**The Origin of this Course**

This course grew out of the observation that students were not skilled at testing and debugging and that this skill has become required by the industry. The goals of this course are:

1. To show students the differences between testing and debugging.
2. To show students how to do proper software testing.
3. To show students many debugging techniques and tools.
4. To introduce students to how testing is done in industry and the tools used to manage it.

The fourth goal came from discussions with a test lead in industry. She is a developer who became a passionate tester of her company's point of sale product. She pointed out that new graduates came into the industry with little knowledge of the software testing life cycle nor the tools used to manage it. She also pointed out that working in Quality Assurance (QA) is where many new graduates and those on co-op work terms start out. Her input shaped the second half of the course where students get to use tools to manage a small project and do the necessary testing and debugging.

By the end of this course, the student will know how to test and debug code as well as be ready to move into a team in industry that uses QA as part of the software development life cycle.

**Introduction to Testing**

Software is one of the most complex things built by humans. The complexity of modern software is such that it stretches the mental capabilities of almost all of us. Getting it right is far more difficult than it looks. You have to consider:

* will the software produce the correct results?
* will the software work with edge cases like no data?
* will the software have the required efficiency?
* will the software fail under high loads?
* will the software do something sensible when it receives bad data?
* has the software been integrated with the rest of the system so that there are no interactions which cause errors?
* if there is an unanticipated error, can the software recover and return itself to a stable state?

Software testing is the process of testing your software to see if it works and meets all of its performance and functional requirements. In the early days of software development, this area was neglected or given minimal time and effort. Over the years, software got more complex, bigger and the users expected higher quality. Thus, the once neglected area of testing has risen to being a vital part of the software development life cycle.

**Debugging**[**​**](https://software-testing.sdds.ca/A-Introduction/introduction-testing#debugging)

Testing to determine if there are problems in the software is only part of the process. After you determine that the software is incorrect, you need to find and fix the problem. This can be a highly complex and involved process that can take a long of time.

The process of debugging can be broken down into smaller steps:

* locate the source of the bug,
* determine how to fix the bug,
* implement the solution.

The first step, locating the source of the bug, is often the most time-consuming. The job of the developer is similar to that of a detective. You gather evidence and look for clues as to what might have caused the bug. Often, you do not have the information you need and need to insert print statements in the code to produce additional information. Tools like debuggers can also be valuable aids in tracking down bugs as they allow you to pause the code at key points and examine the values of the variables.

**What Does Testing Prove?**[**​**](https://software-testing.sdds.ca/A-Introduction/introduction-testing#what-does-testing-prove)

Testing proves that the tests you ran work correctly. It does not prove that your software is correct. Your software could pass all tests with flying colours yet still have undiscovered bugs in it. The amount of bugs discovered is proportional to the amount of time spent debugging. However, this is not a linear relationship as you discover lots of bugs when you start testing and fewer and fewer as you continue testing. At some point, you reach the point of dimishing returns where it will take enormous time and effort to discover more bugs. This is when testing usually stops. You have to hope that if you cannot find any more bugs, you customers will not find any more either.

Many people think that, after testing, the software is correct. While this might be true, it is usually false. Getting software perfectly correct is almost impossible. Great effort has been expended in finding ways to prove software correct but this has failed for all but the smallest programs.

Another common fallacy is that "I use XXX every day and never see a problem so it must be debugged." Most people use applications the same way every day and do not push the applications to their limits or explore all the features. All you are observing is that the program works correctly for what you use it for. Most applications are shipped with many known bugs. The developers thought that they were not worth fixing because the customer would probably never encounter them or there would be work-around solutions. The fact is that almost all software has bugs in it.

**Testing Versus Debugging**[**​**](https://software-testing.sdds.ca/A-Introduction/introduction-testing#testing-versus-debugging)

Testing and debugging are not the same thing. Testing is the process of running the software and determining that the results are what you expect them to be and, therefore, correct. Debugging is the process of trying to locate and fix a problem in code that you know has a bug in it. While testing and debugging are often carried out one after the other, they are separate processes and should not be confused with one another.

