DEBUGGING: TESTING

WS 2017/2018

Martina Seidl
Institute for Formal Models and Verification

JOHANNES KEPLER
UNIVERSITY LINZ

Testing is a Huge Field ...

Acceptance testing

Accessibility testing

Acceptance testing

Installation testing

Non-functional testing

Sanity testing

Unit testing

Continuous testing

Regression testing

Smoke testing

Beta testing

Compatibility testing

Component interface testing

Integration testing

Alpha testing

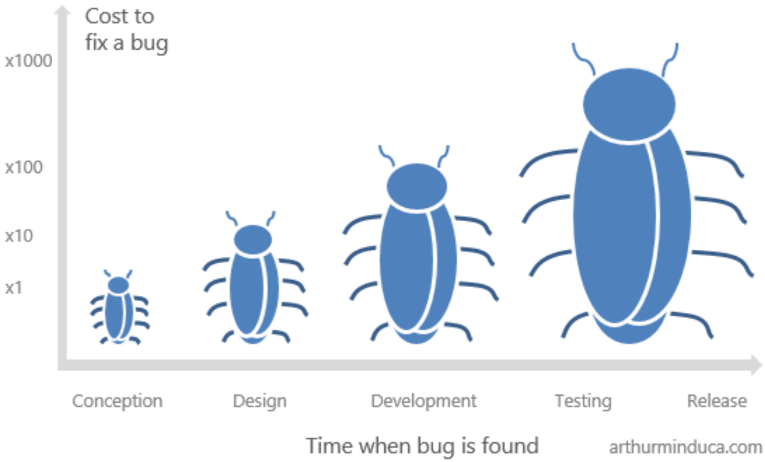
Performance testing

Usability testing

Destructive testing

Functional testing

Costs of Defective Software



Testing

Testing is the execution of a program
with the intent to make it fail.

two views on testing:

- **testing for validation**
  detection of yet unknown failures
- **testing for debugging**
  uncovering a known problem

Tests in Debugging

in debugging, tests help to

- reproduce the problem
- simplify the problem
- observe a specific run
- ensure that a fix was successful
- protect against regression (a similar problem will not occur again)

$\Rightarrow$ **set program up that it can be tested**

Automation of Testing

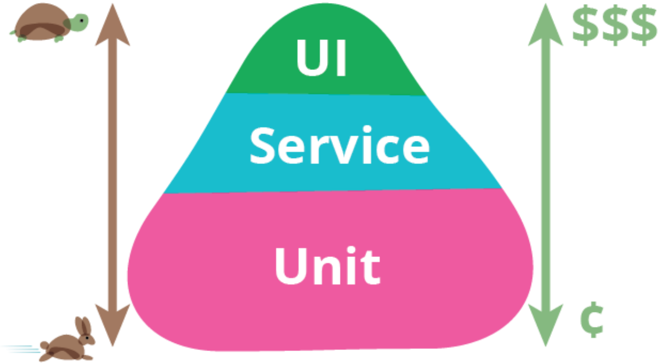
some tests are difficult to perform manually
$\Rightarrow$ automate testing!

benefits of test automation

- more reuse of tests
- increased repeatability
- simplification of test cases
- isolation of
  - failure-inducing input
  - failure-inducing code changes
  - failure-inducing thread schedules
  - failure-inducing program state

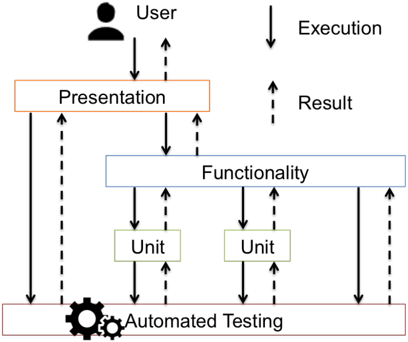
$\Rightarrow$ **increase trust in program**

Test Pyramid



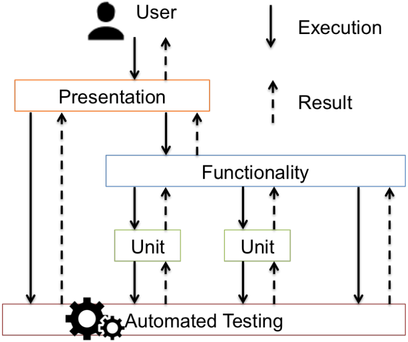
from `https://martinfowler.com/bliki/TestPyramid.html`

