Lecture Notes
November 17, 2020

"Assignment T3: First Iteration" due tonight! https://courseworks2.columbia.edu/courses/104335/assignments/490808 Only one team member needs to submit

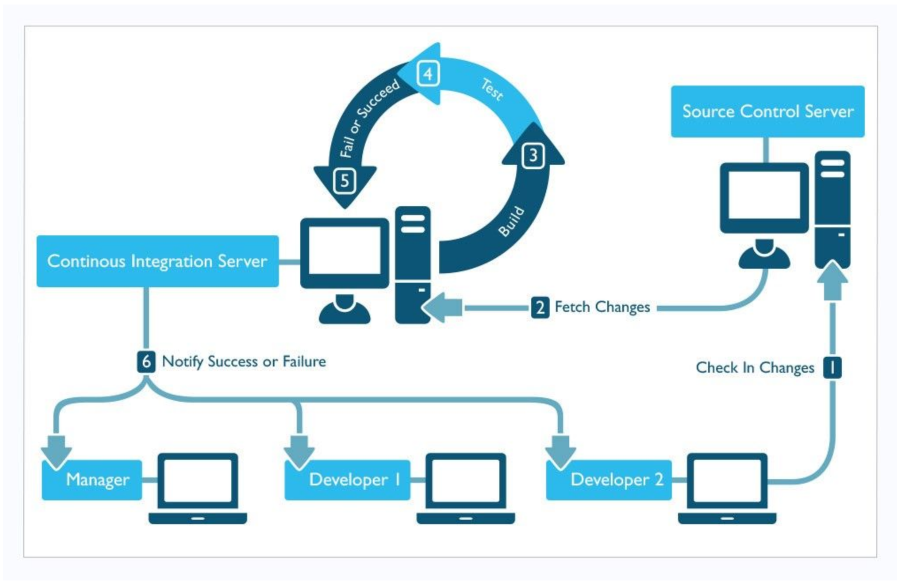
"Assignment T4: Initial Demo" due Monday. https://courseworks2.columbia.edu/courses/104335/assignments/491014 Only one team member needs to submit

For T3, the same rule applies as for the individual project: You can turn in T3 up to two days late with no points penalty, then 20% off each day after that. So up to two days 100%, 3 days 80%, 4 days 60%, 5 days 40%, 6 days 20%, 7 days 0% but we still grade it in the sense of giving comments, 8 days nothing. Courseworks is set up to not accept submissions after 7 days. The days are *not* prorated, one minute more counts as the whole next day.

However, T4 is set up in courseworks to not accept submissions after *two* days beyond the due date. It's ok to demo before submitting T3. Schedule asap with your IA. You can also volunteer to demo in class on Thursday (extra credit!).

Working as a team: Don’t break the build!

Continuous integration (CI) tools like Jenkins and Travis CI hook a version control system (like github) with a build tool to automatically re-run the build (on a build server, not a developer machine) after every commit to the shared repository, so errors can be detected quickly.



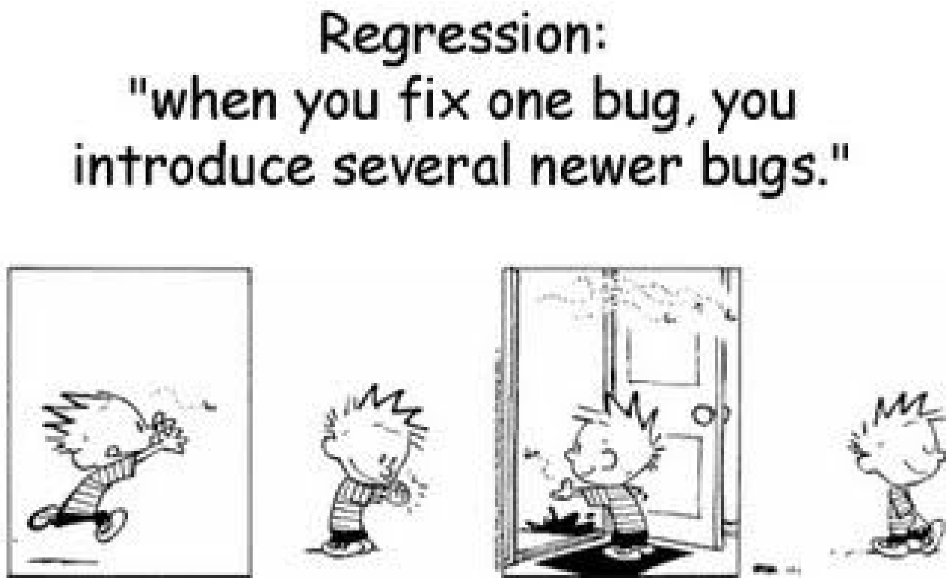
In the CI context, “build” usually refers to testing, not just literally build (compile and link), and applies to non-compiled languages. *Which* test cases actually run on commit may vary. In some organizations CI runs selected tests on each commit and certain other tests on a timer (e.g., every night).

