124 G. Fraser and J. M. Rojas

1. **Introduction**

Software memiliki bug. Ini memungkinkan dikarenakan code dibuat oleh manusia, dan manusia dapat membuat kesalahan. Persyaratan yang salah/ambigu, persyaratan dapat disalahpahami yang dapat membuat software dapat disalahgunakan, developer dapat membuat kesalahan saat menulis source code, yang dapat membuat code yang sebelumnya berkeja menjadi tidak berkerjalagi Ketika terjadi perubahan tanpa pemberitahuan sebelumnya. Penguji perangkat lunak mampu intutif terhadap masalah ini seperti halnya membangun sistem,kemudian menjalankan sistem, dan memeriksa apakah program sudah berfungsi sesuai dengan apa yang diharapkan.

Meskipun prinsipnya mudah, tetapi menguji dengan baik tidak begitu mudah. Salah satu alasan utama untuk ini adalah kemungkinan tes yang ada untuk setiap sistem nontrival. Bahkan sistem sederhana yang mengambil satu nomor bilangan bulat 32 bit sebagai input yang sudah memiliki 232 kemungkinan tes. Manakah dari ini yang perlu kita eksekusi untuk memastikan bahwa kita menemukan semua bug.Jawaban untuk ini adalah bahwa kita perlu mengeksekusi semuanya, hanya saja jika kita mengesekusi semua input dan mengamati bahwa sistem yang dihasilkan output yang diharapkan apakah kita tahu pasti bahwa tidak ada bug. Sangat mudah untuk melihat bahwa semua tes tidak bersekala. Fakta ini diambil dengan baik oleh pengamatan Dijkstra yang terkenal bahwa pengujian tidak pernah dapat membuktikan tidak adanya bug, itu hanya dapat menunjukan keberadaan bug.Tantangan terletak pada menemukan subset dari kemungkinan input ke sistem yang akan memberi kita kepastian yang masuk akal bahwa kita telah menemukan bug utama.

Jika ingin memilih tes yang baik, maka konsekuensinya dapat berkisar dari gangguan ringan pengguna hingga efek buruk bagi pengguna/user. Banyak contoh terkenal tentang kosekuensi bug perangkat lunak, dan dengan peningkatan ketergantungan pada sistem perangkat lunak dalam kehidupan sehari-hari, kita dapat mengharapkan bahwa pengaruh bug untuk perangkat lunak hanya akan meningkat beberapa. Contoh paling terkenal diman bug perangkat lunak menyebabkan masalah perangkat radiasi Therac-25 (Leveson dan Turner 1993) yang memberi enam pasien terkena overdosis radiasi, yang mengakibatkan cedera serius hingga kematian. Contoh penerbangan perdana Ariane 5 (Dowson 1997) yang meledak 40 detik setelah lepas landas karena perangkat lunak yang digunakan pada sistem Ariane 4 belum cukup di uji pada perngakat keras baru, sehingga terjadi masalah akselerasi yang tidak diinginkan pada mobil Toyota ( Kane et al.2010), yang menyebabkan kecelakaan fatal dampak ekonomi yang besar.

In this chapter, we will explore different approaches that address the problem of how to select good tests. Which one of them is best suited will depend on many different factors: What sources of information are available for test generation? Generally, the more information about the system we have, the better we can guide the selection of tests. In the best case, we have the source code at hand while selecting tests—this is known as *white box* testing. However, we don’t always have access to the full source code, for example, when the program under test accesses web services. Indeed, as we will see there can be scenarios where we will want to guide testing not (only) by the source code, but by its intended functionality as captured by a specification—this is known as *black box* testing. We may even face a scenario where we have neither source code, nor specification of the system, and even for this case we will see techniques that help us selecting tests.

Software Testing 125

Even if we have a good technique to select tests, there remain related questions such as who does the testing and when is the testing done? It is generally accepted that fixing software bugs gets more expensive the later they are detected (more people get affected, applying a fix becomes more difficult, etc.); hence the answer to the question of when to test is usually “as soon as possible.” Indeed, some tests can be generated even before code has been written, based on a specification of the system, and developers nowadays often apply a test-driven development methodology, where they first write some tests, and then follow up with code that makes these tests pass. This illustrates that the question of who does the testing is not so obvious to answer: For many years, it has been common wisdom that developers will be less effective at testing their own code as external people who did not put their own effort into building the software. Developers will test their own code gentler, and may make the same wrong assumptions when creating tests as when creating code. Indeed, testing is a somewhat destructive activity, whereby one aims to find a way to break the system, and an external person may be better suited for this. Traditionally, software testing was therefore done by dedicated QA teams, or even outsourced to external testing companies. However, there appears to be a recent shift toward increased developer testing, inspired by a proliferation of tools and techniques around test-driven and behavior-driven development. This chapter aims to provide a holistic overview of testing suitable for a traditional QA perspective as well as a developer-driven testing perspective.