Interaction Layers of a Program



adapted from [Zeller09]

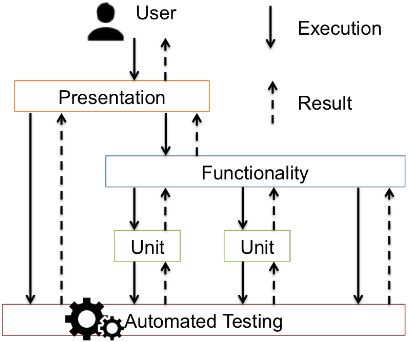
Interaction Layers of a Program



adapted from [Zeller09]

presentation layer: interaction with the user/environment

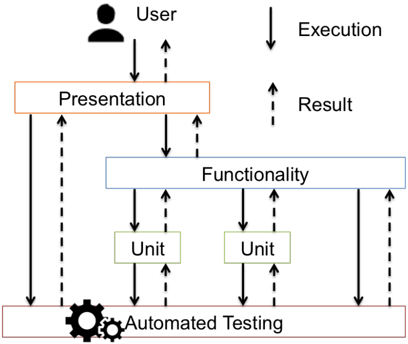
Interaction Layers of a Program



adapted from [Zeller09]

functionality layer: encapsulate the functionality (independent from a specific presentation)

Interaction Layers of a Program



adapted from [Zeller09]

unit layer: splitting of functionality across cooperating units

Challenges in Automated Testing

automated testing can be performed on all layers
(with different benefits and drawbacks)

- **ease of execution**: How easy is it to get control over program execution?
- **ease of interaction**: How easy is it to interact with the program?
- **ease of result assessment**: How can we check results against expectations?
- **lifetime of test case**: How robust is my test when it comes to program changes?

Testing at the Presentation Layer (1/2)

benefits:

- simulate and automate user behavior

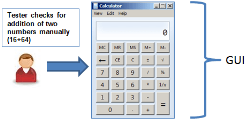
challenges:

- synchronization
- abstraction
- portability
- assessment of output

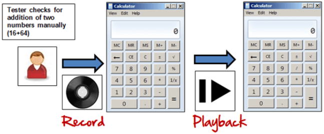
> rule of thumb: the friendlier an interface is to humans, the less friendly it is to computers

Testing at the Presentation Layer (2/2)

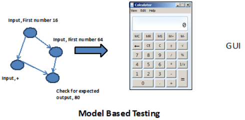
- manual testing



- record and replay

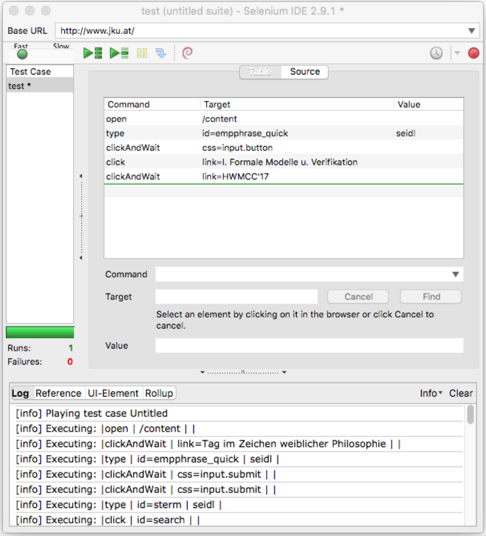


- model-based testing



pictures from `https://www.guru99.com/gui-testing.html`

Example: Record-And-Replay (Selenium)



Testing at the Functionality Layer

benefits:

- direct access of the program's functionality
- automation support by computing infrastructure
- programatic access and evaluation of results
- less fragile than testing at the presentation layer

requirement:

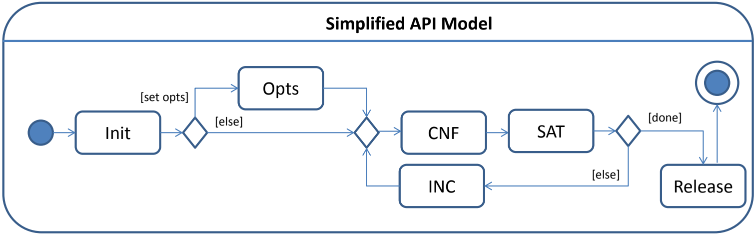
clear separation between presentation and functionality