A common choice for commit testing is to use a “test minimization” strategy that selects only new test cases (committed at the same time) plus existing test cases whose results (pass or fail) could have been affected by the committed code changes (test impact analysis).

This may be a relatively small subset of all the tests, but there’s a cost to accurately computing the affected tests (without actually running any tests). For example, track all code c exercised by each test t during *previous* testing. If the commit changes c, or changes any other code that c depends on directly or indirectly (determined by static analysis), then re-run t. It may be less expensive to run full regression testing than to run full static analysis.

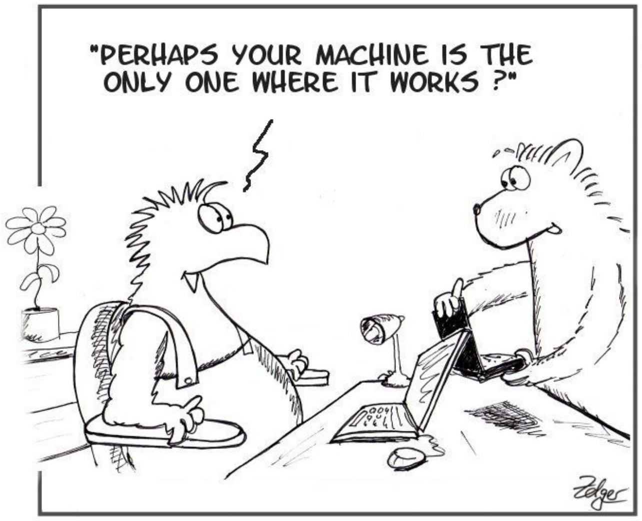
Regression testing is the typical choice for nightly builds.

A *regression* is when fixing a bug or adding a new feature causes previously working code to break, or “regress”, sometimes code that seems completely unrelated in another part of the codebase (because there was a hidden dependency).



To detect all regressions, you need to re-run *all* test cases that previously passed - and even tests that previously failed, in case they now fail in different ways or (unexpectedly) pass.

Of course, if the code change fixed a bug, you would expect the corresponding test case(s) that detected the bug to now pass. The developers should test this before commiting. But other test cases that do not seem relevant may also now pass. Duh what? (Hidden dependencies.)



It works on my machine

"You broke the build" = Errors detected during continuous integration were traced to *your* commit. If all developers had fully tested their changes locally (on their own machine) prior to commit, and fixed any errors prior to commit, in theory CI should not find any errors.

Let's watch Don't break that build!

You used the Maven build tool for the individual project and all teams are using either a build tool or a package manager (or both) for their team projects.

The earliest widely-known build tool was make, developed to compile and link C and other 1970s Unix languages. (The original version of make is described here.) Modern versions of make are in common use today

