1-1-2010

The tribe & the software tester

Piotr "Pete" Wegier
Ryerson University

THE TRIBE & THE SOFTWARE TESTER

by

Piotr "Pete" Wegier

Bachelor of Science, Ryerson, 2008

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The Tribe & the Software Tester

Master of Science 2010

Piotr "Pete" Wegier

Computer Science

Ryerson University

The human mind is truly a marvel and with it humanity has managed incredible feats in so short a history it is staggering. Even so, the mind is flawed. Psychologists have shown time and again that we can fail–constantly and predictably–at even simple and common tasks, such as tracking changes in a visual scene. As software development races forward, and the need for human software testers continues, these cognitive biases must be considered and compensated for during the software testing process. This thesis examines a number of these biases and their applicability to software testing, it proposes the integration of aspects of psychology into the software testing process, and shows experimentally that the human software tester can very easily be tricked and manipulated in simple ways.

Acknowledgements

Thanks must first go to my supervisor, Dr. Peter Danziger. He took me on when I lacked a supervisor, encouraged me when I needed it, talked me down when I was worried, and is the reason this thesis is complete. Peter excellently played the triple role of wise teacher, guiding mentor, and good friend. His contributions to this work will not be forgotten.

For all her help, my thanks also go to Dr. Julia Spaniol of Ryerson's psychology department. Julia provided helpful advice on the psychological aspects of this work, including poking a number of holes in preliminary designs for the experiments. Her help made this work much more rigorous and brought it up to proper experimental snuff. For all her good deeds, she is now stuck with me for the next few years as I further avoid the real world through more graduate schooling.

Professor Sophie Quigley of Ryerson's computer science department also deserves my thanks for allowing me the use of her usability lab, in which I carried out my experiments.

My final appreciation and thanks go to my parents, Ewa and Richard; my friends, Aidan, Pat, and Sima; and my love, Madhu.

Dedication

For my parents.

Thanks for everything.

Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

The Introduction

There's a sign on the wall but she wants to be sure
'Cause you know sometimes words have two meanings

—"STAIRWAY TO HEAVEN" BY LED ZEPPELIN

1.1 *With that lineup, you'll go down like a lead zeppelin*

It is the winter of 1970-71 and English rock band Led Zeppelin prepare their fourth studio album. Much of the recording work will be done at the remote estate house of Headley Grange and these sessions will go on to produce one of the greatest rock albums of all time. Bassist John Paul Jones will write the famous riff for the song "Black Dog". Drummer John Bonham will record the legendary and oft-copied beat for "When the Levee Breaks". Vocalist Robert Plant will deliver what will become his crowning achievement as a vocalist and lyricist. Finally, Led Zeppelin's guitarist, songwriter, and leader, Jimmy Page, will create some of the greatest rock music in history. His guitar solos, arrangements, and cutting-edge production techniques will generate an album that will go on to sell over 37 million copies, making it one of the highest selling hard rock albums of all time [10, 29].

***

On 8 November 1971, Led Zeppelin's fourth untitled studio album was released. The final song on side A of the album was called "Stairway to Heaven", often cited as the most requested and most played song on the radio of all time. A 1991 magazine article claimed that since its release, the amount of radio time for "Stairway to Heaven" was equivalent to 44 years of continuous play [7]. Its popularity with budding guitarists is so great that there is tell of guitar shops that forbid customers from playing the famous guitar solo when trying out guitars. However, the most commonly repeated story about "Stairway" is that the song contains a set of hidden messages within its lyrics that can only be heard if the song is played backwards.

To create these types of hidden messages, a technique known as *backmasking* is employed and is typically done with a vocal track that is then hidden amongst the music of a song. To do this, record the voice message that you would like to hide, reverse it, and insert it within the song of your choice. If played back normally, you would hear the music plus the reversed message, which would sound like unintelligible gibberish. However, if the song is played backwards, the reversed portion would be played back in its original direction and would be completely comprehensible. This technique has been used (or said to have been used) numerous times in musical history. A conspiracy theory came about when it was claimed that the Beatles used backmasked lyrics on the song "Revolution No. 9", which supposedly revealed that bassist Paul McCartney had died and been replaced by a lookalike. People then claimed to have discovered other "clues" pertaining to this theory in other Beatles songs, such as on the song "Strawberry Fields Forever" which contains a fadeout with the words "I buried Paul". Forgetting the difficulty of covering up the death of one of the most famous people in the world, from one of the most well-known bands in history, members of the Beatles explained that the words heard were actually "cranberry sauce", a fact that can quickly be confirmed by actually listening to the song. Some who believe that backmasking is done for nefarious purposes make the nonsense claim that a person listening to a song containing backmasked messages would subconsciously hear the hidden messages, understand them, and accept them as fact; the person would not even have to listen to the song in reverse [6].

Other musicians and groups quickly began to backmask lyrics into their own works, simply to ridicule those who believed that someone would hide potentially career-destroying information in their own music. In 1979, Pink Floyd released "Empty Spaces" on their album *The Wall*. The song contained the backmasked lines "Congratulations! You just discovered the secret message. Please send your

answer to Old Pink, care of the Funny Farm". The messages pokes fun at the backmasking hysteria and also references founding member and original vocalist of Pink Floyd, Syd Barrett. Barrett ("Old Pink") had left the band by the time *The Wall* was released due to severe mental health problems and was believed to be seeking help from mental health professionals ("the Funny Farm"). An extreme example of this baskmasked ridicule was done by Frank Zappa, first hiding a profanity-laced tirade in "Hot Poop" from the album *We're Only in It for the Money* by his band The Mothers of Invention, and secondly the song "Ya Hozna" from his solo album *Them or Us*, in which all the lyrics of the six minute song were backwards.

