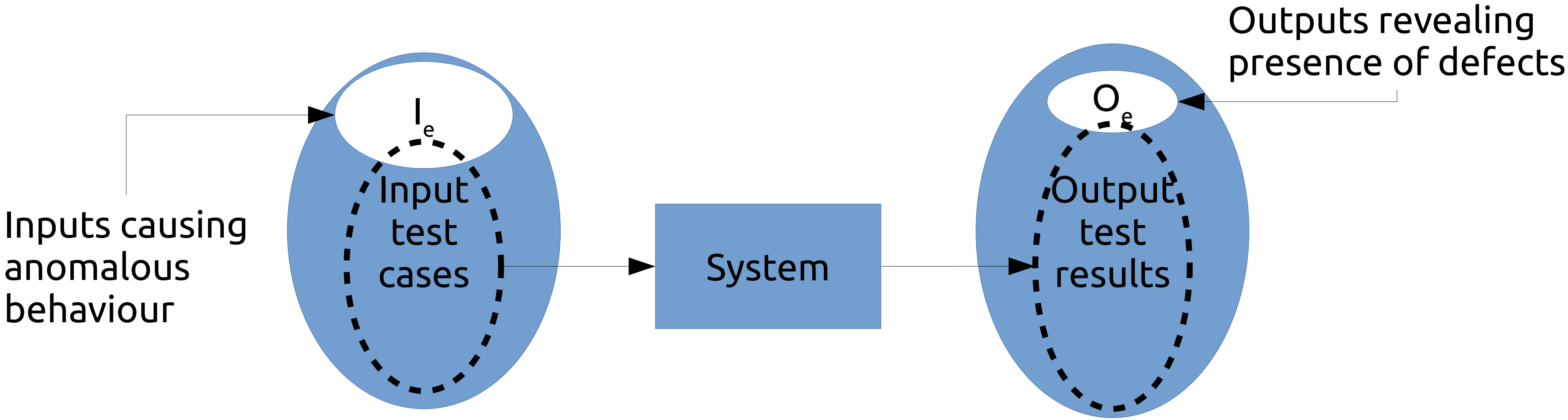
Software Engineering

Lecture 11 – Testing & Continuous Integration

© 2015-19 Dr. Florian Echtler
Bauhaus-Universität Weimar
<florian.echtler@uni-weimar.de>

Testing (Recap)

- Abstract: process test cases, check results
- However: tests can only show *presence* of errors, not *absence*.


Inputs causing anomalous behaviour
I_e
Input test cases
System
Output test results
O_e
Outputs revealing presence of defects

Testing (Recap 2)

- Validation testing
  - Show that software meets requirements
  - Test cases modelled after typical use cases
- Defect testing
  - Obvious goal: find bugs/errors/design flaws!
  - Test cases contain atypical/erroneous data

Testing: Variants

- Testing is possible at many levels/stages
- Development testing
  - unit testing
  - component testing
  - system testing
- Performance testing
- User testing
- Release testing

Development testing strategies (1)

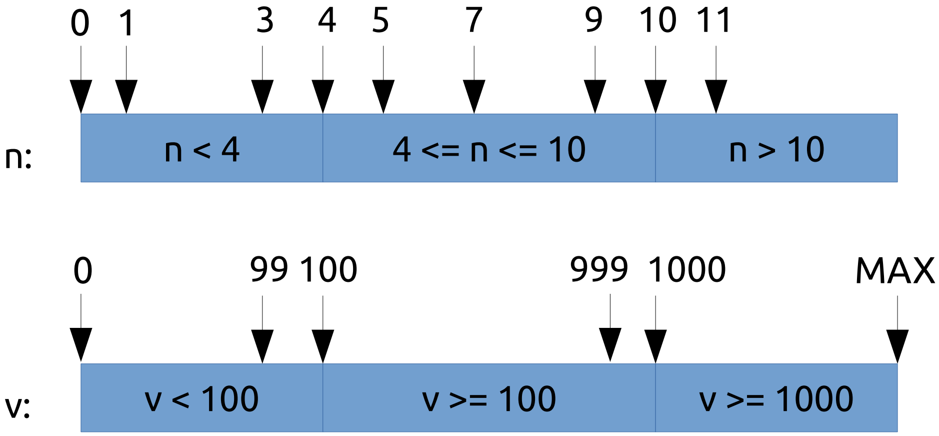
- Partition testing (*defect* testing)
  - Determine *equivalence partitions* for input data
  - Equivalent behaviour for all inputs from one partition
  - Select test cases from each partition and at partition boundaries
  - Related to path testing/code coverage (equivalent behaviour → same execution path), see lecture 8
  - Usually requires some knowledge about internals, i.e. pure black-box testing difficult