Model-Based Testing

- **model**:
  - finite state machine
  - specification of intended behavior
  - representation of test strategies and testing environment
- **execution**:
  - generic framework (e.g., Modbat)
  - specific framework (e.g., lglmbt)
- **different kinds of models**, for example:
  - API model
  - option model
  - data model

⇒ very powerful in combination with fuzz testing

Example: Model of a SAT Solver



from `http://fmv.jku.at/papers/ArthoBiereSeidl-TAP13.pdf`

Testing at the Unit Layer

idea:

- decomposition of program into units (subprograms, functions, libraries, classes, ...)
- automation of the execution of a specific unit
- test the behavior of the individual units

tasks of a unit testing framework:

1. set up environment for embedding the unit
2. execute the unit's testcases and verify the outcome
3. tear down the environment

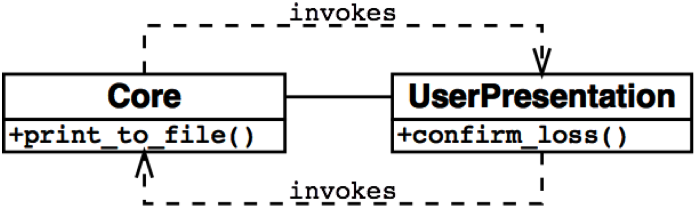
Isolating Units

requirements:

- clear separation between presentation and functionality
- availability of results

problem: (circular) dependencies

example [Zeller09]:



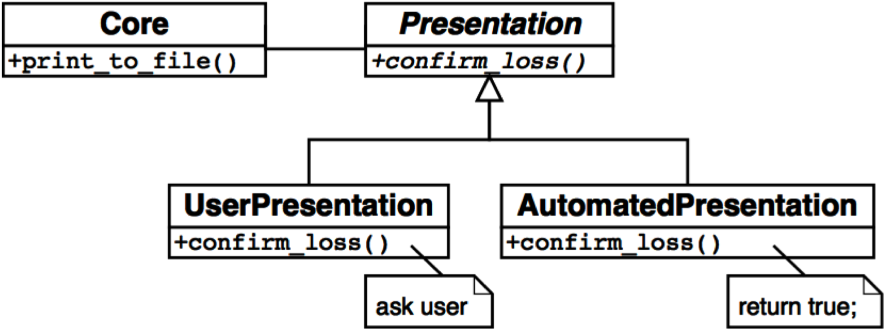
Isolating Units Example: Problem

```
void print_to_file(string filename) {

    if (path_exists(filename)) {

        // FILENAME exists; ask user to confirm overwrite

        bool confirmed = confirm_loss(filename);
        if (!confirmed)
            return;
    }
    // Proceed printing to FILENAME...
}
```

Isolating Units Example: Fix (1/2)

```
void print_to_file(string filename,
                   Presentation presentation) {
    if (path_exists(filename)) {

        // FILENAME exists;
        // ask user to confirm overwrite
        bool confirmed =
            presentation.confirm_loss(filename);
        if (!confirmed)
            return;
    }
    // Proceed printing to FILENAME
    ...
}
```

Isolating Units Example: Fix (2/2)



Breaking Dependencies

dependency inversion principle: depend on abstraction rather than details

to break the dependency from class A to class B

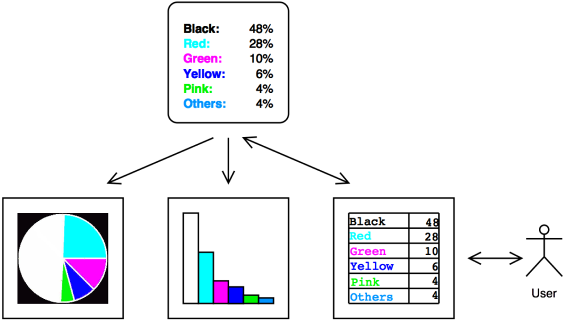
1. introduce an abstract superclass B' of B
2. change A such that it depends on B' (rather than B)
3. introduce new subclasses of B' that can be used with A