In 1981, a Christian radio DJ named Michael Mills claimed that Led Zeppelin had hidden satanic messages in "Stairway to Heaven", albeit in a very different way. When backmasking is normally done, the reversed message tends to sound like gibberish. The content of the reversed message only becomes clear when played in reverse. However, Mills claimed that the messages hidden in "Stairway" were not only satanic in nature, but were hidden within the actual lyrics of the song. (Before continuing, I urge readers to visit `http://jeffmilner.com/backmasking.htm` to see this effect in action. Click on "Stairway to Heaven" and listen to the song forwards. Then listen to the song in reverse and see if you can hear anything strange. Finally, listen to the song in reverse but click "Show Reverse Lyrics" to see the supposedly backmasked lyrics. The rest of this chapter will be much more exciting if you do all this.) This specific portion of the song is often cited:

If there's a bustle in your hedgerow, don't be alarmed now,
It's just a spring clean for the May queen.
Yes, there are two paths you can go by, but in the long run
There's still time to change the road you're on.

When played in reverse, believers in the "Stairway"-satanic-backmasking theory claim you will hear:

Oh here's to my sweet Satan,
The one whose little path would make me sad, whose power is Satan.
He'll give those with him 666.
There was a little toolshed where he made us suffer, sad Satan.

If we assume for a moment that Robert Plant, Led Zeppelin's vocalist and lyricist, did this purposefully when he wrote the words to "Stairway to Heaven", we would have to conclude that he must be

the greatest lyricist of all time. To be able to write lyrics to music that already existed,[1] have the lyrics make sense when played forwards and backwards, and for the backward lyrics to be praises for his supposedly beloved Satan, Plant must be incredibly gifted, perhaps even blessed with supernatural powers.[2] Or more likely, Michael Mills and his ilk, spinning rock records in reverse in their basements, have been fooled by the cognitive biases that we are all subject to as part of the human condition.

We like to think of ourselves as intelligent and logical beings but there are many things that we simply are not very good at, with the "Stairway"-backmasking theory being a particularly telling example. Michael Shermer argues in [30] that this idea of finding hidden lyrics in songs is a result of humans being pattern-seeking animals. Our ability to derive connections between various events has allowed us to survive over thousands of years. If the berries make you sick when you eat them, Then avoid eating them. Being a Christian, concepts such as Satan are part of Mills' life. Hearing Satan mentioned in a song is just the pattern seeking portion of Mills' brain doing its job and looking out for a familiar idea.

But why do people hear the lyrics if you play the song backwards for them? Firstly, the believers cannot agree on a single version of the backward lyrics [6, 7], the version given above is the one I have encountered the most.[3] When you play the song in reverse for someone for the first time all that is typically reported to have been heard is gibberish. A person will report to hear the "hidden lyrics" only if you show them the lyrics that are supposedly there. This act of prompting the person results in a form of *expectancy effect*, specifically the *observer-expectancy effect*. If an observer (someone listening to "Stairway" in reverse) expects a particular result (evil lyrics of praise for "my sweet Satan"), the brain will make the observer believe that they have indeed had that experience. It is almost as if the brain is given a square peg to fit into a round hole and simply hammers the peg in.

These ideas all fall under the category of *cognitive biases*, various common mistakes that humans make, predictably and consistently. Although we believe we are logical and reasoning beings, we, as a species, constantly make the same kinds of mistakes and fall into the same types of cognitive traps, over and again. These biases can be as small and unremarkable as hearing odd things in our favourite rock songs, or can be much greater and much more problematic.

[1] Jimmy Page had written the music of the song earlier.
[2] One can only assume that these were of a satanic nature.
[3] That a single version of the lyrics cannot be agreed upon is highly suspect.

1.2 The Idols of the Mind

Consideration for the presence of cognitive biases has previously been done in history. In 1620, renowned philosopher, statesman, and scientist Francis Bacon published the *Instauratio Magna*, an incomplete, but no less revolutionary, composition. Within the various philosophical works of the *Instauratio Magna* was the *Novum Organum* [4], a book which laid the foundations of scientific thought and the scientific method thereafter.

In his book, beginning with aphorism XXXIX, Bacon addressed what he called the *Idols of the Mind*, common errors, each arising from a distinct source, which can cause us to misinterpret nature and the world around us. In other words, these were cognitive biases. He named four classes: *The Tribe*, *The Cave*, *The Market Place*, and *The Theatre*, each described below.

The Tribe

> "The Idols of the Tribe have their foundation in human nature itself, and in the tribe or race of men. For it is a false assertion that the sense of man is the measure of things. On the contrary, all perceptions as well of the sense as of the mind are according to the measure of the individual and not according to the measure of the universe. And the human understanding is like a false mirror, which, receiving rays irregularly, distorts and discolors the nature of things by mingling its own nature with it."[4]

The Tribe has its basis in the fact that we, as humans, experience a world that differs from the real world. Our perceptions are those of beings with limited senses and limited cognitive abilities and, unfortunately, these limits are built into humanity and impossible to remove. We make numerous mistakes and assumptions in our perception of the world around us, such as the imposition of order where none may exist. We seek evidence that supports our beliefs and discount that which does not, even when the contradictory evidence is overwhelming. We will assume that since one thing led to another there exists a causal link between the two. However, correlation does not imply causation, and often evidence that denies this link is ignored. Bacon points out that "[...] what a man had rather were true he more readily believes", and thus we are active in our distortions of the world around us. We reject that which is difficult, hard to hear, or that goes against the grain, and rather accept the easier routes of laziness, ignorance, and conformity. These limitations–these biases–are responsible for

[4] *Novum Organum*[4], book one, aphorism XLI.

the idea of hidden satanic lyrics in rock music described in the previous section. We expect to hear something and our brain will makes us hear it when it does not occur.

The Cave

> "The Idols of the Cave are the idols of the individual man. For everyone (besides the errors common to human nature in general) has a cave or den of his own, which refracts and discolors the light of nature, owing either to his own proper nature; or to his education and conversation with others; or to the reading of books, and the authority of those who he esteems and admires; or to the differences of impressions, accordingly as they take place in a mind preoccupied and predisposed or in a mind indifferent and settled; or the like. So that the spirit of man (according as it is meted out to different individuals) is in fact a thing variable and full of perturbation, and governed as it were by chance. Whence it was well observed by Heraclitus that men look for sciences in their own lesser worlds, and not in the greater or common world."[5]

The focus of the Cave is the person and their individual and unique set of peculiarities. A person's tastes, interests, habits, education, heroes and villains, all play a part in forming their personality, and it is that personality which has its own set of hurdles for understanding. Education and interests sharply focus the mind in that direction, such that chemists may see chemistry everywhere they look, physicists see physics, and we computer scientists see computer science wherever we turn.[6] While some minds focus on differences, others focus on resemblances, or some on the novel, others on the established. Balances must be struck between each to prevent the mind from stumbling and misunderstanding the world.