In a large company, the tests will be run by the quality assurance team. Once a bug is discovered, the bug report will be sent in to the project manager or team lead who will then assign a developer to debug and fix the problem. In a smaller company, they might not have sufficient personnel to staff the different roles and the developer might be the one who both runs the tests and fixes the bugs. Regardless of who actually does the testing and debugging, the important point to remember is that they are separate tasks.

**The Need for Testing**

We can see the need for software testing by looking at some of the famous failures in the software field. As we look at a few of these failures, ask yourself what is the root cause and what could be done to prevent these failures in the future. You should also remember that each of these failures was produced by teams who would be considered experts in their fields. This is a lesson to all of us on how easy it is to introduce bugs into software and how we should not assume we have expertise in areas where we do not.

**Famous Software Bugs**[**​**](https://software-testing.sdds.ca/A-Introduction/need-for-testing-2#famous-software-bugs)

The history of software is littered with some of its greatest failures. In this section, we will look at some of the most famous failures in software. These are certainly not all of the failures but a small number that turned out to be significant. Each of these failures happened for a different reason and yet all of them had a significant effect. As you read through these failures, you should look at the reasons behind them including the failure of the human thinking that led to these epic bugs.

**The First Bug**[**​**](https://software-testing.sdds.ca/A-Introduction/need-for-testing-2#the-first-bug)

The first bug was discovered some 70 years ago by Grace Hopper, who was the developer of the COBOL language. She was using the Harvard mark II computer and discovered that a calculation was not being performed correctly. The problem was tracked down to a moth which was stuck between relays inside the computer. As a result, all problems associated with software have been described as bugs.

**Therac-25**[**​**](https://software-testing.sdds.ca/A-Introduction/need-for-testing-2#therac-25)

The Therac-25 was a medical instrument that delivered radiation to cancer patients. The initial version had hardware controls that determined exactly how the machine should operate. At some point in the 1980s these hardware controls were replaced with software controls. The machine could operate in two modes: a low power mode as well as a high-power mode. The machine switched from one mode to the other by spreading some magnets which caused the beam to spread either wide for a shallow dose or narrow for a deeper dose of radiation.

On the fateful day of the incident, the operator made a mistake on the entry keypad which triggered a bug and gave the patient a lethal dose of radiation. Initially, they felt that it was an electrical problem inside the machine, although they could not find any electrical problems. At this point, they did not believe that software could be at fault and therefore did not look at the software to see if there was a bug in it.

Subsequent analysis of the software showed the problem to be a race condition. They had developed the software so that it would run as two separate threads that shared variables. Unfortunately, they had not synchronized access to these variables and it became random which thread would change the variable first.

Further analysis of the incident showed that the programmer who had worked on it had no experience working with concurrent software. Secondly, there had been no formal testing and evaluation of the software by a third party. Thirdly, the system did detect an error which it displayed as "malfunction 54" and the operator had no idea what that meant. Finally, the company itself, Atomic Energy of Canada Limited, said that the programmer no longer worked for the company, had moved and not left a forwarding address.

**The Ariane 5 Disaster**[**​**](https://software-testing.sdds.ca/A-Introduction/need-for-testing-2#the-ariane-5-disaster)

In June of 1996 the first Ariane 5 rocket was launched from French Guiana carrying a payload of scientific instruments into space. Thirty seven seconds after launch, the rocket rotated 90 degrees in the wrong direction, and less than two seconds later started to break up due to aerodynamic forces. This triggered the rocket's self-destruction mechanism causing it to explode in midair.

This disaster cost approximately $370 million and led to a public inquiry asking what happened. It resulted in a delay of scientific research for many years until new equipment could be launched into space. So, the question is, how did the rocket manage to go off course in mid-flight causing its own destruction?

The cause of the disaster was tracked down to a bug in the inertial reference system. A 64 bit floating point value was used to track a value called the horizontal bias. This variable was large enough to hold the required precision but at some point in the calculation the value was assigned to a 16 bit integer. This worked for the first few seconds of the flight, but the value soon exceeded the capacity of a 16 bit integer. The software detected the problem and then populated this variable with a diagnostic value.