$\Rightarrow$ new subclasses of B' can be used without changing A

Design for Debugging

- decompose the system such that dependencies between components are minimized
- one way of realization: **model-view-controller pattern**

example: information system for elections

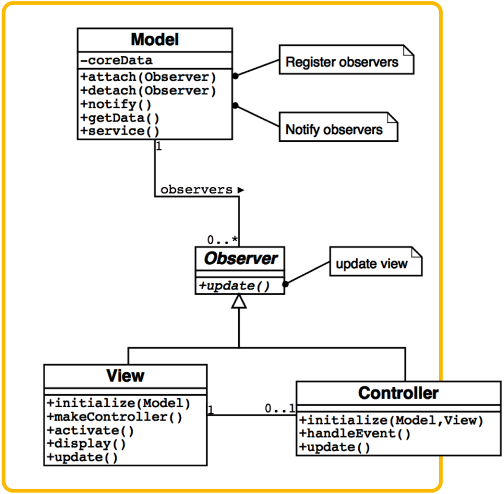


MVC Pattern

- **model**:
  managing the data
- **view**:
  displaying the data
- **controller**:
  processing the data

benefits for testing

- controllers for automated execution
- dedicated views
- independent testing of M and C

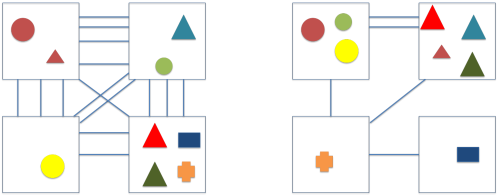


Design Rules

reduction of dependencies by

- **high cohesion**: Those units that operate on common data should be grouped together.
- **low coupling**: Units that do not share common data should exchange as little information as possible.

low cohesion – high coupling vs high cohesion – low coupling



⇒ use features of programming languages

Rules for Quality Assurance

specify	test early	test first
test often	test enough	have others test
check	verify	assert

Reproducing the Problem

- first step in debugging: reproduce the problem
- necessary for
  - observing
  - fixing
- generate a test case that triggers the failure if problem was reported by user

challenges: reproducing the

- problem symptoms
- environment (problematic setting)
- history (necessary steps to create the problem)

Reproducing the Environment

debugging in the problem environment is often not possible because of

- **privacy**: users and companies don't want other on their computers
- **ease of development**: development environment (incl. diagnostic software is not available on the user's machine
- **cost of maintenance**: users cannot stop working while their machines are used for debugging
- **travel costs**
- **risk of experiments**

$\Rightarrow$ diagnostic actions use local environment

Diagnosis in Local Environments

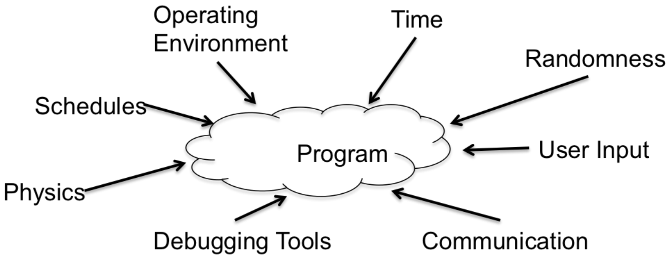
iterative process for reproducing the problem in the local environment

1. attempt to reproduce THE problem
   (as described in the problem report)
2. adopt properties (e.g., config files, drivers, hardware)
   prefer properties that are
   - most likely responsible
   - easy to change/undo
3. stop adopting properties if
   - the problem is reproduced
   - the local and the problem environments are identical
     - incomplete or wrong problem report?
     - overseen difference

side-effect: learn about failure-inducing circumstances

Reproducing Program Execution

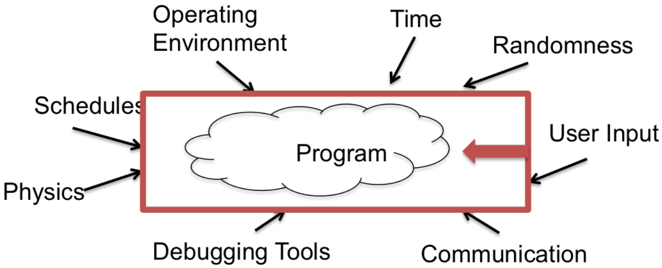
- generation of individual steps that resulted in failure
- **challenge**: reproduce the program input by
  - observing the program input
  - controlling the program input
- **types of input**:



Controlling Inputs

introduction of control layer between real input and input perceived by program

$\Rightarrow$ isolation of program under observation from environment



Reproducing Data

- data comes from files, databases, etc.
  - documents
  - configuration files
- under control of user
- usually easy to transfer and replicate
- **challenges**:
  - get ALL the data that is necessary
  - get ONLY the data that is necessary
  - privacy (sensitive information)
    - sign non-disclosure agreement
    - anonymize data
    - simplify data such that sensitive information is removed