The Market Place

> "There are also Idols formed by the intercourse and association of men with each other, which I call Idols of the Market Place, on account of the commerce and consort of men there. For it is by discourse that men associate, and words are imposed according to the apprehension of the vulgar. And therefore the ill and unfit choice of words wonderfully obstructs the understanding. Nor do the definitions or explanations wherewith in some things learned men are wont to guard and defend themselves, by any means set the matter right. But words plainly force and overrule the understanding, and throw all into confusion, and lead men away into numberless empty controversies and idle fancies."[7]

[5] *Novum Organum*[4], book one, aphorism XLII.
[6] A terrifying and truly depressing thought.
[7] *Novum Organum*[4], book one, aphorism XLIII.

Whereas the Tribe and the Cave focus on man and man's particular dispositions, the Market Place focuses on the communication between men. We believe that we control words and use them as we please to express our thoughts. But words can distort our intended meaning when heard by someone else, as not all people have the exact same set of word definitions in their minds. Definitions have their own set of problems as they themselves are created out of words. Bacon points out two specific categories of confusion in the Market Place. Firstly, words that are stand-ins for things that do not exist and therefore are sloppy and loosely defined. Secondly, words for things that do exist, but with vague definitions that can vary widely between individuals. For example, "dark matter" would belong to the former category, while Bacon himself cities the word "humid" as belonging to the latter.

The Theatre

> "Lastly, there are the Idols which have immigrated into men's minds from the various dogmas of philosophies, and also from wrong laws of demonstration. These I call Idols of the theater, because in my judgement all the received systems are but so many stage plays, representing worlds of their own creation after an unreal and scenic fashion. Nor is it only of the systems now in vogue, or only of the ancient sects and philosophies, that I speak; for many more plays of the same kind may yet be composed and in like artificial manner set forth; seeing that errors the most widely different have nevertheless causes for the most part alike. Neither again do I mean this only of entire systems, but also of many principles and axioms in science, which by tradition, credulity, and negligence have come to be received."[8]

The Theatre is concerned with the biases that one acquires when adhering to a specific system of philosophies. Bacon is primarily concerned with three different (erroneous) classes of philosophy here: (a) those that grasp at a few instances, not really examining them in detail, and making grand assumptions and speculations; (b) those that focus on a very small number of experiments or insights, no matter how well this tiny base has been examined, and build their theories and ideas upon it; and (c) those that mix philosophy and theology where "[...] the vanity of some has gone so far aside as to seek the origin of sciences among spirits and genii." Bacon is warning us of the dangers of adhering to erroneous methods of acquiring knowledge and how we can miss things if we follow these systems blindly.

[8] *Novum Organum*[4], book one, aphorism XLIV.

1.3 The Idols of the Software Tester

Software testing holds a unique position in that it requires human beings to examine, often in great detail, rather odd things. Not only must we check that software functions correctly, we must also check that it looks correct. This can often lead to people checking if the colours of borders are correct, or if the button with the little floppy disk is centred, or if the man on the screen runs and jumps in accordance to the laws of physics. These are bizarre tasks and only a small fraction of the odd things that human software testers must look out for. Due to this vast variety of tasks we have a vast number of points of failure. Any one of the biases Bacon described may creep in and impact the software testing process. I realized that Bacon's *Idols* can be used as a framework to examine and improve the field of software testing. Bacon's original intent with *Novum* was to help refine the scientific process, thus allowing scientists a clearer and greater understanding of the universe. Software testing is similar as testers must use the process of software testing (their science) in order to learn more about the problems in software (their universe), and as such, we also must be wary of our own set of idols. I propose that we use Bacon's Idols as a framework to reexamine the current state of the art of software testing and work to improve it by integrating safeguards against the various cognitive biases and logical fallacies that plague the human mind. To approach this task, I have reframed Bacon's original definitions into testing-specific versions.

The Tribe

The Tribe of the software tester is very similar to Bacon's version. It is made up of all the pitfalls and limitations of the human mind: cognitive biases, logical fallacies, tricks of the mind, and various types of illusions (optical, auditory, etc.). Combatting these is not as difficult as it may seem; the key is in being aware of their existence and making the necessary allowances. This thesis focuses on the Tribe and the software tester and as such, this topic will be discussed in great detailed in later chapters.

The Cave

The Cave of the software tester encompasses the biases inherent in the specific disposition of the tester. These vary from person to person and are rooted in the character, disposition, social mores, but primarily the training of the tester. Testers with no programming experience will approach testing as a

black box that must be prodded and poked and have its reaction examined. Their testing will be less restrained than that of the programming-experienced tester but this freedom has its own limitations in the form of ignorance. Testers with programming experience will often think about the code itself. They will speculate about its structure and organization–if they do not have access to it–and their testing will be dictated by these speculations. The programming languages known to this brand of tester will also influence their thinking. Certain languages will focus on a certain type of control flow through the program, and the tester will model their approach on those experiences, whether they are correct to apply or not. Some languages will have exposed the tester to very specific forms of testing useful to those languages but perhaps not universally applicable to all software under test.

The Market Place

The Market Place of the tester is not restricted to tester-tester communications but must also include tester-developer and tester-manager dialogues. The vocabulary of software testing is not in any way uniform across all companies and all individual testers but is standardized enough to allow for adequate tester-tester conversation. The latter two categories mentioned above are the main sources of concern.

Tester-developer communication is very important as the person testing the application is rarely the person who wrote it and, therefore, not the person who will be fixing it.[9] This line of communication must remain clear and free of confusing or misleading information. If the developer who is repairing the software cannot accurately reproduce the bug reported they may simply close it and the bug will remain in existence. Thus, it is the responsibility of the tester to be as clear as possible when reporting any bug they have discovered. Descriptions must be free of all unnecessary and confusing information, step by step instructions should be included for bug reproduction, and the tester should use all additional forms of media necessary (screen image captures, video captures, cocktail napkin drawings, etc.) to provide the needed detail to the developer.

