Stepping Up Your Programming Projects: Transitioning to Software Development

Eric Ung (ungxx011@umn.edu)
CSCI 3081: Program Design and Development
ungxx011

Contents

1) So you want to make bigger projects?

[Why should you read this paper?]

After taking some introductory courses in programming, you should have done some projects for class and maybe on your own time. This paper is aimed towards you, the reader, who has some experience programming and wants to take your programs into a higher level. I will cover three essential tools that you can use immediately for your own projects.

[Three tools you are going to learn and use effectively!]

The three tools covered in this paper are Apache Subversion, make files, and iterative development. They are the essential tools for any project and there are other tools covered during classes and in your future, but I will keep it simple yet practical for your interest.

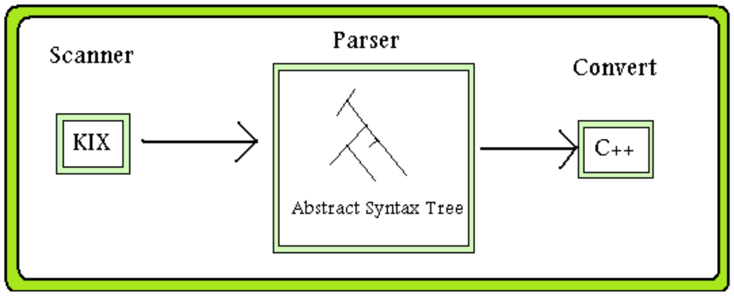
[What are Makefiles, Subversion is repository, Iterative style of development?]

Makefiles	**A file for a project that compiles as many targets (eg. programs) that the user wants in a single command line on terminal based consoles.**
Subversion	A version control system that solves the problem of managing sets of files and tracking changes between groups of people.
Iterative Style Development	A common and professional method of programming where planning and making tests are emphasized before writing code.

2) Background of my project to elaborate a point

[My Project]

Throughout the paper I will refer to a project I am currently doing and I feel it is best that you should be aware of the higher level details of it. The project I did to illustrate that these tools are useful involves converting a programming language, called KIX, into C++ code. This may seem simple but involves a lot of complicated ideas that can't fit in my head at once. In order to tackle this project, the project was broken into three iterations: scanner, parser, and convert. The scanner checks each word and makes sure each of them is in the KIX language. After, the parser creates some sort complicated structure, called an abstract syntax tree, to add meaning in the programs written in KIX. If the parser fails to make a tree, then that program failed to meet the grammar of a KIX program. After, we create code in C++ and make sure it compiles. This part takes the structure created by parser and converts it into code.



3) Essential Subversion <–> Repository

3.0) What is subversion?

[Subversion is one of many open source version control software to build software.]

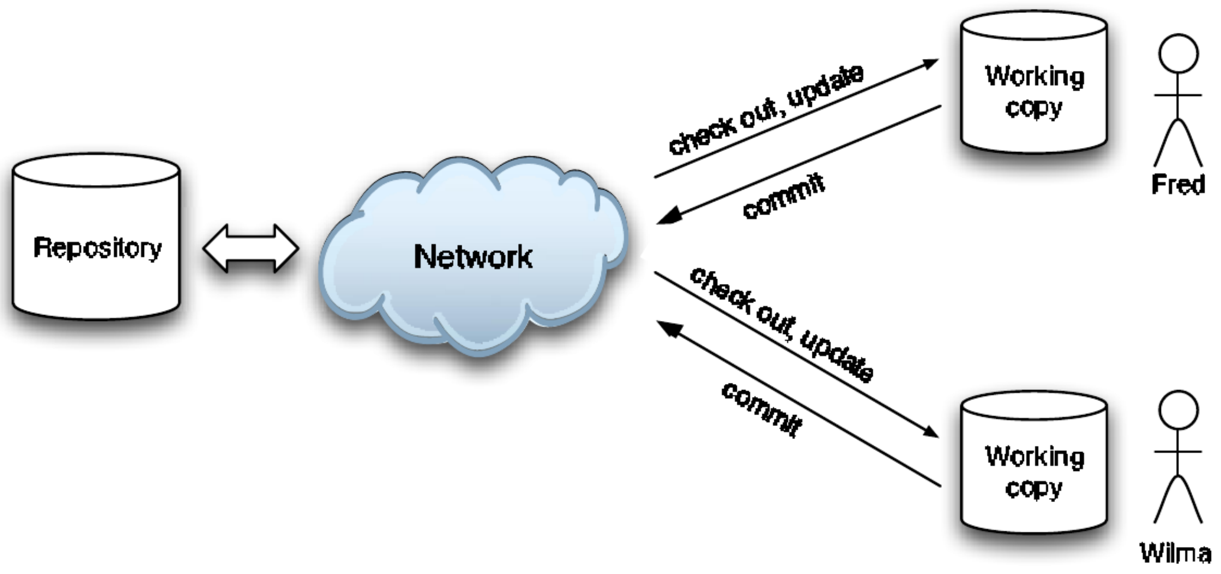
Subversion is a free version control system that stores and manages data between a main repository and individual workspaces where workspaces refer to the users' local set of data.