Further analysis of the problem yielded a lot of very concerning facts. First, the diagnostic value put into the variable by the processor was intended for debugging purposes only, yet it had been interpreted as a valid flight value. All the Ariane 5 software was inherited from the software for the Arian 4 rocket. The Arian 4 did not actually use the value in its calculations therefore it had never detected any problem with the value. Finally, the Ariane 5 was launched with a much steeper trajectory than the Ariane 4 and this caused the problem to manifest itself.

This disaster shows several common mistakes in software development testing. One is to inherit software from a previous software base without making the necessary changes. The second, is an obvious lack of testing to make sure things worked before using them in an actual production system. The third is related to changes in project requirements late in the project when you decide that you're going to launch the rocket on a steeper path than any rocket has been launched before. Suddenly your software is exposed to a new environment in which it has never been tested. We also see that because the same software had worked on the Ariane 4, they had unreasonable confidence it would work on the Ariane 5. Finally, we see that part of the testing code itself was responsible for the failure. As we look at this situation, we see that the same failures are applicable to many software systems.

**The Mars Climate Orbiter**[**​**](https://software-testing.sdds.ca/A-Introduction/need-for-testing-2#the-mars-climate-orbiter)

The Mars climate orbiter was a $235 million weather satellite designed for Mars. It would orbit the planet over a Martian year and report on the planet's temperature and photograph dust storms. All this information would be sent back to earth for analysis. On September 23rd, 1999 the orbiter was approaching Mars and stowed its solar arrays to protect them during its descent into the upper layers of the Martian atmosphere. It then used its reaction control systems to position itself in the upper Martian atmosphere. At one point it passed behind Mars, losing radio communication with Earth, and was supposed to reappear and start transmitting data. In reality, the craft was not on the correct trajectory and it had actually entered and burned up in the Martian atmosphere.

The story of how this could happen is similar to the story of many airline disasters. It is not the result of a single problem but the result of several different problems occurring at the same time. One of the problems was cosmic radiation. During the flight, the satellite had to reposition itself several times. The calculations for this were done assuming that it was operating in a vacuum. Unfortunately, it had large solar arrays which were catching the sun's rays causing a solar sail effect. This happens because the light from the sun exerts a very small force which actually moves the spacecraft. Although the force is minuscule, over a trip of 196 million kilometers, the effect added up to place the satellite 170 kilometres from where it expected to be.

Investigators also found a bug in the ground control software. Software from a third party used values in an Imperial unit of LB-SEC. NASA's own internal software operated in metric using the unit Newton-seconds. The software which was supposed to convert one value to the other had a small bug in it which led to slight discrepancies in the position of the spacecraft. These were combined over millions of kilometers to yield a significant deviation from the intended course. Worryingly, the quality assurance team had not even discovered that part of the software was using one set of units and another part of the software was using a different set of units.

There was a lack of communication between various parts of the teams. An opportunity to do a course correction was missed because the team was busy controlling three satellites at once. Calculations on the ground were performed manually rather than using the software which the satellite had used and this meant that they were doing one set of calculations while the satellite was doing another set of calculations and no one realized that they were coming up with different answers.

**Summary**[**​**](https://software-testing.sdds.ca/A-Introduction/need-for-testing-2#summary)

This is just a small selection of famous software bugs. You are probably aware of many more bugs that have far less devastating consequences. WHile we are getting better at producing high quality software, we are far away from perfection.

What we should learn from this is that it is easy to get bugs in software and it is very hard to discover and fix them. We also see that many bugs occur not for a single reason but due to many things happening at the same time. These situations make it even more difficult to detect a bug in advance unless you have the ability to anticipate everything that is going to happen in the real world. Software is usually written by developers sitting in rooms filled with computers. They are asked to anticipate everything that can happen in the real world and to program for it. However, they cannot anticipate everything which can happen in the real world.

**Types of Testing**

There is not one single type of software testing, but rather, different types that test the software in different ways. The following sections will investigate each of these different types of testing and show how it contributes to complete testing of your code. This is not an exhaustive list and you might find other types of tests listed elsewhere. This does, however, capture most of the important testing types.