Development testing strategies (2)

- Guideline-based testing (*defect* testing)
- Select test cases known to be error-prone
  - NULL for pointers
  - NaN, -0, inf for float/double values
  - INT_MAX, -INT_MAX, 0 for integers
- Sequences/arrays/vectors
  - Sequences with 0 or 1 values
  - Sequences with different lengths for each test
  - Access first/middle/last element of sequence

Development testing strategies (3)

- Example:
  - Function which accepts 4-10 input values
  - Each value is 3-digit integer >= 100
- Partitions → see diagram
- Guidelines:
  - Also test with empty sequence/ single value
  - Test with input values of 0/INT_MAX



0 1 3 4 5 7 9 10 11

n:

n < 4

4 <= n <= 10

n > 10

0 99 100 999 1000 MAX

v:

v < 100

v >= 100

v >= 1000

Performance/stress testing

- Mainly relevant for back-end systems (servers, databases) – hybrid *verification/defect* test
- Usually relies on required performance (e.g. transactions/second) and exceeds this limit
- Goal: test failure behaviour
  - soft fail: just fewer transactions than requested
  - hard fail: system crash/data loss

Fuzzing

- Intentionally flood the component with random/garbage input
  - More data per time than during normal operation
  - Malicious/garbage data values
- Also possible for UIs, e.g. *monkeyrunner* on Android (generates random touch events)
- Often used for security testing, i.e. to find exploitable bugs

User testing

- Tests performed by end-users, not developers
- Focus on user interface, not internals
  - Paper prototypes (before any code is written), mockups (e.g. using HTML5/Flash)
  - “Classic” usability study, think-aloud testing (invite testers to lab, observe usage)
  - “In-the-wild” study → daily usage scenario + recording/logging of comments, interactions, …
  - A/B testing: provide two different variants of UI to two groups of people, compare e.g. efficiency

Release testing

- Final *verification* tests before delivery
- Usually black-box testing, relying only on specification/requirements
- Also called acceptance testing, may involve customers/users
- In agile processes (no rigid requirements):
  - Part of each cycle (e.g. Scrum)
  - Performed by “product owner”
  - Some documentation/”sign-off”recommended

Continuous Integration (1)

Source: https://en.wikipedia.org/wiki/Continuous_integration

CI: agile method, collection of "best practices"

- Maintain a code repository
  - Use branches sparingly
- Automate the build
  - A single command (e.g. "make") should build everything
- Make the build self-testing
  - Tests should be integrated into build process

Continuous Integration (2)

Source: https://en.wikipedia.org/wiki/Continuous_integration

Often considered the most central part of CI:

- Everyone commits to mainline every day
  - Keeps number of conflicts low
- Every commit to mainline should be built
  - Should also be automated, e.g. with Jenkins, Travis-CI (integrated with Github), …

Continuous Integration (3)

Source: https://en.wikipedia.org/wiki/Continuous_integration

- Keep the build fast
  - Prerequisite for frequent re-builds
- Test in a clone of the production environment
  - e.g. test apps on real phone, not simulator
  - Separate test env. can introduce new bugs
  - Use scaled-down production environment
- Make it easy to get the latest deliverables
  - e.g. direct download access for customer

Continuous Integration (4)

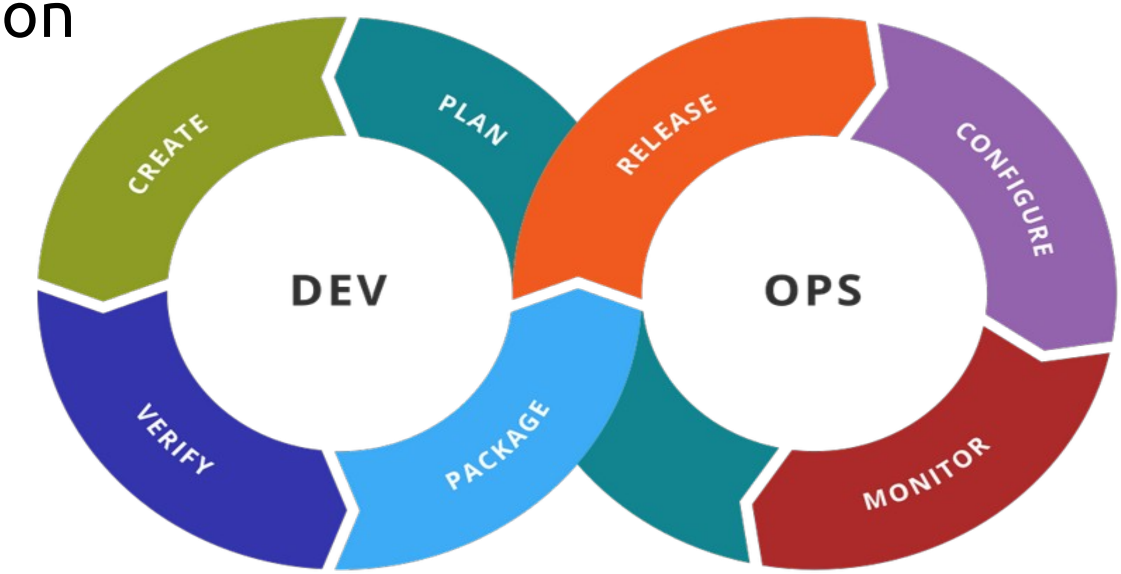
Image source (CC): https://en.wikipedia.org/wiki/Build_light_indicator