[Repository? What is that?]

A repository is storage location for all the data (e.g. files, folders, and programs) that keeps track of changes for the project.

[Users can get separate individual copies of data to modify]

Instead of sharing a computer, users can get their own copy/copies of the project. For the project, I worked with another person and we couldn't meet up all the time, thus having separate workspaces helps us work separately and effectively in our natural environments. Also, I don't like sharing my computer and I like to work without someone staring at me most of the time so this is a big deal keeping me sane.

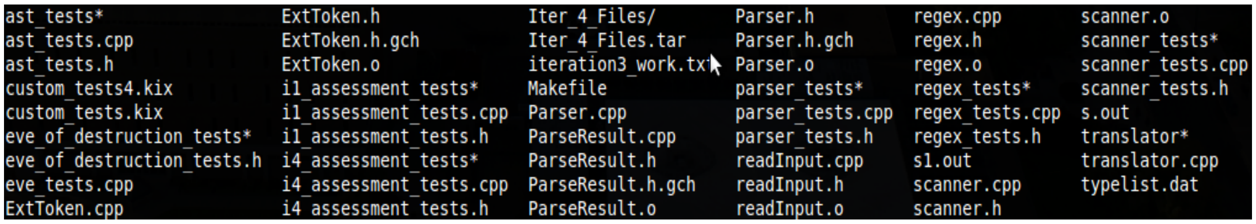


Taken from figure 2.1 of the Pragmatic Book.

3.1) Sharing files has never been easier!

Subversion manages all the folders and files the project contains so sharing files between two or more people is easy. Furthermore, there is password protection to access the server so that malicious users can't corrupt the data. In accessing the website, I was immediately asked for a password to view the files. This not only saved countless hours hand organizing the project but made it an obsolete problem. Near the end of the project, I realized that I never had to worry about which files were important even after a month of working on the project.

[Want to use email?]



```
ast_tests*                ExtToken.h                Iter_4_Files/      Parser.h            regex.cpp        scanner.o
ast_tests.cpp             ExtToken.h.gch            Iter_4_Files.tar   Parser.h.gch        regex.h          scanner_tests*
ast_tests.h               ExtToken.o                iteration3_work.tx Parser.o            regex.o          scanner_tests.cpp
custom_tests4.kix         i1_assessment_tests*      Makefile           parser_tests*       regex_tests*     scanner_tests.h
custom_tests.kix          i1_assessment_tests.cpp   Parser.cpp         parser_tests.cpp    regex_tests.cpp  s.out
eve_of_destruction_tests* i1_assessment_tests.h     ParseResult.cpp    parser_tests.h      regex_tests.h    translator*
eve_of_destruction_tests.h i4_assessment_tests*     ParseResult.h      readInput.cpp       s1.out           translator.cpp
eve_tests.cpp             i4_assessment_tests.cpp   ParseResult.h.gch  readInput.h         scanner.cpp      typelist.dat
ExtToken.cpp              i4_assessment_tests.h     ParseResult.o      readInput.o         scanner.h
```

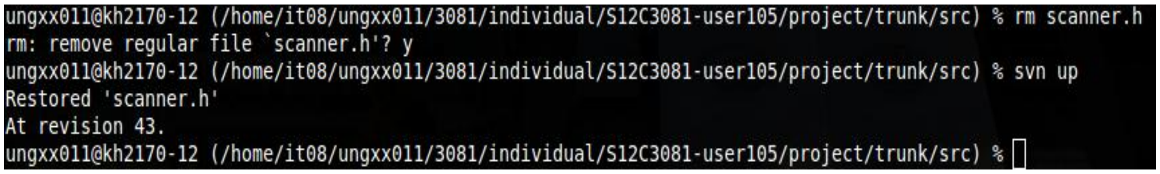
Subversion helps manages all these files, or at least the most important ones. I don't think I could remember which ones were vital to the project if I left it for a month.