Tester-manager[10] communication can be significantly different to the previous two forms. Whereas a developer is versed in technical jargon and the connotations carried thereby, a manager may be of

[9]The developer who wrote the buggy code originally may also not be the person who goes backs to make the needed repairs. This is especially true in large corporations or in open source software projects.

[10]The term *manager* here is used to denote any individual to whom the tester may be reporting, or who has decision making power over the project, and may not posses a substantive technical background.

a non-technical background and be ignorant to the differences in meaning among certain terms. For example, a manager may be more concerned about bugs that are labelled as "Application Failure" over "Cosmetic" bugs, but may not realize that the latter can be just as difficult to find and damaging to a product as the former, even though the former sounds more detrimental.

The Theatre

The Theatre of the software tester covers the problems inherent to various software testing techniques. Specifically, I am referring to the limitations of application of each technique. Some software testing philosophies will be better suited to a certain type of software. Others may be better at dealing with the intricacies of a peculiar language. Still others will be convenient for replicating various types of usage of the software. In short, we cannot focus on using just one tool for the job, as testers we must use the right tool for the right job.

1.4 Goals, Scope, and Organization

The goal of this thesis is threefold: awareness, assessment, and adaption. Firstly, as software testers, we must be made aware of the problem of cognitive biases. Chapter 2 introduces a number of biases[11] that I believe are applicable to the field of software testing. By no means does this list exhaust all biases applicable to software testing, and the biases included are not always applicable to all possible software testing scenarios. This chapter provides overviews of the biases themselves and a look at significant papers and findings of each. To assess the impact of these biases on software testing, Chapter 3 examines software testing, existent test case writing strategies, and past attempts to combine the idea of cognitive biases with software development in general. Chapter 4 presents a pair of experiments I designed to examine the impact of biases on a software tester and the results are discussed in Chapter 5. Attempts have been made at considering biases in other aspects of software development but never to testing. Finally, Chapter 6 presents a summary of the work and a discussion of possible guidelines that could be implemented to adapt the testing process to consider biases.

[11] From this point forward, I will use the word "bias" as a shorthand to denote all various aspects of the Tribe as it applies to software testing.

Chapter 2

The Tribe

The choice I made doesn't seem to make much sense

—SUBJECT IN THE COLD WATER IMMERSION TRIALS [14]

We are all bound up by the limits of our tribe–by the limits of the human condition. Bacon's original definition of the Tribe focused primarily on biases of judgement, those that could negatively impact the process of scientific discovery. In addition, we must also consider biases in the realms of vision, attention, and decision making. The following chapter examines (a) attentional biases in the forms of *change blindness* and *inattentional blindness*; (b) ways to influence judgement and valuation with *anchoring and adjustment effect*; (c) how memory can be manipulated just via the order events are presented in with *peak-end effect*; and (d) *expectancy effects* and their impact on our actions and experiences.

2.1 Change Blindness

As a film is being produced it is rare for scenes to be completed in any kind of order as they are often shot out of sequence and over a period of weeks or months. Inevitably, small details in the scene, such as which hand a character is holding something in or the placement of objects on a table, will change. This results in a break in *continuity* and many productions attempt to avoid these breaks by hiring *continuity directors* to monitor for them; however, these changes do slip in but audiences rarely notice

them. This inability to notice is known as *change blindness*, during which "observers do not appear to retain many visual details from one view to the next" [33]. Many experiments have been conducted that prove the existence of this effect, and with a remarkable range of variations on each.[1]

2.1.1 Changes During a Saccade

A *saccade* is a very fast movement of the eyes, and it is how our eyes move from one interesting point to another when we look around. In [11], Grimes presented a study in which subjects were tasked with studying photographs for a memory test. Subjects were warned that parts of the images may change as they studied them; these changes occurred during saccades. For example, as a participant studied a picture of two men in hats the two men would switch hats, or even their heads may have been swapped as a saccade occurred. Participants reported no swapping of hats, and only 50% noticed a swapping of heads. Numerous other changes were made as the participants studied the images but only about 30% of all changes were ever reported.

2.1.2 Changes During Simulated Saccades

Another common technique for demonstrating change blindness is known as the *flicker paradigm*. An image is presented for a brief moment, followed by a blank screen, followed by a modified version of that image. This cycle is repeated until a participant reports a change or some set time limit is reached. Simons and Levin [33], reported that a participant can almost never notice the change during the first cycle of flicker and some even miss changes after a minute of watching the same cycle. These studies demonstrated that during an interruption (either a real or simulated saccade) little information is retained in order to notice that something has changed. One possible explanation was that the saccade acts to hide the change from the participant.

To counter this, O'Regan, Rensink, and Clarke [22] presented the idea of *mudsplashes*. Participants were presented with a series of images and after viewing an image for three seconds a modified version was presented. As the change occurred, six groups of black and white dots (the mudsplashes) appeared on the screen, but did not obscure the changes themselves. This was then cycled in much the same way as the flicker paradigm. The changes were either central-interest changes, such as changes to a per-

[1] For a fun example of change blindness, see `http://www.youtube.com/watch?v=ubNF9QNEQLA`

son; or marginal-interest, such as a guard rail appearing/disappearing. It was found that participants were able to detect central-interest changes as they occurred; however, marginal-interest changes were much more difficult for participants to detect. Participants missed 13-30% of marginal-interest changes even after 40 seconds of viewing the cycle. This result may have partially occurred because the mudsplashes drew attention away from the changes. A followup–in which a single mudsplash occurred at the point of change–showed that while central-interest changes are encoded and participants did notice differences, marginal-interest changes were rarely reported.