**Unit Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#unit-testing)

Unit testing is one of the most basic types of testing. It breaks the code into small units, usually functions, and tests them to determine if they produce the correct results. The idea is to test small units of code and make sure that they work perfectly before they are combined with other pieces of code into much larger programs. It is easier to find a bug in a small piece of code than it is to find it in larger program. Further, we like to build on code which is known to be solid and bug free. If we know that the individual units that we use in a program are free from bugs, we do not have to look at the low level code to check for bugs but rather look at the way it has been integrated to find the bugs.

Unit tests normally test functions by passing them a known piece of data and checking to see that the result is what is expected. In most cases, this is done by automated tools that run a series of tests and report the results. Later in the course, we will be looking at tools to aid in the running of unit tests.

**Black Box Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#black-box-testing)

Black box testing is a way of designing unit tests. It treats every functional unit as a black box. This means that we have no idea what is inside the black box but we know that if we put a certain value into the box then a certain value should come out of the box. Because it is a black box, we have no idea what is happening inside the box and therefore cannot structure our tests to take advantage of this. Black box testing relies entirely on knowing that a certain input should produce a certain output. Black box testing is normally the first approach to designing unit tests.

**White Box Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#white-box-testing)

White box testing is almost the opposite of black box testing. Whereas with black box testing we know nothing about the internal structure of the code, with white box testing we take advantage of the internal structure of a code to design tests that test every path through the code. If we simply rely on black box testing there, is no guarantee that we would have tested every single path through the code. There could be rarely used pieces of code which have never been tested by black box testing and therefore we have no proof that they work correctly. White box testing produces additional tests which test every path through the code. This results in much higher confidence that the code works correctly.

**Integration Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#integration-testing)

While unit testing tests the low level functionality, we also need to test to make sure that the individual units were combined correctly and that they work as a whole. This is the job of *integration testing* which does not look at the individual functions but looks at groups of functions to ensure that they work together correctly. A unit testing framework can be used to carry out integration testing or it can be carried out in other ways.

**Functional Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#functional-testing)

Functional testing is similar to integration testing but focuses on the business requirements of the application. Functional testing verifies that the output is correct without checking the internal state of the system. An integration test might check that you accessed the database, updated some information, and committed your change. The functional test for the same thing might simply check to see that the correct value had been stored in the database without checking to see exactly how it was done.

**End to End Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#end-to-end-testing)

End to end tests replicate the user behavior and make sure that all the tasks the user would perform work as expected. In some cases, these can be difficult to automate. They can also be difficult to maintain as scenarios of how the user employs the software change over time.

**Load Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#load-testing)

Load testing checks that the software functions correctly under high loads. For a web application it might test to see if it can handle 100 simultaneous connections. A large data application might want to ensure that the database can handle 1000 simultaneous queries and give the required response time. Load testing is often combined with stress testing which tries to take it beyond the normal operating requirements to see when it will actually break. If stress testing reveals that your software will not break until well beyond your expected usage of it, then you are assured your software is in good shape.

**Security Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#security-testing)

Security testing determines if the software meets the security requirements. It might involve checking that every web page can only be reached after signing on to the website. It might involve checking the level of user access to make sure that users can only access the information they are allowed to. It could also involve attempts to break into the system and either steal or alter information stored within it.

**Acceptance Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#acceptance-testing)

Acceptance testing verifies that the application meets all the business requirements. The entire finished application is tested, usually by running examples of how the user would actually use the system. If all of the acceptance tests are met, then it indicates that the customer requirements have been met.

**Regression Testing**[**​**](https://software-testing.sdds.ca/A-Introduction/testing-types-3#regression-testing)

Regression testing means we perform our existing tests after every change to the code. The goal of regression testing is to make sure that any changes we made to the code do not break the existing code. Regression testing is used in many software development methodologies where, after a change is made to the code, the regression tests are run to ensure that it is bug free before being checked into the repository. The goal of regression testing is to make sure that the code checked into the shared repository is free of bugs. This prevents other developers building upon code which has bugs in it.