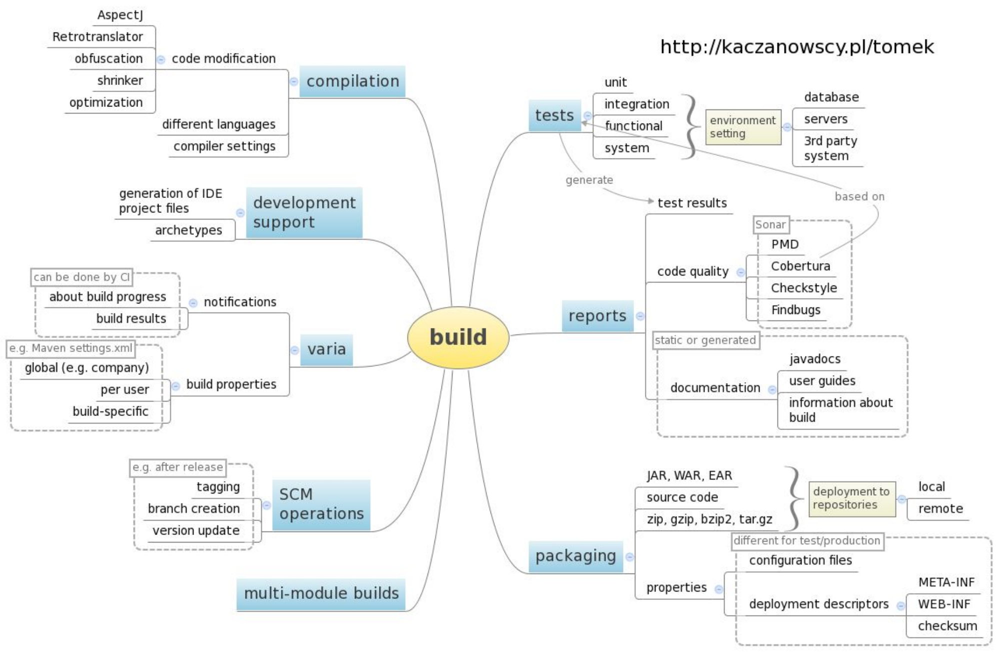
Today’s build tools, including make, automate much more than compile/link and often call many other tools - particularly testing tools.

Build tools provide some notation for writing down all the steps to be automated. Some but not all package managers provide an analogous notation.