2.1.3 Changes to Central-Interest Objects

Previous work seemed to indicate that central-interest changes were detected without the same difficulty of marginal-interest changes. If a single object of central-interest were to change during viewing a participant should notice the change; however, Levin and Simons [19] attempted to disprove this idea. Participants watched a short video of a person sitting at their desk, hearing a phone ring, and walking over to answer it. The video consisted of two cuts: the person getting up from their desk and walking out of frame, followed by the person walking up to a wall-mounted phone and answering it. Between the two cuts, the actor portraying the person changed. The two actors were of similar appearance (gender, race, hair colour, etc.), but were nonetheless noticeably different. Their hair styles were different, one actor's shirt was blue and the other's grey, one shirt was buttoned and one was not. Participants were asked to watch the video and then provide a written description of what they had seen. Of 40 participants, only 33% reported the change in actors, even though some had provided very detailed descriptions of the action depicted. When asked directly whether a change in actors had occurred, only two additional participants claimed it had.

The same researchers, Simons and Levin, continued this work in [34] in order to examine changes during real-world interactions. The following situation was created: one of the two experimenters would approach a pedestrian walking on a university campus and ask for directions. After a short period of conversation, workers carrying a large door would walk between the experimenter and the pedestrian and obscure their view of one another for a moment. During this time, the experimenter would be replaced by a second experimenter who would then resume the conversation. The two experimenters wore different clothing, were of different heights, and had noticeably different voices. After the pedestrian had finished giving directions, the second experimenter (the new "original" ex-

perimenter) would ask if the pedestrian had noticed anything remarkable as the door passed. If no change was reported, the pedestrian was directly asked if they noticed that the person who started the conversation was not the person who ended it. Of 15 participants, only 7 reported noticing the swap.

2.1.4 Changes Without Visual Interruption

Thus far, all research into change blindness has required some form of interruption to occur in order to either mask or distract from the change. These have included saccades, blank screens, mudsplashes, film cuts, and doors bisecting conversations in order to demonstrate change blindness effects. In [32], however, Simons and Franconeri presented a study in which no visual interruption occurred. Participants were instructed to search for changes in a series of photographs displayed on a computer screen. These changes were accomplished by presenting a photograph followed by a modified version of the same photograph. The modified version was created by digitally adding or removing some object from the photograph or changing the colour of an object. The additions, deletions, and colour changes always resulted in plausible scenarios in the modified photograph (i.e. no people with missing heads or purple dogs). The changes were presented as a series of 12 second movies, with the original version of the photograph fading into the modified version. This would give the effect of an object slowly appearing, disappearing, or changing colour. A control group was given the same series of original/modified photos; however, the original would be presented for 11.25 seconds, followed by a 0.25 second blank grey screen, followed by the modified image. For both groups, once the modified image had been fully presented, participants were asked to identify the location of the change. It was found that the participant groups were approximately equal in their abilities to detect changes under both the gradual and disruption conditions.

It appears that although we can view and focus our attention to a scene, very little information is kept in memory to be used to make comparisons. The results of [22, 32] in particular show that change blindness can occur without direct visual disruption. This has potentially severe consequences as large changes can occur without being noticed, even though nothing has obscured the changes themselves.

2.2 Inattentional Blindness

Following from the previous section, *inattentional blindness* is a related effect which often manifests when an individual is involved in a continuous task requiring they focus their attention on some changing scene. During this continuous observation an unexpected event occurs, clearly visible to those not focusing on the task, but usually invisible to the individual engaged in observing. This effect seems to occur even if the unexpected event happens near or at the point the individual is focusing on. Studies described below demonstrate that the features of a scene are not even perceived without the individual actively focusing on them.[2]

Mack and Rock [20] presented a study in which participants were told to watch a fixed point on a screen on which a cross would appear for several hundred milliseconds and to determine which of the two arms of the cross were longer. After several of these trials, an unexpected object–usually a simple geometric shape–would appear along with the cross but slightly off to one side, and participants were asked if they noticed anything other than the cross on the screen. Of those tested, 25% failed to report the unexpected object on the screen. This led Mack and Rock to the idea that if there is no attention, there is no perception. A followup experiment was conducted in an attempt to eliminate inattentional blindness. A change was made during the trail where the unexpected event occurred: The unexpected object appeared at the fixation point while the cross appeared off to the side. However, during these modified versions of the experiment, inattentional blindness rose and 60–80% of participants reported nothing other than the cross appearing on the screen. The shape was replaced by the participant's name and inattentional blindness dropped. If the name was replaced by a word or someone else's name, the effect would return. The effect would also be present if the person's name was misspelled (e.g. "Jack" spelled as "Jeck"). It appeared that the meaning of the unexpected object may be perceived, but not the object itself.

A concept known as *selective looking* was proposed by Neisser and Becklen in [21]. Participants were shown two videos superimposed on one another. The first was of a hand-slapping game played by two people and the second was of three men passing a basketball back and forth between one another. Ten trials were conducted, beginning with only one of the two videos being shown and later with the videos superimposed on one another. Participants were asked to pay attention to one of the two videos

[2] See `http://www.youtube.com/watch?v=Ahg6qcgoay4` for an example of this effect.

and make reports on what was seen. However, during certain trials unexpected events occurred. For example, if the participant was told to pay attention to the hand-slapping game, the basketball tossers might all leave the frame and be replaced by women. Of 24 participants, only 1 to 3 would catch the unexpected events during the trials in which they occurred. Overall, only half of participants noticed any of the unexpected events that occurred in the videos they were not attending.

Neisser and Becklen's work was extended in [31] by Simons and Chabris who created a series of four videos in which two teams of three players passed a basketball to members of their team. During the video, an unexpected event occurred: either a woman with an umbrella walked through the scene, or a person wearing a gorilla costume walked through. Videos were also of two types: a superimposed version[3] as used by Neisser and Becklen, or a "normal" version in which all elements were filmed at once. Participants were shown one of the videos and told to count the number of passes made by a single team (dressed in white or in black). Participants would then provide the number of passes they had counted and asked a series of increasingly specific questions, from whether anything unusual happened, to did a woman (gorilla) walk through the scene. Only 54% of 192 participants noticed the unexpected event. Overall, the woman with the umbrella was noticed more often than the gorilla; however, if the participant was tasked with monitoring the passes of the team dressed in black, they were much more likely to notice the gorilla. Simons and Chabris suggest that this was due to the gorilla being black in colour, and hence sharing a characteristic with the target group, making it more likely that a participant would focus their attention on the gorilla at some point during the video.

