Due dates:

Late work should only follow an explanation as to why it will be late, except for medical emergencies, etc. Warn me and the TAs, and we'll discuss the right response.

Assignment 1: **April 16 (deadline is MIDNIGHT of deadline day in all cases)**

Assignment 2: **April 23rd**

Assignment 3: **May 7th**

Assignment 4: **May 21st**

Final Project: **June 8 (first day of finals)**

Assignment 1:

1. Create your own copy of the base dominion code, including everything required to compile the code. Check it in the class github repository in a directory: projects/<onid-id>/dominion
2. Read the rules of Dominion, and understand the game sufficiently to be comfortable with testing an implementation of it! If you search online, you can find the official rules and multiple web sites allowing you to play the game for free, as well as forums discussing rules questions for most cards. Your first job is to become a "subject expert" in Dominion, since you will be testing an implementation of it. Note that the primary source of information about the Dominion implementation itself is the dominion.c and dominion.h files provided in the class repository. The specification you use will have to combine this information with knowledge about how the game works, discovered by investigation. This is a typical testing experience, where you are not given a complete specification, but must discover one for yourself.
3. Pick 5 cards implemented in dominion.c. **Refactor** the code so that these cards are implemented in their own functions, rather than as part of the *switch* statement in cardEffect. You should call the functions for these cards in the appropriate place in cardEffect. Check in your changes, with appropriate svn commit messages. Document your changes in a text file in your dominion source directory, called "refactor.txt." Your implementation of at least two of these 5 cards should be incorrect in some way. Do not document the bugs.

Assignment 2:

1. Write unit tests for four functions (not card implementations or cardEffect) in dominion.c. Check these tests in as unittest1.c, unittest2.c, unittest3.c, and unittest4.c.
2. Write unit tests for four Dominion cards implemented in dominion.c. Do not test more than two of the cards you chose to refactor. Write these tests so that they work whether a card is implemented inside cardEffect or in its own function. These tests should be checked in as cardtest1.c, cardtest2.c, cardtest3.c, and cardtest4.c
3. Execute your unit tests and describe any bugs you find.
4. Use gcov to measure code coverage for all of these tests. Report your findings, and describe their implications for the tests in a file called coverage2.txt, also checked in to your dominion directory.
5. Add a rule that will generate and execute all of these tests, and append complete testing results (including coverage %ages) into a file called "unittestresults.out". The rule should be named "unittestresults.out" and should depend on all your test code as well as the dominion code.

Assignment 3:

1. Write a random test generator for two Dominion cards, including the adventurer card. Check these testers in as randomtestcard.c and randomtestadventurer.c.
2. Add rules to the Makefile to produce randomtestcard.out and randomtestadventurer.out.
3. Write up the development of your random testers, including improvements in coverage and efforts to check the correctness of your specification by breaking the code, as randomhistory.txt.

Assignment 4:

1. Write a random tester that plays complete games of dominion, with a random number of players (from 2-4) and a random set of kingdom cards. Check it in as testdominion.c, and add appropriate rules to the Makefile to create testdomininon.out by running this tester.
2. Writing a specification for full games of Dominion is very difficult! Instead of producing a complete specification (though you can check as much as your wish in the tester), write your tester such that it prints key information about the game state to a file called gameResults.out. This file should contain all information needed to completely reproduce a game of Dominion – the choices made by players and the results of actions (did a card play or not, etc.).
3. Pick a classmate with a working Dominion implementation. In a scratch directory, not in the svn repository, run testdominion.c with their code as the dominion implementation. Use diff or another tool to compare the

gameResults.out from their code with your own code's results. Explain these results, and their value in a file called differential.txt.

Project:

In the course of assignments 1-4 you will perform testing on a Dominion implementation. Your project comes in four parts:

1. Use the unit and random tests (both card level and "whole game" level) to test code written or modified by your classmates. Find and document at least three bugs, and produce written bug reports (possibly including pointers to code to expose those bugs). Submit these reports to your classmates by adding them to the directory projects/<onid-id>/bugreports. Each bugreport submitted should be in a file whose name starts with your onid ID, followed by a number: e.g. grocea1.txt.
2. Document the process of identifying and fixing a bug in your own code. You may start with a bug report from a classmate or from your own testing. Show how you used a debugger to understand the problem, and describe how (if) any of Agans' principles applied to the process. Submit this file in your dominion directory as debugging.txt. If any files or logs are discussed in debugging.txt make sure to also submit those!
3. Pick one of the following:
    a. Use delta debugging tools (downloaded from Zeller's website) to minimize a failing test case for Dominion. Describe the process and what you did in deltadebug.txt, and include any relevant code (modifications of the python scripts) or files. One way to go about this is to take your work from Assignment 4 and modify it so that it produces a standalone C file containing calls to play the randomly generated Dominion game.
    b. Implement Tarantula using gcov, and describe the process of using it to localize a bug in Dominion. Write this up as tarantula.txt, and include any relevant code or files.
    c. Organize and moderate an online software inspection of one of your classmates implementations (with their permission!). Document the process in inspection.txt. Ideally, if you lead an inspection for X, X leads an inspection for you. Make sure you have at least 2 other participants. Describe the level of formality, and any bugs found. This probably works best if groups of four work together, with each person leading 1 inspection and each implementation receiving 1 inspection.
4. Write a test report, in pdf format, describing your experience testing Dominion. Document in detail, including code coverage information, the status and your view of the reliability of the Dominion code of at least two of your classmates. This file is submitted as testreport.pdf in your dominion directory. Your test report should have a minimum of 1,000 words of text.