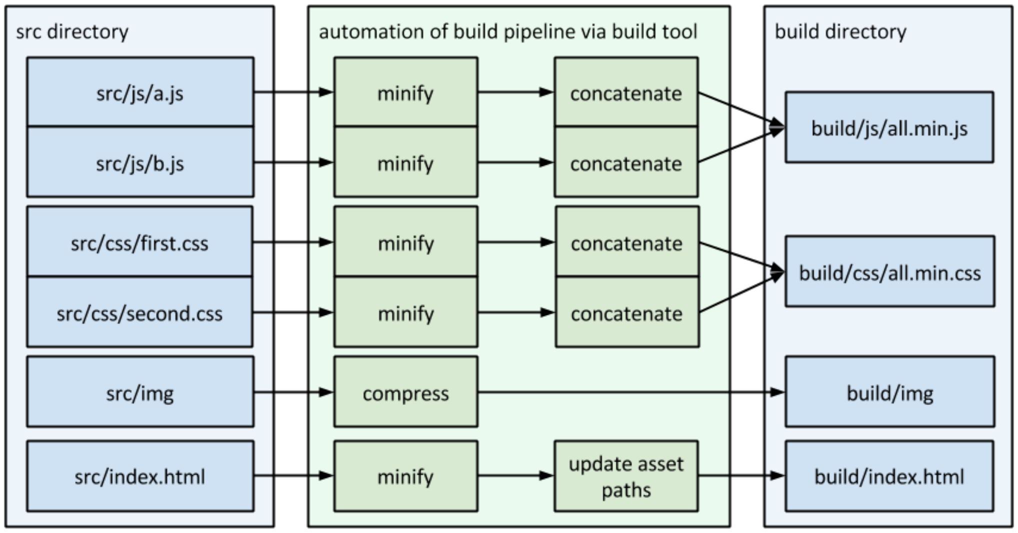
```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Given this makefile, what does “make” do?

What does “make clean” do? Why would you want this?



A makefile prescribes make’s “build configuration”. A *build configuration* is a repeatable, reproducible, standard sequence of steps so developers do not have to remember the steps and everyone in the team uses exactly the same steps



A build configuration defines compiling and linking, and/or packaging, in terms of dependencies. Non-trivial software has *dependencies* on other software, and even trivial software usually has dependencies on standard libraries.

When you organize your own codebase into multiple components - classes, modules, packages, libraries, etc. - then some of your components are likely to be *dependent* on some of your other components

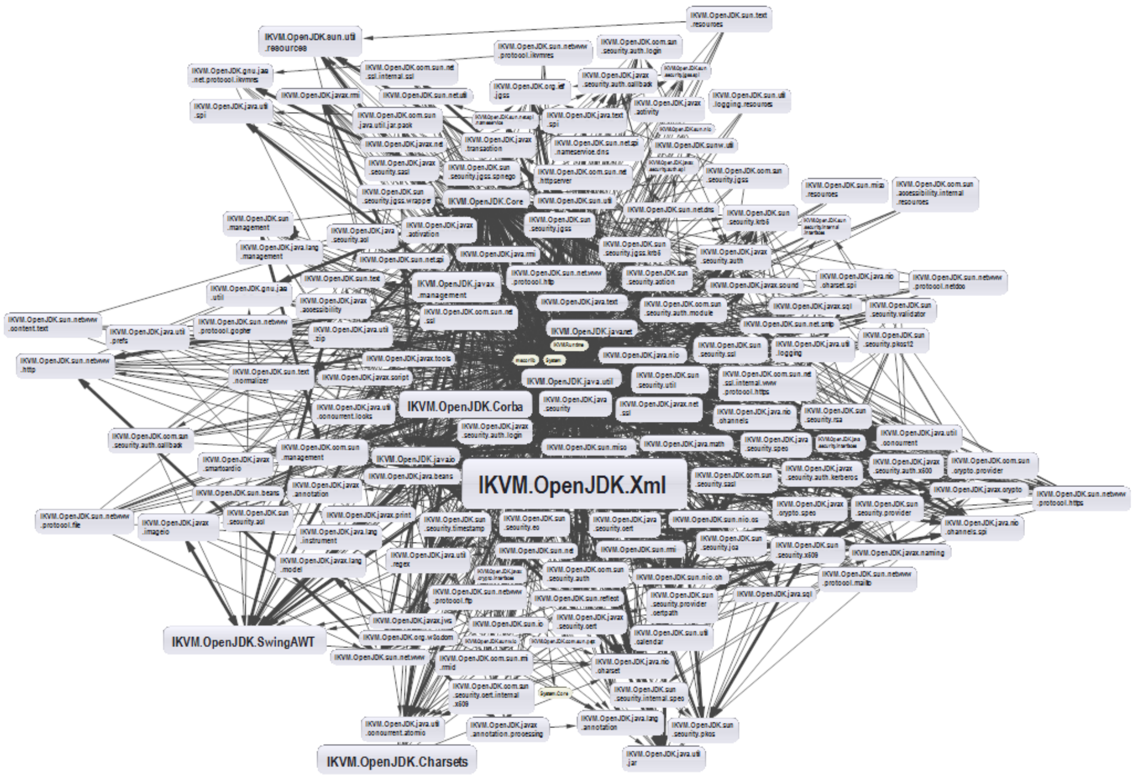
When you use frameworks, libraries, APIs, and other code and/or data resources, your code *depends* on those resources

For example, if you write an application that requires authentication and authorization (e.g., login userid and password, maybe 2-factor), instead of implementing this capability yourself, you should use a widely-used package written by experts. Then your program depends on that package.

Polymorphism and interfaces complicate dependency analysis, since you need to account statically for all possibilities that could occur at runtime.

Dependencies are *transitive*, A depends on B depends on C depends on D.

Highly interconnected dependencies and/or reliance on specific versions of the external components (“DLL hell” on Windows) is **bad**. Why?



In order for your software to run, the compiler or runtime environment needs to know what all the dependencies are and where to find them

Package managers (e.g., maven central, pip, npm) keep track of the third-party components that your codebase uses, beyond those that “come with” the language platform

They utilize the “metadata” associated with each package - the software's name, description of its purpose, version number, vendor, checksum, and a list of its own dependencies

Many of them work like a mobile app store:

- Download third-party libraries and keep up to date
- Based on ecosystem with central repository
- Ensures placed in right spot on your file system
- Can also remove and clean up
- Eliminates the need for manual installs and updates
- Ensures integrity and authenticity by verifying digital certificates and checksums

The term *dependency* is also used in software engineering to refer to the *coupling* code smells - the degree to which your component depends on the “innards” of another component, as opposed to its interface, so that if the other code changes internally you need to change your code.

Dependency may also mean a relationship among work tasks on an agile task board, such that you cannot do one task (e.g., running a test) until another task has been completed (e.g., writing the test). Ideally, dependencies among tasks by different software developers should be minimized, so they can work in parallel.

There are also control flow and data flow *dependencies* between program elements (e.g., statements, expressions, variables) within a single code unit.

We need to be clear what we mean by “dependency”!

Although application code necessarily has dependencies on other application code and (in most cases) on third-party code, and test cases are necessarily dependent on the application code they test, test cases are supposed to be *independent* from each other.

Dependencies among test cases are the main cause of “flaky tests”. A [flaky test]() is a test case that produces different results (pass or fail) on different test suite executions *even when no application code has changed.*