These studies, combined with those from change blindness, show a remarkable lack of ability in individuals to capture various elements of what they see. Although we may feel that we are continuously "seeing" as we look about our world, it appears that precious little remains in the mind. Without being properly known about and compensated for, these blindness effects may have significant impact on various aspects of software testing.

2.3 Anchoring and Adjustment Effect

Let us say a person is given a value (the anchor) which is typically numeric, and then asked to answer a question which has a numeric answer. The person's answer will be influenced by the anchor

[3]This was achieved by filming the three elements (each team and the unexpected event) separately and then superimposing each together.

that was provided (the adjustment), even if the anchor was randomly generated, completely arbitrary or unrelated to the question. Studies done into the anchoring and adjustment effect suggest two forms: (a) judgement anchoring, in which a person's judgements of quantities and other numerical values is skewed; and (b) valuation anchoring, in which the value a person may assign to an item is affected.

2.3.1 Judgement Anchoring

The anchoring and adjustment effect was first described by Tversky and Kahneman [36] with a pair of experiments. In the first, participants were recruited to estimate several quantities. Before the participant was asked to answer, the experimenter would spin a wheel with numbers from 0 to 100, allowing a random number to be generated in full view of the participant. The participant would then answer if the quantity that was asked for was above or below this anchor value and then were asked for the value itself. It was found that this initial anchor had a significant influence on the answers of the participants, even though the value itself was purely random. For example, when asked to estimate the number of African countries in the United Nations, the participants that received an anchor of 10 gave a median answer of 25; however, the participants that received an anchor of 65 gave a median answer of 45.

Tversky and Kahneman also found that calculations were impacted by this effect. Long calculations require intermediate steps; however, if not enough time is provided to complete the calculation, the intermediate steps may act as an anchor themselves. This was shown when groups were asked to calculate the following:

$$1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8$$

and

$$8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

The calculation is too long to complete in a single step, thus necessitating the use of intermediate steps. Participants were only given 5 seconds to answer, preventing a complete calculation. Some estimatation was necessary to answer the question and the intermediate steps acted as an anchor. It was found that participants given the ascending product reported a median estimate of 512, while the participants with the descending product reported a median estimate of 2250. The answer to the product is 40,320.

These two studies show the heavy influence of anchors and a participant's dependence upon them.

Although the anchors provided in the first experiment were generated randomly, in full view of the participant, they were still influential when the participant was asked to provide an answer. In the second experiment, the order of the product changed the intermediate steps, which acted to change the (perceived) growth of the product, leading to a large variation in the estimates given by the two groups of participants.

2.3.2 Valuation Anchoring

Extending the initial findings from above, Ariely, Loewenstein, and Prelec [3] described an auction conducted with students from MIT's Sloan School of Management. Participants were presented with a series of items, including: a cordless keyboard, a trackball mouse, two different bottles of wine, a book on design, and a box of fancy chocolates. Participants were then asked to write the last two digits of their social security numbers down on a page and to indicate if they would pay that number, in dollars, for each of the items. Following this, participants were asked to indicate what would be the maximum amount they would bid for each of the items. It was discovered that participants with social security numbers above the median offered maximum bids 57–107% higher than those below the median. Additionally, those participants in the top quintile (i.e. those with the last two digits of their social security numbers between 80–99) were found to have valuations three times higher than those in the bottom quintile. Interestingly, Ariely [2] pointed out that participants in this study logically ordered their valuations for the similar products. The cordless keyboard was valued higher than the trackball mouse, and the higher-rated wine was valued above the average-rated wine.[4] Therefore, although a participant's valuations made sense in relation to real-world pricing schemes, they were nonetheless highly influenced by the presence of the anchors provided.

2.4 Peak-End Effect

The ability to judge past experiences is a necessary skill for human survival. Without it, we could not make judgements about future actions based on past experiences. Did the berries make me sick in the past? I better avoid them if they did. However, are our experiences remembered in the way they

[4] When the items were presented to participants, detailed descriptions were provided that included industry reviews of the two wines.

actually happened or do our memories of an experience differ from the experience itself?

We require some sort of mental algorithm–some "experience calculus"–to help us form judgements of past experiences. A simple enough rule exists in *temporal monotonicity*, described by Fredrickson and Kahneman in [9], which states that if a moment of pleasantness (unpleasantness) is added to an experience, the overall memory of the experience is more pleasant (unpleasant). This thesis can be used as an example. You are currently reading and enjoying the interesting ideas being presented in beautiful and elegant prose. However, if a moment of unpleasantness is added (perhaps a pretentious and self-congratulatory statement is inserted by the author to inflate his ego) your overall impression of the work is decreased and, by extension, your memory of the experience is now lessened. Fortunately, research into how we remember experiences has shown that humans do not do this type of mental tally when judging an experience from memory. We can judge experiences just fine "in the moment" but when asked to recall those experiences from memory we make a mess of things. We stop taking duration into account (*duration neglect*), and we often prefer situations that include higher overall unpleasantness (violating temporal monotonicity) due to our memory of the experience being based on just two specific moments: the peak and end moments of the experience (*peak-end effect*).

The peak-end effect (or sometimes the *peak-and-end rule*) states that humans remember experiences based on just two moments: the peak moment of intensity, and the final moment of the experience, and from these two moments the individual will remember if the experience was a pleasant or unpleasant one [9]. The duration of the pleasant or unpleasant portions of the experience has virtually no impact on our memories. Negative experiences are remembered more favourably if the moment of peak unpleasantness occurs before the end of the experience. This is in preference to experiences that end with the peak moment of intensity. Essentially, we are averaging the two moments. The experience is preferred if the average of the moments is lower than the peak value. In this thesis, the peak-end effect will primarily be examined in the context of unpleasant experiences being recalled from memory, although the effect is also present in pleasant experiences.