- Everyone can see results of latest build
  - Build problems are fixed quickly
  - Often shown by physical indicators (see image)
- Automate deployment
  - e.g. automated upload to app store/beta testers
  - “Continuous Deployment”

DevOps

Image source (CC): https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg

- “Development” + “Operations”
- DEV side: very similar to, e.g., Scrum
- OPS side: stronger focus on software maintenance
- Heavy reliance on automation tools
- Useful integration of expertise, or just a way to reduce personnel?



Build systems

- Compile & link
  - See lecture 9
- Dependency resolution
  - Internal: determine dependencies of objects, modules, source code etc. (often via timestamps)
  - External: locate/install missing libraries, tools, headers etc.

Build systems (2)

- Test management
  - Run test suites after (each?) successful compilation
  - Provide overview of succeeded/failed tests, test coverage
- Install products – e.g. …
  - Copy to suitable filesystem locations
  - Create archives/packages
  - Upload to app store

(Meta-)build systems: examples

- Make
- Autotools/CMake
- Ant/Maven/Gradle
- Eclipse/Xcode/Visual Studio

Make

- Ancient in computing terms – created 1976
- Somewhat obscure syntax ("Makefile")
- Only deals with internal dependencies
- Can be extended using external tools/scripts

Autotools

- Makefile generator
- Widely used in open-source projects
- Only available for Unix-like environments
- Consists of multiple sub-tools (automake, autoconf, configure) which create a Makefile
- Also deals with external dependencies
- Very powerful, but also very obtuse

CMake

- More modern replacement for autotools
- Also generates Makefile *or* Visual Studio XML
- Cross-platform (Windows, Linux, MacOS)
- Mostly a standalone scripting language

Ant/Maven/Gradle

- Standalone build systems
- Focused on Java projects
- XML-based (Ant/Maven) or JSON-based (Gradle) project description files
- Cross-platform (Windows, Linux, MacOS)
- Often used for Android projects (esp. Gradle)

Eclipse/Xcode/Visual Studio

- Integrated Development Environments (IDEs)
- Build system, editor, RCS frontend, test manager, UML tools, …
- Support multiple languages (usually at least Java/C++)
- Typical examples of CASE tools (Computer Aided Software Engineering)

Build systems: Summary

- Once again: one size does not fit all
- Build system can add lots of complexity
- Try to avoid “feature creep”
- Most open-source projects focus on CMake (C, C++) or Ant/Maven (Java)

Questions/Comments?

- Thanks for listening!

I'D LIKE TO THANK ALL OF THE PEOPLE WHO HELPED DESIGN THE TECHNOLOGY TEST PARAMETERS.
DilbertCartoonist@gmail.com
THANKS TO YOUR INPUT, THE TEST HAD NOTHING IN COMMON WITH HOW THINGS WORK IN THE REAL WORLD.
SO I WASTED TWO WEEKS OF MY LIFE ON A TEST THAT IS NOT ONLY MEANINGLESS. . .
© 2010 Scott Adams, Inc./Dist. by UFS, Inc.
. . . BUT ALSO DANGEROUSLY MISLEADING.
THIS SLIDE SHOWS THE GAP BETWEEN THE TEST RESULTS AND REALITY.
WE'LL USE THE TEST RESULTS ANYWAY BECAUSE IT'S THE ONLY DATA WE HAVE.
FINE. I HOPE YOU ALL CHOKE TO DEATH ON YOUR LUNCHES.
www.dilbert.com
11-7-10
WHY'S HE SO CRANKY?
SOME-THING ABOUT DATA.