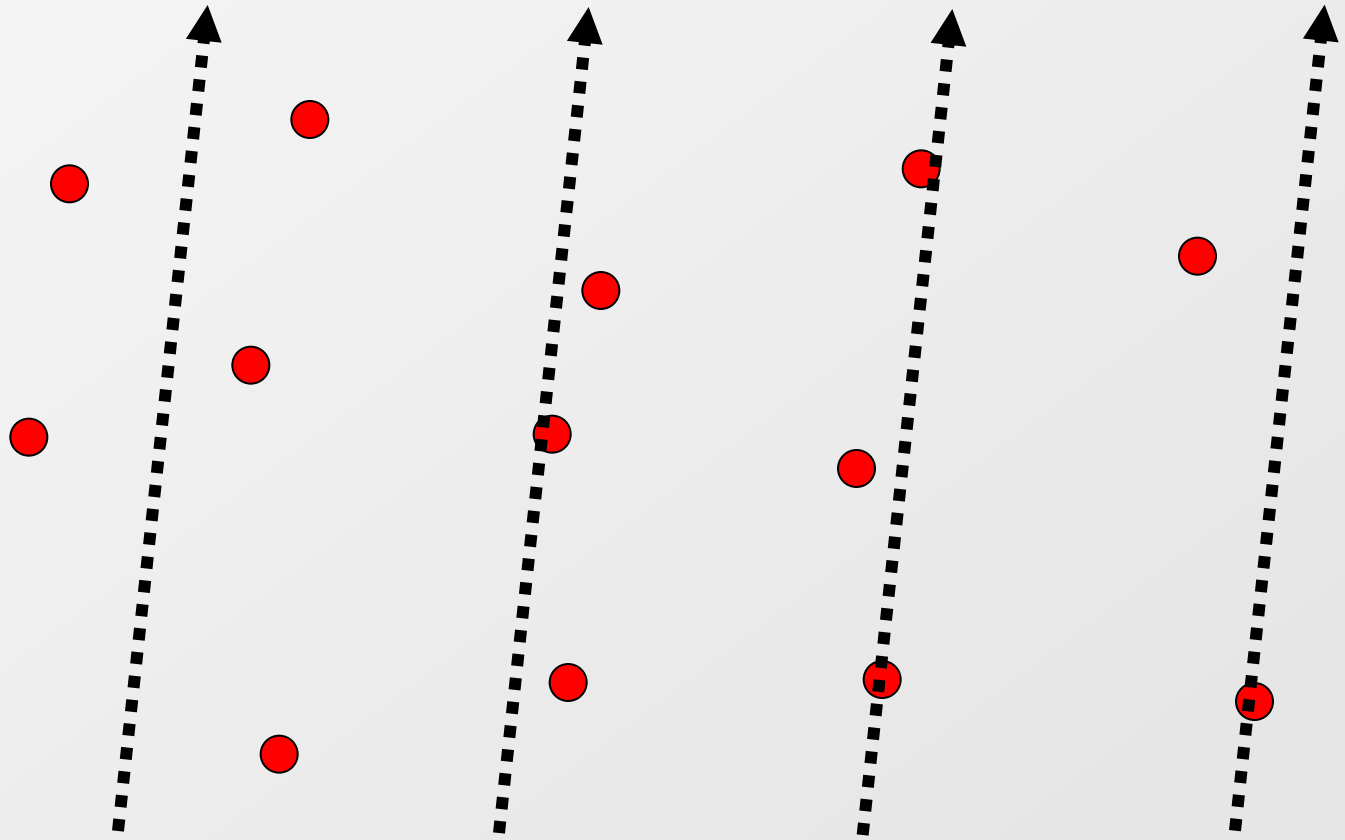
- Introducing test automation is sometimes like a romance: stormy, emotional, resulting in either a spectacular flop or a spectacular success.

  Bogdan Bereza-Jarocinski, 2000

Test Automation - Why

- Automated testing is a foundation for any kind of iterative or agile development
- Daily builds and small releases are useless if they cannot be validated
- Find more regression bugs
- Run the most important, useful, valuable tests more often (continuously, overnight, on weekends)
- Reduce testing stuff
- Reduce elapsed time for all tool-supported testing activities (setup, execute, analyze, …)
- Control cost of automation effort vs. effort saved by automation
- Testing and Automation -> different objects !

Test Automation – Limitations (Minefield metaphor)



Regression Testing

- The fundamental problem with software maintenance is that fixing a defect has a substantial (20-50 %) chance of introducing another.

  Frederick P. Brooks, Jr., 1995

- When you fix one bug, you introduce several newer bugs.
- ISTQB Glossary (2007)
  - Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

Regression Testing – test selection strategy

- Retest all
- Retest by risk – priority, severity, criticality
- Retest by profile (frequency of usage)
- Retest changed parts
- Retest parts that are influenced by changes

A real life problem – AT&T Phone System Crash, 1990

- What happened
  - Mal-function in central server led to chain reaction
  - Service outage of half of the system for 9 hours
  - Loss of 75 million dollars damage for AT&T
- Reasons
  - Wrong usage of break command
  - Software update directly in largest part of the system

```
switch expression {
  ...
  case (value):
      if (logical) {
          statement;
          break;
      } else {
          statement;
      }
      statement;
    ...
}
```