2.4.1 Cold Water Immersion Trials

One of the most illustrative early studies was described by Kahneman, Fredrickson, Schreiber, and Redelmeir in [14]. Subjects were asked to undergo the unpleasant experience of immersing their hands in cold water. Each subject put one hand in a tub of 14 °C water for 60 seconds and removed it. The

subject repeated this with their other hand; however, after the initial 60 seconds the water temperature was raised to 15 °C and the subject kept their hand submerged for an additional 30 seconds. These were referred to as the *short* and *long* trials, respectively. Both trails were painful due to the temperature of the water. The long trail was less painful at the end, but contained just as much pain as the short trail.[5] After both trials were complete and the subjects had some time to rest, they were asked about the trials. Subjects were told that a third trial had to be completed and asked which of the two previous trials they would like to repeat. Of the 32 subjects participating, 22 of them selected the long trial for repetition. Even though the long trial had more pain overall, each subject's memory of that experience was favourable to that of the short trial, violating the rule of temporal monotonicity. If the mind sought to minimize pain, each subject should have chosen the short trial to repeat. Subjects made additional mistakes in their recollection of the two experiences: a majority of subjects claimed that the longer trial was easier to handle, less painful at the peak moment of intensity, and even less painful overall. Duration neglect appeared to play a large factor in judging the experiences of the two trials as nine subjects believed the trials were of the same length and six thought the long trial was shorter in length.

2.4.2 Colonoscopy Trials

Trials were done outside of laboratory conditions to see if the effect was present in other types of experiences, namely in hospital patients undergoing colonoscopies. Three specific studies are particularly noteworthy. The first by Redelmeier and Kahneman [24], showed that the peak-end effect held in real-world patient populations, i.e. that subjects neglected the duration of painful moments and relied on the peak and end moments when recalling an experience. The more interesting results were described in by Kahneman, Wakker and Sarin in [15], and Redelmeier, Katz, and Kahneman in [25]. The goal of these studies was to attempt to improve a patient's memory of the colonoscopy procedure. Almost 700 participants were involved, half of whom were randomly assigned a modified version of the procedure in which the scope was left in for a short period of time after the completion of the actual colonoscopy. This was not as painful as the procedure itself, but was more painful than having the scope removed. The idea was to add a moment of lessened intensity, prolonging the experience and total pain, but generating an end moment of much less pain than would have otherwise been experienced. As the peak-end effect would predict, this lower intensity end moment caused the overall procedure to be

[5] The long trial actually contained more pain as the short trial due to the extended length of the trial.

remembered more favourably. A 10% quantitative decrease in remembered pain was reported by the subjects.

2.4.3 Why Peaks and Ends?

Peak and end moments of experiences seem to carry tremendous importance to the brain if they are the primary determinants of how experiences are recalled. How can just two moments determine our memories of an experience that could be minutes, hours, or days long? One would hope that everything in between meant something and counted towards some overall memory of our lives. Sadly, it seems that the brain cares little for duration when it comes to recalling experiences.

Kahneman explains this in [13] as being a conflict between two versions of ourselves. The first, the *experiencing self*, is the in-the-moment version of us that experiences everything. Every moment of joy and sadness, every time we fall in love or lose someone dear to us, the experiencing self goes through all of it. But, the experiencing self has little to do with the memory of those experiences. The *remembering self* only cares about peaks and ends and it is this version of ourselves that controls our memories. Further, Kahneman describes this in terms of stories. In a story, the most important part is the ending. A good ending makes a story, while a bad one can ruin a story. We apply this to people experiencing a painful medical procedure and the peak-end effect makes sense. If a person is having an experience that ends with the peak moment of intensity, it would be like a story that ended in terrible pain and sadness. This would make for a terrible story and not one any person would like to repeat anytime soon. However, if the peak moment occurs earlier for the person we have a story in which terrible pain has been overcome, the person survived, and was able to continue on. This makes for a much better story that is worth repeating.

In addition to this, Barbara Fredrickson makes the case that these two specific moments "carry more personal meaning than other moments" [8]. Peaks show us the limits of the experience. If we handled the peak moment of experience, then we handled the entire experience. By extension, peaks of extreme moments show us our own limits, perhaps limits we never knew we had, or that we are capable of handling more than we ever believed possible. Therefore, we need peaks to help figure out if we are up to the task of repeating a particular experience. Ends remove fear of the unknown from an experience. Once we have reached the end, there is nothing more to fear, nothing more will happen. They also allow us to determine exactly when the peak moment occurred. We know that we have survived the

experience, we have lived to tell the tale, and we may reflect on the experience as a whole, something that is impossible when you are still in the thick of it.

2.5 Expectancy Effects

In Chapter 1, the concept of *expectancy effects* was introduced as the root cause of why people hear diabolical lyrics in innocuous rock music. Bacon recognized, although did not name, these effects in his descriptions for the Tribe and the Cave (see Section 1.2). These effects can have a significant impact on experimental procedures and the data collected thereby, and these same effects can be extended outside of the laboratory and affect how other individuals with expectations (e.g. testers) do their jobs.

When conducting research, the experimenter herself is a prime source of possible errors. There may be problems during subject observation, such as recording errors. Aspects such as the age, race, or gender of the experimenter can bias how the experiment is conducted [27]. The experience level of the experimenter, their anxiety, their mood, etc. can all impact how the experiment is run and how data is gathered. The interpretation of that data may be debated, with different styles of interpretation possibly leading to contradicting conclusions. An experimenter may already have a conclusion in mind that they would like to draw from their data set and consciously manipulate, or drop altogether, data that go against their goals. Two categories of expectancy effects are of primary concern: (a) experimenter expectancy effects; and (b) subject expectancy effects.

2.5.1 Experimenter Expectancy

Mentioned briefly above was the possible conscious influence of the experimenter's hypothesis, but this can also appear unconsciously as the *experimenter-expectancy effect*, otherwise known as the *observer-expectancy effect*. In essence, the experimenter unconsciously influences the participants in their experiment, or the experiment itself. As an experiment is designed, various controls are decided upon in order to conduct a valid test of the hypothesis. If the controls are too lax, the effect or result may not manifest. If the controls are too strict, it may bias the experiment to ensure the hypothesis is correct, possibly calling into question, or even invalidating, the result. These may manifest as probabilities that would otherwise not occur, answers that subjects would otherwise not give, or events to be chosen that would otherwise not be selected. Biases in these control and design decisions may not even be evident

to the experimenter and it is precisely these unconscious biases which may drastically alter the results.