```
Assert.AreEqual(
    GetTimeOfDay(),
    "Morning" );
```

Some testing frameworks allow developers to specify test case ordering, some don’t.

Some testing frameworks run test cases in unpredictable order.

set up global state then run tests in sequence do lots of tests; test1 () { do something that changes global state; }; do lots of other tests; test2 () { do something that uses global state; }
set up global state then run tests in sequence do lots of tests; test2 () { do something that uses global state; } do lots of other tests; test1 () { do something that changes global state; }

What is the problem?

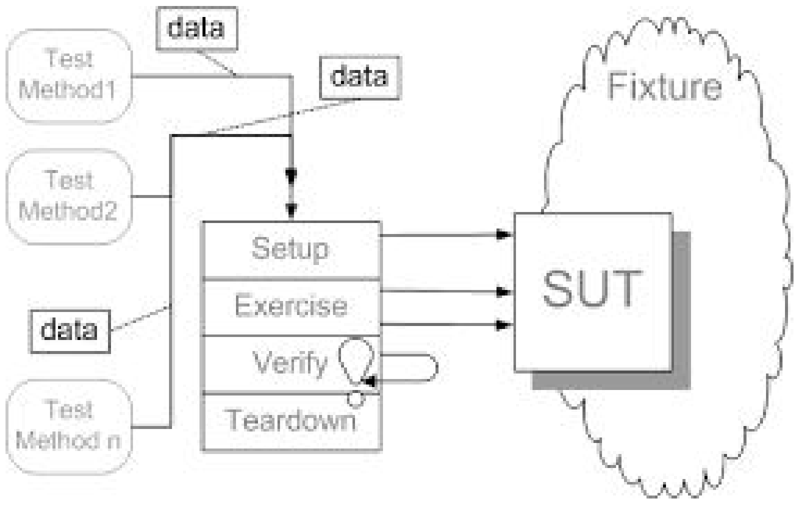
Some testing frameworks run test cases in parallel to speed up test suite execution.

set up global state then run tests in parallel;	
do lots of tests; test1 () { do something that changes global state; }	do lots of other tests; test2 () { do something that uses global state; }

What is the problem?

Any tests *intended* to share application state and run in a specific order should be included in the same test class - which testing frameworks treat as an atomic unit.

Setup/teardown fixtures associated with test classes construct and clean up the shared state.



But setup and teardown may be incomplete (and may have their own bugs).

Finding all hidden dependencies among test classes is extremely expensive - for n test classes, in worst case O(2^n). Why?

Rather than try to find all hidden dependencies, some testing frameworks offer an option to *prevent* any hidden dependencies from affecting test results by restarting/rebooting the environment in between test classes - just in case.

This is extremely expensive: It might take 3-5ms to run each JUnit test, but ~1.4s to restart JVM between each pair of tests. It takes much longer to reboot Linux. But, technically, only O(n) cost for n tests…

Is there a cheaper way to make sure there are no dependencies between test classes?

“Assignment T3: First Iteration” due tonight!
https://courseworks2.columbia.edu/courses/104335/assignments/490808 Only one team member needs to submit

“Assignment T4: Initial Demo” due Monday.
https://courseworks2.columbia.edu/courses/104335/assignments/491014 Only one team member needs to submit

For T3, the same rule applies as for the individual project: You can turn in T3 up to two days late with no points penalty, then 20% off each day after that. So up to two days 100%, 3 days 80%, 4 days 60%, 5 days 40%, 6 days 20%, 7 days 0% but we still grade it in the sense of giving comments, 8 days nothing. Courseworks is set up to not accept submissions after 7 days. The days are *not* prorated, one minute more counts as the whole next day.

However, T4 is set up in courseworks to not accept submissions after *two* days beyond the due date. It’s ok to demo before submitting T3. Schedule asap with your IA.

Or your team can volunteer to demo your application *in class* on Thursday, to the whole class using zoom screen sharing, for extra credit. If you are interested, contact me asap on piazza (post to ‘instructors’ not just me).