3.2) Keeps history of different versions of project

The necessity to keep a history of the project becomes more apparent as the project grows. As of now, I am finishing the project and I still make simple mistakes like hitting the keyboard and scrolling somewhere else. It would take about 5 minutes to solve this solve this small problem but I can easily revert back to the previous saved file if I don't want to deal with it. It might not be a problem for small projects but when the project gets bigger and it takes more time to compile, I would find an easier alternative to do undo small mistake. Instead of spending 15 to 30 sifting through one thousand long line codes just to revert back to the previous state, I can go back to a previous task with a few commands which takes a few seconds.

3.3) Oops, I broke something and I don't know what to do!

There are neat ways to undo the changes you did to the files in the project. One trick I used to undo those changes is to remove it from my working directory then do an update which restores the files from the last save.



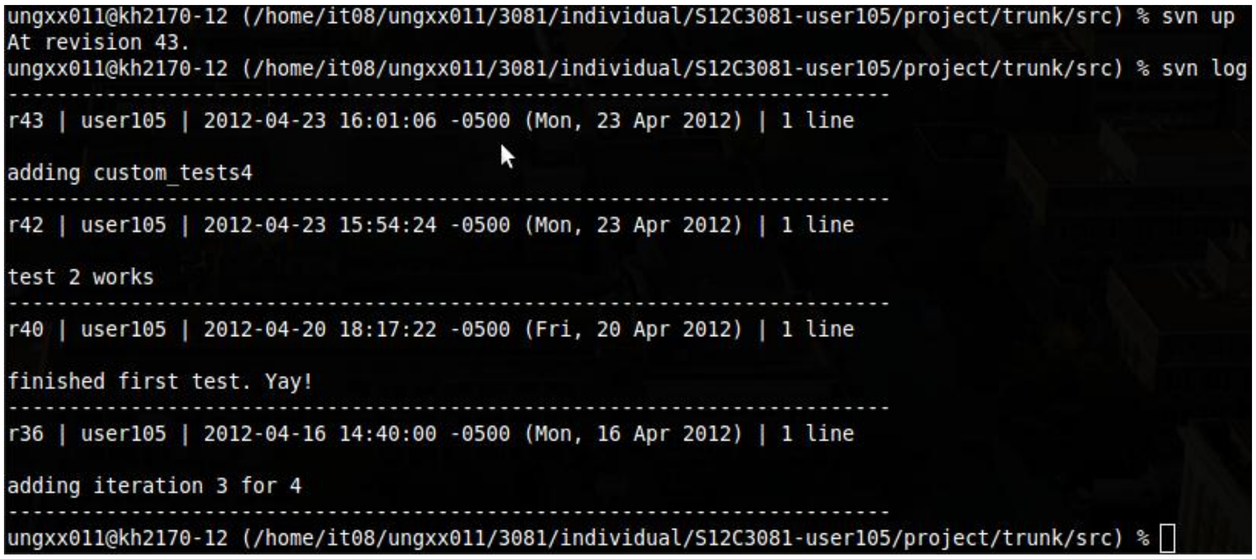
```
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) % rm scanner.h
rm: remove regular file `scanner.h'? y
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) % svn up
Restored 'scanner.h'
At revision 43.
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) %
```

Here, I removed the file and did an update. The update feature restored that file, scanner.h, from the last revision, in this case it is 43.

3.4) Break something even more worse?

The merge command in Subversion allows me to go back to any previous state which I have used in times of need. This saved me a few hours of undoing big changes if I made a big mistake and wanted to go back to a different version.

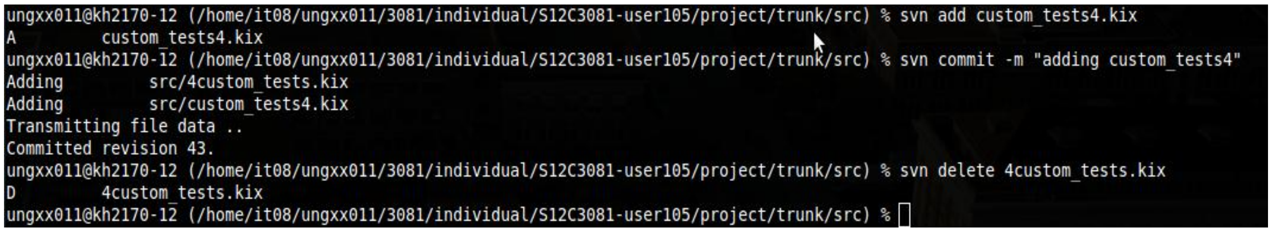
[Here are the number of revisions that occurred in the repository]



```
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) % svn up
At revision 43.
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) % svn log
------------------------------------------------------------------------
r43 | user105 | 2012-04-23 16:01:06 -0500 (Mon, 23 Apr 2012) | 1 line

adding custom_tests4
------------------------------------------------------------------------
r42 | user105 | 2012-04-23 15:54:24 -0500 (Mon, 23 Apr 2012) | 1 line

test 2 works
------------------------------------------------------------------------
r40 | user105 | 2012-04-20 18:17:22 -0500 (Fri, 20 Apr 2012) | 1 line

finished first test. Yay!
------------------------------------------------------------------------
r36 | user105 | 2012-04-16 14:40:00 -0500 (Mon, 16 Apr 2012) | 1 line

adding iteration 3 for 4
------------------------------------------------------------------------
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) %
```

