

csc444h: software engineering I

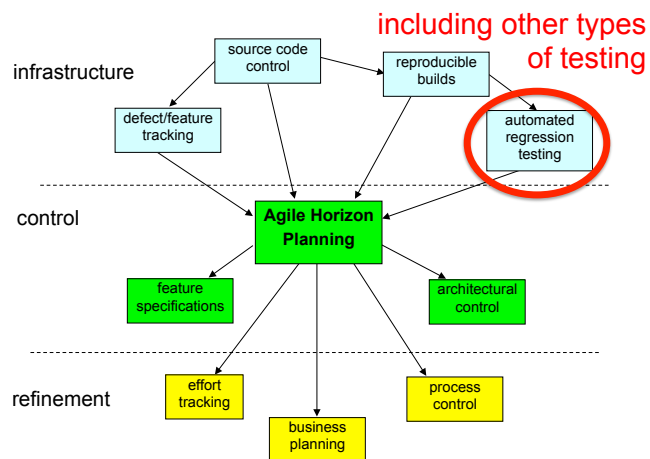
matt medland

matt@cs.utoronto.ca

<http://www.cs.utoronto.ca/~matt/csc444>

testing

top-10



should have read ch 1-12 now (ch 10 today, skip ch 6)

software quality assurance

- humans are fallible
 - infeasible to completely fix the humans
 - need to double and triple check their work to find the problems
- testing
 - running the software to see if it works the way it is supposed to.
 - works according to specifications
 - ensures specifications are reasonable (that they solve the intended problem)

software quality assurance (2)

- reviews
 - inspecting written work products looking for errors
 - requirements, specifications, designs, and code
- proofs
 - proving that the software behaves according to a written, formal specification
 - important in control systems and other critical software amenable to proof
 - can useful for general-purpose software as well

proving programs correct

- should think of programs logically, not operationally.
- understand the program as a *predicate transformer*
- *predicate*:
 - a logical expression that characterizes the state of the system
- *pre {P} post*
 - the program transforms the pre predicate into the post predicate.
 - each line of the program should be thought if in those terms
 - each line transforms the pre condition closer and closer to the post condition

example

precondition: array has ≥ 5 elements

code

post-condition: # of elements printed $= 5$

proven!

no “off-by-one” errors here
this kind of thinking becomes second nature when programming
a very, very powerful tool

induction

- proving by induction is also a useful technique
- ex, prove that:
$$factorial(n) = n!$$
- for all natural numbers n
 - start with base case, usually $n = 0$ or $n = 1$
 - prove by induction that if it’s true for n then it must be true for $(n+1)$

unit test

- testing performed by the coder as they are coding.
- will test in their dev debug build
- will want to build “test scaffolding” to test the code they have written independent of the final application.
 - can use pre-build unit testing frameworks such as xUnit (Kent Beck – Extreme Programming)
 - JUnit, CUnit, CPPUnit, PyUnit Test::Unit, VbUnit, ...
 - best practices is to not just test and discard, but consistently maintain the automated unit tests and have them execute after every nightly build.
 - try to break dependence on any other modules, use “mockups” and DI (dependency injection) instead.
 - catches problems very early, right at the source.
 - confident in changing a module
 - living “documentation” of how to use a module
 - strengthens interface v.s. implementation

component (or function) test

- started when a feature is relatively complete and stable.
- occurs during coding phase (pre-dcut).
- performed by a tester, not by the coder.
- uses a nightly dev release build.
- tester will:
 - try out those parts of the feature that the coder says are supposed to work
 - communicate issues back to the coder in an informal fashion
 - i.e., not counted as “defects” yet
 - re-test as coder works out issues
 - develop a test plan for the feature
 - a document describing how the feature will be tested
 - develop automated tests for the feature

integration test

- after dcut.
- all features of all executables have been coded
- testers begin executing their test plans
- test that the features work together as expected
- problems are recorded as formal “defects”.

system test

- as the system stabilizes.
- tests of full production installs
- tests on how this application works with other related applications

final release test

- last minute checklist before a release goes out the door
 - not rushed!