Although many introductions to software testing begin by explaining the life- threatening effects that software bugs can have (this chapter being no exception), it is often simply economic considerations that drive testing. Bugs cost money. However, testing also costs money. Indeed, there is a piece of common wisdom that says that testing amounts to half of the costs of producing a software system (Tassey 2002). Thus, the question of testing well does not only mean to select the best tests, but it also means to do testing well with as few as necessary tests and to use these tests as efficiently as possible. With increased automation in software testing, tests are nowadays often executed automatically: Tests implemented using standard xUnit frameworks such as JUnit are often executed many times a day, over and over again. Thus, the costs of software testing do not only lie in creating a good test set, but also being efficient about running these tests. After discussing the fundamentals of testing, in the final part of this chapter, we will also look at considerations regarding the efficiency of running tests.

124 G. Fraser and J. M. Rojas

1. **Introduction**

Software has bugs. This is unavoidable: Code is written by humans, and humans make mistakes. Requirements can be ambiguous or wrong, requirements can be misunderstood, software components can be misused, developers can make mistakes when writing code, and even code that was once working may no longer be correct when once previously valid assumptions no longer hold after changes. Software testing is an intuitive response to this problem: We build a system, then we run the system, and check if it is working as we expected.

While the principle is easy, testing *well* is not so easy. One main reason for this is the sheer number of possible tests that exists for any nontrivial system. Even a simple system that takes a single 32 bit integer number as input already has 232 possible tests. Which of these do we need to execute in order to ensure that we find all bugs? Unfortunately, the answer to this is that we would need to execute *all* of them—only if we execute all of the inputs and observe that the system produced the expected outputs do we know for sure that there are no bugs. It is easy to see that executing all tests simply does not scale. This fact is well captured by Dijkstra’s famous observation that testing can never prove the absence of bugs, it can only show the presence of bugs. The challenge thus lies in finding a subset of the possible inputs to a system that will give us reasonable certainty that we have found the main bugs.

If we do not select good tests, then the consequences can range from mild user annoyance to catastrophic effects for users. History offers many famous examples of the consequences of software bugs, and with increased dependence on software systems in our daily lives, we can expect that the influence of software bugs will only increase. Some of the most infamous examples where software bugs caused problems are the Therac-25 radiation therapy device (Leveson and Turner 1993), which gave six patients a radiation overdose, resulting in serious injury and death; the Ariane 5 maiden flight (Dowson 1997), which exploded 40 s after lift-off because reused software from the Ariane 4 system had not been tested sufficiently on the new hardware, or the unintended acceleration problem in Toyota cars (Kane et al. 2010), leading to fatal accidents and huge economic impact.

In this chapter, we will explore different approaches that address the problem of how to select good tests. Which one of them is best suited will depend on many different factors: What sources of information are available for test generation? Generally, the more information about the system we have, the better we can guide the selection of tests. In the best case, we have the source code at hand while selecting tests—this is known as *white box* testing. However, we don’t always have access to the full source code, for example, when the program under test accesses web services. Indeed, as we will see there can be scenarios where we will want to guide testing not (only) by the source code, but by its intended functionality as captured by a specification—this is known as *black box* testing. We may even face a scenario where we have neither source code, nor specification of the system, and even for this case we will see techniques that help us selecting tests.

Software Testing 125

Even if we have a good technique to select tests, there remain related questions such as who does the testing and when is the testing done? It is generally accepted that fixing software bugs gets more expensive the later they are detected (more people get affected, applying a fix becomes more difficult, etc.); hence the answer to the question of when to test is usually “as soon as possible.” Indeed, some tests can be generated even before code has been written, based on a specification of the system, and developers nowadays often apply a test-driven development methodology, where they first write some tests, and then follow up with code that makes these tests pass. This illustrates that the question of who does the testing is not so obvious to answer: For many years, it has been common wisdom that developers will be less effective at testing their own code as external people who did not put their own effort into building the software. Developers will test their own code gentler, and may make the same wrong assumptions when creating tests as when creating code. Indeed, testing is a somewhat destructive activity, whereby one aims to find a way to break the system, and an external person may be better suited for this. Traditionally, software testing was therefore done by dedicated QA teams, or even outsourced to external testing companies. However, there appears to be a recent shift toward increased developer testing, inspired by a proliferation of tools and techniques around test-driven and behavior-driven development. This chapter aims to provide a holistic overview of testing suitable for a traditional QA perspective as well as a developer-driven testing perspective.

Although many introductions to software testing begin by explaining the life- threatening effects that software bugs can have (this chapter being no exception), it is often simply economic considerations that drive testing. Bugs cost money. However, testing also costs money. Indeed, there is a piece of common wisdom that says that testing amounts to half of the costs of producing a software system (Tassey 2002). Thus, the question of testing well does not only mean to select the best tests, but it also means to do testing well with as few as necessary tests and to use these tests as efficiently as possible. With increased automation in software testing, tests are nowadays often executed automatically: Tests implemented using standard xUnit frameworks such as JUnit are often executed many times a day, over and over again. Thus, the costs of software testing do not only lie in creating a good test set, but also being efficient about running these tests. After discussing the fundamentals of testing, in the final part of this chapter, we will also look at considerations regarding the efficiency of running tests.