Svn up shows us which revision I am currently on, which is revision 43. Svn log shows all the revisions that happened alongside comments below them.

3.5) Simple commands and rules!



```
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) % svn add custom_tests4.kix
A         custom_tests4.kix
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) % svn commit -m "adding custom_tests4"
Adding         src/4custom_tests.kix
Adding         src/custom_tests4.kix
Transmitting file data ..
Committed revision 43.
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) % svn delete 4custom_tests.kix
D         4custom_tests.kix
ungxx011@kh2170-12 (/home/it08/ungxx011/3081/individual/S12C3081-user105/project/trunk/src) %
```

Svn add, commit, and delete are all the most commonly used commands in the subversion. Add and delete puts a flag on the files as you can see. Commit tells Subversion to update the repository. In this case, I added custom_tests4.kix, giving a mark on the left, then I committed it. I then deleted it, but I have not told Subversion to remove it yet.

3.6) It must get complicated at some point! (And I am not comfortable with command line.)

The basics of Subversion are practical and it isn't necessary to go through a textbook to start using it with your projects. It took me a few days to get familiarized with Subversion, but the cost is insignificant to the benefits of learning how it works. The most important benefit is that it allowed me to spend more time with command line interfaces. If you tend to use IDEs like Eclipse before then Subversion is a great way to start familiarizing yourself with the command line. Maybe you might start preferring command line as you spend more time on it just as I had.

3.7) Still… I bet there is a catch…

Internet not included. Subversion requires the Internet to change the repository and grab data. For the duration of the project, in order to interact with the repository via Subversion, there must be access to the Internet. Beforehand, I could grab the latest update from the repository to work on and leave the Internet. Afterwards, I needed to be connected to the Internet if I wanted submit my finished work.

3.8) Difficulties with setting up!

Setting up subversion can be tricky and time consuming if not done right. Luckily the project already had Subversion set up for me so I didn't have to worry about it. This did not stop me from trying to set it up myself, but I slowly realized that I needed a lot of resources to do so, one being where I should put my repository. Also, inside the individual workspaces, there is a file that allows the users to interact with Subversion. They must be in the correct place otherwise Subversion will not manage files properly. Countless hours of time of time were spent setting up the project alone from the professor and the assistants.

3.9) Features to see what the other guy/gal/s did!

There are some neat features to see what other people changed in the project. When my partner changed some code, I would do an update and see what the difference was between my code and my partner. This was interesting and useful if I saw conflicting interests if we were working on the same files.

3.10) Just don't overdo it

You might think that seeing what others have done and being able to change it is a great feature, but doing it all the time gets frustrating. When my partner and I first started out, we thought the same so we worked on the same file for the project. We worked on the project on separate computers and realized that it was annoying seeing the markers up all the time. We had to go back to the code and erase those markers and this took more time than it was worth. Luckily, we could choose to ignore it and override either of our code completely. The downside is that the work done on by the other was not taken into account.

A marker generally looks something like this:

```
<<<<<<<<<
Either you or your partners work.
=========
The other's work.
>>>>>>>>>
```

Seeing this every five lines of code and having to delete it manually gets frustrating.

3.11) It is still great though. Don't believe me? Others will say the same.

I have only described a few of the benefits I have experienced directly as to why Subversion and other version control are necessary for software programming. Code Complete 2.0 (pg. 667 - 668) describes more briefly as to why version control tools like Subversion are cost effective along with other sources of reference.