regression testing

- tests made to ensure that functionality that once worked continues to work.
- test made to ensure that previously discovered and corrected defects do not re-appear
 - a fertile source of defects
- can be performed manually
 - but would take too long

automated regression testing

- an extension of the nightly builds
- software scripts will execute a set series of tests and report the results back into a database
- QA will examine the results each morning
 - 4 reasons for a failure:
 - the function was broken
 - the function was changed
 - the function was improved
 - the test is faulty
- the function of the test team is to ensure good coverage on automated regression tests
 - each new function should get a suite of regression tests
 - should be formalized in the feature creation process
 - each defect should get a test that would have caught it
 - should be formalized in the defect resolution process

performance regressions

- easy to build test cases and forget to measure the time it takes to execute them
- systematically
 - collecting this information,
 - consolidating it,
 - and reporting on itwill show up performance trends
- required because sometimes coders will check-in a change that looks to be functionally ok, but has very negative performance implications
 - e.g., if coder only tested on a few simple test cases and did not notice because the run-time was swamped by the overhead

memory leak regressions

- run a special version of the software, instrumented to find memory leaks, bad memory allocation errors, and bad pointer chasing
 - e.g., Purify from IBM/Rational/Pure
- runs slowly, but can use a representative sample of the nightly regression tests.
- less required when running *managed code*
 - *C# .NET*
 - *Java*

benefits of regression testing

- **locks-in quality**
 - once you achieve quality, you don't backslide
 - everybody focuses on new features and forgets the old
- **finding defects sooner**
 - finds the defect nearest the point in time it was injected
 - freshest in the coder's mind
 - least expensive time to fix it
- **development aid**
 - can work on complex, central bits of the code without fear of breaking something major on not finding out
- **releasing**
 - if need a last minute critical defect fix to release
 - if no/poor automated regression, might have to delay until re-tested

regression coverage

- to manage a program to institute or improve automated regression testing, you require a coverage metric.
- what % of the application is tested.
 - can count functions from the outside
 - coverage of all functions
 - # of tests per function
 - can count lines of code traversed
 - excellent coverage metric
 - will not necessarily get all combinations
- other measures of coverage...

testing the GUI

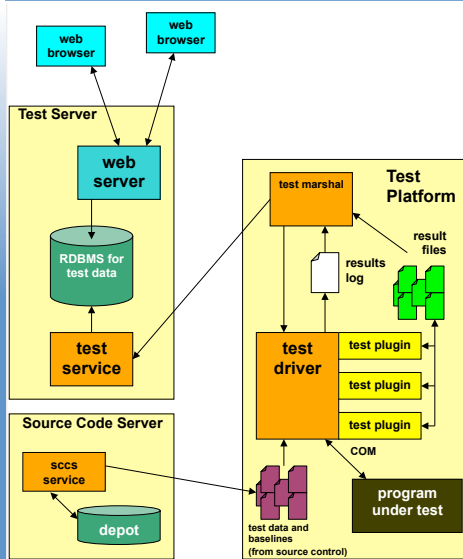
two general approaches to testing GUI-based apps:

- 1) use a GUI test tool (ex. selenium)
 - pumps UI events at the app
 - extracts results from text widgets, bitmaps, files
 - problems:
 - very sensitive to changes in the GUI
 - very sensitive to changes in GUI sequencing
 - many false positives
 - costly to maintain
 - easy to drive the app, hard to see if results are correct
 - hard to get at the results
 - throw it all away if make a big gui change

testing the GUI (2)

- 2) architect to test at a layer just beneath the GUI
 - create an a.r.t. API
 - might use an embedded interpreter
 - Perl, Python, VBScript
 - might hit the app from outside
 - COM
 - C/C++ API
 - problems:
 - not really testing the GUI, testing something a bit different
 - coders need to develop and maintain APIs

example automated regression architecture



- execute nightly, and from dev/ test desktops
- cross-platform
- plug-ins for new types of tests
- extreme fault tolerance
 - constantly monitoring itself
 - re-start if hangs or dies
 - try last test again
 - if fails then go on
- log all actions
 - maintain history prior to a crash
- records results to an RDBMS
- records timings as well
- reports accessible via web browsers
- all test cases and baselines in source control

SaaS automated regression

- trick with SaaS is so much code is now javascript running in various browsers (not all of which behave the same way)
- open source frameworks to the rescue:
 - **Selenium**
for recording and executing “in browser” tests (also has Selenium Hub for parallelizing tests) can output tests in a scripting language for storage.
 - **Bromine**
for storing tests, organizing them, scheduling them, recording results, and reporting on results
- commercial services
 - **SauceLabs**
for running Selenium instances in the cloud (pay per “test-minute”) – Bromine integrates with SauceLabs