Reproducing User Interaction/Communication

- input comes from complex user interfaces or via networks
- often difficult to observe and control
- **possible approach**:
  - capture interaction: record input
  - replay interaction: execute program with previously recorded input
- similar as testing on the presentation layer
- additional challenge in reproducing communication:
  - huge amount of input
    $\Rightarrow$ bad impact on performance
  - solution: start from last correctly reproducible state

Reproducing Time/Randomness

- indeterministic input: time/date, random number
- reproducibility for pseudo-random input:
  - make time/random input configurable
  - save time/date
  - save random seed
- real random input:
  - capture sources
  - replay input sequence

Reproducing the Environment (1/2)

- interaction between programs and environment is typically handled via the operating system
  $\Rightarrow$ monitor and control of input and output
  $\Rightarrow$ recording and replaying OS interaction thus makes entire program run reproducible

example: monitoring tools `strace` and `dtrace`

- diverting operating system calls to wrapper functions
- log incoming and outgoing data by diverting a specific interrupt routine that transfers control from program to system kernel
  $\Rightarrow$ no re-linking is necessary

Demo: strace

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 68 vars */]) = 0
brk(NULL)                               = 0x149a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=115809, ...}) = 0
mmap(NULL, 115809, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd2de63e000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260Z\0\0\0\0\0\0"..., 8
fstat(3, {st_mode=S_IFREG|0644, st_size=130224, ...}) = 0
mmap(NULL, 2234080, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7
mprotect(0x7fd2de235000, 2093056, PROT_NONE) = 0
mmap(0x7fd2de434000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENY
mmap(0x7fd2de436000, 5856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANON
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

Reproducing the Environment (2/2)

tracing

- a huge amount of data
- replay everything to reproduce failure $\Rightarrow$ huge performance penalty

alternative: checkpoints

- records entire state that it can be restored
- ideally, record stable state (e.g., between two transactions)
- replay interaction since checkpoint
- problem: states are usually huge

Reproducing Schedules

- many concurrent threads and processes on modern computing systems ⇒ operating system defines *schedule* in which individual parts are executed
- ideally, program behavior is independent of schedule
  - schedule is indeterministic
  - program behavior is deterministic
- non-deterministic programs are very challenging to debug
- example:

Thread A	Thread B
open(file)	
read(...)	
modify(...)	
write(...)	
close(...)	
	open(file)
	read(...)
	modify(...)
	write(...)
	close(...)

Thread A	Thread B
open(file)	
	open(file)
	read(...)
read(...)	
modify(...)	
write(...)	
close(...)	
	modify(...)
	write(...)
	close(...)

updates get lost!

Reproducing Schedules

if the problem has been found: fix the problem with synchronization mechanisms, otherwise:

- solution 1: record the schedule
  $\Rightarrow$ enable deterministic replay
  - huge amount of data
  - performance
  - scalability
- solution 2: uncover differences in execution
  - massive random testing
  - program analysis

Physical Influences

ways to influence a computing device:

- energy impulses
- quantum effects
- real bugs
- humidity
- mechanical failures
- ...

rare and hard to reproduce

Effects of Debugging Tools

debugging tools might change the behavior of a program

- differences between debugging environment and production environment:
  - uninitialized memory
  - corrupted memory
  - insertion of output statements
  - different compiling options
- results:
  - problem is masked by another problem
  - problem is gone
- counter-measures:
  - checking the data flow
  - assertions

Reproducible and Less-Reproducible Problems

- **Bohrbug**:
  - repeatable
  - manifests reliably under a possibly unknown but well-defined set of conditions
- **Heisenbug**:
  - disappears or changes when one tries to isolate it
- **Mandelbug**:
  - appears chaotic/non-deterministic
- **Schroedinbug**: manifests only if someone
  - reads the source code
  - uses the program in an uncommon manner

Focusing on Units

- reproduce the execution of a specific unit
  (might be easier than controlling the whole program)
- example: problem with database
  $\Rightarrow$ execute only SQL statements instead of whole application
- approach:
  1. introduce logging for recording the behavior
  2. set up mock object that simulate the recorded behavior

Reproducing a Crash

case of a crash recording is efficient and effective

different approaches:

- keep a copy of the calling stack
  expensive, because of permanent monitoring
- remember the failing state
  less expensive, but different information
- wait for a second chance
  activate monitoring after crash