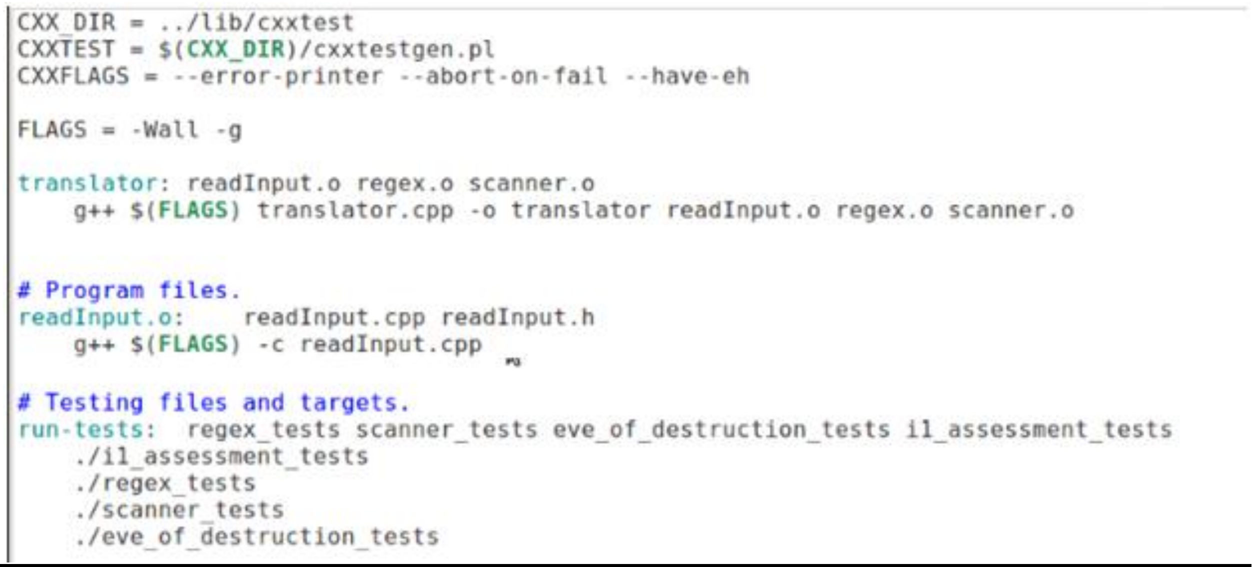
Short Version	
Manages files between users effectively.	Hard to set up.
Easier to see changes from other users.	Potential to be annoying.
Simple command line.	Internet needed to update.

4) Essential Making files of Makefiles

4.0.1) When to use it and who uses Makefiles?

- Small to large software projects with more than one file being compiled.
- Professionals using command line.

4.0.2) What is a Makefile?



```
CXX_DIR = ../lib/cxxtest
CXXTEST = $(CXX_DIR)/cxxtestgen.pl
CXXFLAGS = --error-printer --abort-on-fail --have-eh

FLAGS = -Wall -g

translator: readInput.o regex.o scanner.o
    g++ $(FLAGS) translator.cpp -o translator readInput.o regex.o scanner.o


# Program files.
readInput.o:    readInput.cpp readInput.h
    g++ $(FLAGS) -c readInput.cpp

# Testing files and targets.
run-tests:  regex_tests scanner_tests eve_of_destruction_tests i1_assessment_tests
    ./i1_assessment_tests
    ./regex_tests
    ./scanner_tests
    ./eve_of_destruction_tests
```

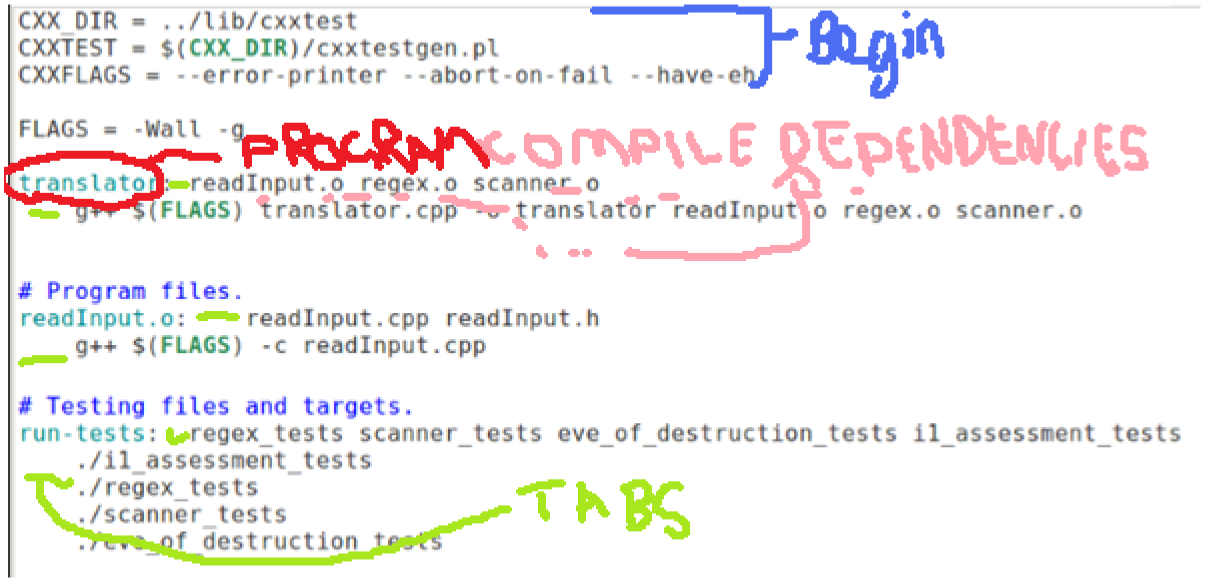
4.1) Why should I use it?

You can compile large projects with one command line. This was vital in saving my time by a minute each time I want to compile my project. This saved me hours of having to recompile my project to make sure everything was updated completely.

4.2) Yeah, but it looks complicated

wow, this is soooo ugly :)

At first, the syntax of makefiles looks overwhelmingly difficult as with learning all subjects. It became much less alien and more enjoyable to make them after I became familiarized with it.



```
CXX_DIR = ../lib/cxxtest
CXXTEST = $(CXX_DIR)/cxxtestgen.pl
CXXFLAGS = --error-printer --abort-on-fail --have-eh

FLAGS = -Wall -g

translator: readInput.o regex.o scanner.o
	g++ $(FLAGS) translator.cpp -o translator readInput.o regex.o scanner.o

# Program files.
readInput.o: readInput.cpp readInput.h
	g++ $(FLAGS) -c readInput.cpp

# Testing files and targets.
run-tests: regex_tests scanner_tests eve_of_destruction_tests i1_assessment_tests
	./i1_assessment_tests
	./regex_tests
	./scanner_tests
	./eve_of_destruction_tests
```

Sloppily highlighted, I have labeled four major components to the makefile. The “begin” shows initial set up. The “program” shows the target of what is being made. “Compile dependencies” refer to the files necessary for the target to compile. “Tabs” means that there are tabs between the spaces and not white spaces. The rest is details and syntax that is covered in tutorials specific to your operating system.

4.3) Okay, what other reason for Makefiles?

Makefiles force users to see how each file relates. This means that you have to plan ahead and see the relation between the files. When I started each part of the project, I first looked at how the files related, then I typed it into my makefile which allowed me to quickly grasp an idea to see how they relate. When you reread your code in a month you might forget how the program relates, but a makefile doesn’t.

4.4) No IDE? No Problem (IDE = Integrated Development Environment)

If you prefer working with the terminal for any reason, you can easily compile different versions of projects using a single command. All IDEs create makefiles, but typing your own makefiles gives you much greater control of your project than having it get automatically generated. In addition, if you have a slow computer, the benefit is that running the terminal with makefiles is quicker than having the IDE load.

[This command runs all the tests I have]

```
% make run-tests
```

4.5) Allows all sorts of ways to make stuff!

Makefiles allow different versions of programs to be compiled. You can have organization in a clutter of files and still be able to separate out what programs you are currently working on. For this project, I had around 4 to 5 different sets of tests that were related to each stage of the project. I had a set of tests for the scanner function, a set of tests for the parser function, and a set off tests for the conversion into C++ code. There was no way I needed to run the scanner function tests when I was working with the parser tests and vice versa. Makefiles allowed me to separate the tests and run them separately.

4.6) Sure, but there has to be a downside…

The most basic makefiles are not universal to all operating systems. Simple commands such as rm on Unix/Linux are not universal to Windows. In fact, you might need to download files to get a makefiles running. I have searched the Internet to realize that there is more than one kind makefile software.

4.7) Also, Makefiles all look the same…

In the beginning, simple mistakes can be frustrating when writing makefiles. When I wrote some sections of the makefiles, I made a mistake of not double checking the files for mistakes. I could not spot the mistake easily since the code I wrote all looked the same. The good side is that once the makefile was written, I didn't need to do anything else.

4.8) Windows doesn't have the Unix make, so this is irrelevant to me!

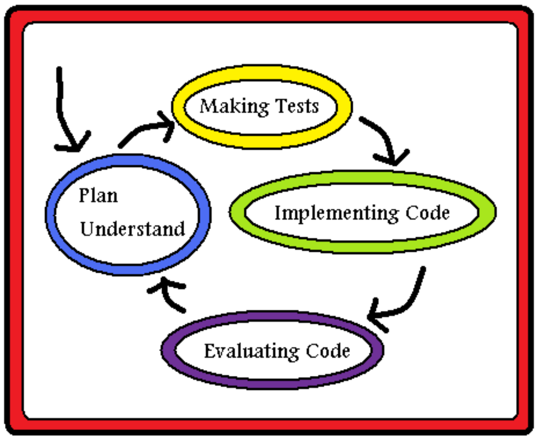
If you want to use make on Windows, you can. Windows has a plethora of different makefiles such as nmake. If you use Windows but prefer the Unix/Linux versions then you can still download the gcc version and implement it on your computer as I did. In fact, you can create rm and ls commands on Windows so that make runs properly and knowing this will enable you to continue using it outside school. I have installed it on my Windows operating system along with a Unix-like terminal to continue practicing using it for my projects.

The Short Version	
Great for command line consoles.	Need to use command line, or customize IDE.
Syntax easy to learn.	Syntax is repetitive.
Lots of kinds of makefiles.	No universal makefile.

5) Essential Iterative Development

5.0) Why should I?

The iterative style of development is a process of four parts: planning and understanding the project, making tests, implementing code, and evaluating the results. You have probably done some projects that had some tests. Maybe you did it at each interval so that they did what needed to be done before moving on to the next part. Iterative development is the most formal way of getting the project constantly moving. By practicing this method of programming, not only will you see practical benefits as your skills develop, but you can also finish more complicated projects in a fixed amount of time.



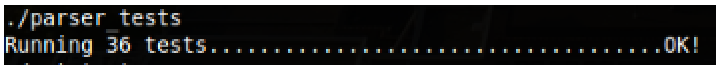
[Defining Requirements]

In my project, the exact details of what needed to be done were often hazy until the professor handed us what needed to be done and it seemed like he made the code up overnight. The details of the program being referred to are called requirements and they often changed in my project. Iterative development plays a major role in keeping my project moving as I will describe in the proceeding sections.

5.1) Breaking it down and keep moving

Iterative development gives you time to think and move. I am sure you have had projects where you had no idea where to begin. Before diving into programming, you probably already think about what you need to code. Sometimes you don't because it is ambiguous, but this method will help you break the ice. At the beginning of each part of my project, I would sit and plan each part of the project that the professor gave us. This allowed my partner and I time to separate out the nuances of each individual part of the project. When we finished scanner, we moved on to the next part of the project. Right after finishing we were given 2000 test cases that our scanner needed to succeed. Luckily, they were independent of the next part of the assignment and we had time to ensure we passed a majority of them. If we had not planned for breaking down the project, we would not be finished with our parser yet and be working on our convert function.

[Wrapping up parser tests, I get a successful run with no errors. No need for scanner.]



```
./parser_tests
Running 36 tests..................................OK!
```

5.2) What if the requirements keep changing?

Often times during the beginning of scanner, parser and convert, there would be some details the program had to have. The professor would then make us add some more details filling in some misconceptions or creating more features. Allowing for planned modularity gave us time to react efficiently because we had broken down the project by practicing iterations. Even though we cringed at the new changes, I never recalled not being completely frozen due to some new changes.

5.3) Time to learn what you are doing

Iterations gave us time to learn about the project and understand the fundamentals. I admit that I was not excited about making a parser at the beginning, but I have grown fonder of it. This might be due to the fact that we did not need to cram in all the information that went along with the project. A complicated project needed more than one day to do and iterations allowed us to keep a steady pace instead of racing through the end.

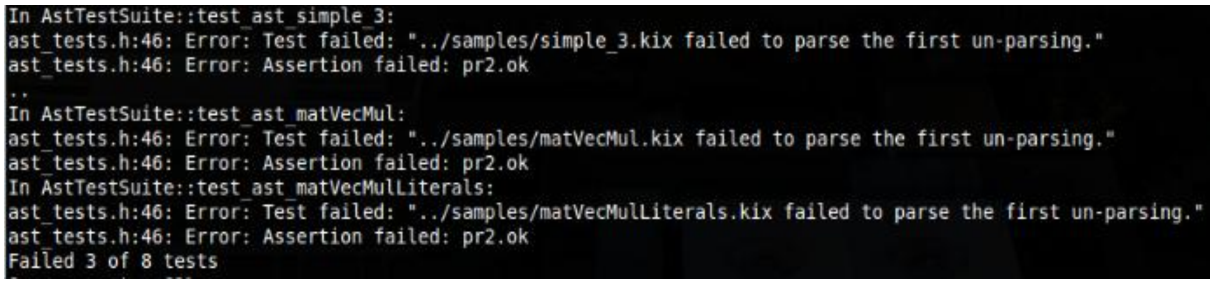
5.4) Chunking code gives natural flexibility and modularity

In your past, you probably wanted to change something in your project but decided it was a risky decision because it would screw the code up. By practicing iterative development, you don't need to worry about this problem because it gives natural flexibility in your code. In my experience with this project, when I had scanner completed I would create another copy of my project directory and work on it separately while the other directory continued on expanding the project. This allowed me to go back and forth between the two as I was ironing out bugs from scanner while also working on parser without fear of dismantling some vital aspects in relating the two. This saved me a lot of time debugging because I was assured that new bugs were from the new features being added to the project.

5.5) Makes it easier to find errors

This next section brings about the need to find errors. From your experience, when you write more code, it probably gets more difficult to track them down. Maybe there were a few tests that you had where they helped you find big mistakes, but sometimes you might not find them until the end of the project. It might be that you could summarize your experiences programming such that a small bug from the beginning of the project might devastate the final product. Even more important is that you probably hit a point in the project where you could not overcome because it was impossible to continue writing working code. By practicing iterations, you will be able to spot bugs faster. When I was working on the scanner function, I labeled tests for each regular expression. When there was a bug, I looked at my console and found which test was causing the errors and I quickly flew over the code to its location. This saved me an hour of having to scan over the code for the simple mistake since I labeled the tests. As I progressed, I reduced my time being frustrated in front of the computer screen by half the length of the project

[Some tests that failed]



```
In AstTestSuite::test_ast_simple_3:
ast_tests.h:46: Error: Test failed: "../samples/simple_3.kix failed to parse the first un-parsing."
ast_tests.h:46: Error: Assertion failed: pr2.ok
..
In AstTestSuite::test_ast_matVecMul:
ast_tests.h:46: Error: Test failed: "../samples/matVecMul.kix failed to parse the first un-parsing."
ast_tests.h:46: Error: Assertion failed: pr2.ok
In AstTestSuite::test_ast_matVecMulLiterals:
ast_tests.h:46: Error: Test failed: "../samples/matVecMulLiterals.kix failed to parse the first un-parsing."
ast_tests.h:46: Error: Assertion failed: pr2.ok
Failed 3 of 8 tests
```

5.6) Makes it easier to guess how much work is need

Oftentimes, you probably don't have a good idea of how much time the project will take. There may be a few times where you know exactly how much time was needed, but those projects were probably not that complicated. In this project, practicing iterations gave me not only a better estimate of how much time to allocate, but also problems that could not be solved via randomly typing. I am sure there were times in your experience where you would randomly type and hope some idea would pop into your head. If I did that in my project, I wouldn't have made it to the last stage, converting KIX into C++. Knowing how time to allocate for each part of scanner, parser, and convert helped me plan out what needed to be done between me and my partner. We could separate the work easier so that we didn't compete as to who finished the same piece of code first.

5.7) Tests for Code Included

By concentrating on writing tests, we ensured that there was a minimal amount of functionality that allowed us to continue to the next part of the project. Each part of the project came with their own tests allowing us to test them separately. This insured that our project was expanding and if we had time at the end, we could go back and rewrite the code. By passing the tests we created, we knew that our functions did what needed to be done giving us a sense of satisfaction. In this project, that satisfaction was vital because it was complex and we needed some landmark to go by to know where we were at.

5.8) But tests aren't fun to do!

For the majority of you reading this paper, you might not enjoy making tests. You might think that if a segmentation fault existed somewhere in the code, you could use a debugging tool because you use it faster than making tests. This might be true and you could probably get away with it in a project like mine. However, I am not good using a debugging tool, so having good tests saved me the time of familiarizing myself with a debugging tool. For my project, making the tests were only painful for a short period of time as I quickly realized it was not as time consuming as I thought it would be, albeit was uncomfortably natural.

5.9) Not so good if requirements are well known

Sometimes you are given a clear and defined task. You know every inch of the way and maybe that task is short. In this case, having to break down the task is unnecessary. Planning doesn't seem to save any time and you only need to do some cod

ing. In these cases, tests may help out to some degree. Iteration becomes unnecessary and perhaps even time consuming.

The Short Version	
Promotes flexibility and modularity.	Think first, code second.
Landmarks and constant progress.	Takes time to plan well.
Find errors faster.	Need to make tests more.

6) End

There you go. I have covered three essential tools for your project that I feel will step up your programming game: Subversion, Makefiles, and the Iterative Style of Development. There are many other tools as well, but these three should give you good grounding even for your current projects.