A classic example that is often citied is that of Clever Hans, the horse what seemed able to count, add, subtract, and perform other simple arithmetic calculations by tapping his feet, described by Rosenthal in [27]. Hans' owner claimed that he was in no way signalling the animal or providing it with the answer, going so far as to allow the animal to be tested without him present. However, it was discovered that Hans was unknowingly being signalled. It was found that he was unable to answer questions when he could not see the questioner. This led to the discovery that the questioner themselves would unconsciously signal Hans, often with slight inclinations of the head. Once a question was asked, the questioner would incline their head slightly, unknown to them, which would signal Hans to start tapping his foot. When the appropriate number had been counted out, the questioner would decline their head, again unknown to them, and Hans would stop tapping. This was further proven when a questioner simply inclined their head and Hans started tapping, but no question had been answered. Rosenthal claims this revealed a self-fulfilling prophecy, as those individuals that played the role of questioner expected Hans to answer correctly, and with this belief, would unconsciously signal the horse to perform the correct action.

The notion of the self-fulfilling prophecy is further described in [26] by Rosenthal with the Pygmalion Experiment. In essence, students who were expected to achieve higher grades by their teacher achieved them. A possible explanation of this was again given by Rosenthal, who co-conducted the study: Teachers that expected high performance from a student may have biased or modified their own actions in instructing the student in such a way as to encourage the student to perform at a higher level than the student would have otherwise performed.

2.5.2 Subject Expectancy

Studies have described that a participants's motivation in joining a study may be because they believed they were helping to improve the welfare of humanity, advancing the progress of science, or aiding the experimenter directly (see a summary by Rosenthal and Rosnow in [28]). Other studies speculated that participants will try to complete whatever task is given to them as that is what is expected of them. The experimenter may represent an authoritative figure, perhaps with a higher status, and thus must be obeyed. Finally, a participant may feel that the experiment evaluates some aspect of them personally and hope to perform well or succeed according to this evaluation. No matter which of

these possibilities are correct, this motivation may lead to a form of expectancy known as the *subject-expectancy effect*. A participant may become aware of what hypothesis the experimenter is attempting to test and change their behaviour, knowingly or not, in order to achieve that result. This, they believe, is what the experimenter would like or what would be best "for science". Although their actions are altruistic, the result of those actions may invalidate their own results, or, in the very least, skew the experimenter's data away from what would have been the true result.

Chapter 3

The Software Tester

QA didn't break anything, it was broken when we got it.

—UNKNOWN

Software testing serves as an integral component of the software development process. The field is wide-ranging with many different approaches and techniques, each suited for particular situations, projects, and companies. The following chapter provides (a) an introduction to the field and the information necessary to understand the remainder of this thesis; (b) a discussion of the design of "good" test cases; and (c) an overview of previous work linking psychology and software development and testing.

3.1 A Gentle Introduction to Software Testing

By no means is this section comprehensive–certain topics are described with depth, others with only a few sentences, others still are completely omitted–as a comprehensive guide to software testing would be unnecessary here. This section is based on a combination of my own knowledge, acquired through coursework and work experience, and Kaner, Falk and Nguyen's excellent book *Testing Computer Software* [18].

3.1.1 Complete Testing is Impossible

To define exactly what testing is, one must first know what testing is not. The goal of testing is not to test a software system completely. This is an impossible task and any person claiming to be able to do it is grossly uninformed at best, a lair at worst. As evidence, consider a simple calculator program that does four operations on pairs of numbers: addition, subtraction, multiplication, and division. To prove the correctness of this program, a tester would first have to test all valid inputs. If we assume the program takes only single-digit, positive or negative integers, a tester would need to run 171 ($\binom{19}{2} = 171$) test cases per operation. The order in which inputs are entered matters, thus an identical set of test cases will be needed for inputs with the numbers reversed, per operation, for a total of 1368 ($171 \times 2 \times 4 = 1368$) test cases overall. Once all those tests have been run, the tester would need to test all invalid inputs, which would require the tester to run all inputs that the program does not allow (i.e. all numbers that are not single-digit numbers), and to do this for all four operations. Once all those tests have been run, the tester would need to test all edited inputs. The tester would have to enter a number, erase the input, enter another number, and continue as usual. This would have to be done for all valid and invalid inputs, all operations, and for both the first and second numbers the program uses to calculate. Finally, once all those tests have been run, the tester would need to enter all state-based inputs. Most programs have some type of internal "state" representing the current status of program execution. In the example above, a tester would need to run all the test cases already generated but after the program had completed a calculation. Then the tester would need to repeat all the test cases once the program had completed two operations, then three operations, then four, five, etc.

Imagine the program is now ATM software, comprised of menus and various operations. A user would insert their card and enter their pin number before being presented with a menu listing all the operations that can be performed. Once an operation is selected, more inputs are required, usually an amount of money the user wishes to manipulate in some fashion. After the operation is completed, the user is returned to the menu. This program has a much more complicated series of states. Many different operations can be executed at many different times during use, requiring many different inputs. If a tester wishes to completely test this software, all paths through the system, regardless of length, would need to be executed, for all possible inputs.

Finally, once a tester has tested all possible paths with every possible input, the software can be

said to be fully tested, according to its specification. Still the software is not completely tested, as the specification itself may contain errors. The errors may not be obvious but may simply be design flaws, such as poorly laid out user interfaces. For testing to be complete, a tester would also need to be fully aware of all design flaws in the system, which is not possible.

All these factors prevent software from ever being fully tested. It is an unachievable goal, a software utopia that is nothing but dreams. This brings us to what testing is. The goal of testing is not to prove software works, but to prove that it does not. A tester's mission in life is to attack the software from all sides, prodding, poking, kicking and punching, so at the end of the day she is able to say: "See? It still works". Kaner, Falk, and Nguyen mentioned Karl Popper's *Conjectures and Refutations* [23], a book on the philosophy of scientific knowledge. In it, Popper presented a series of conclusions regarding the proof of theories, including: "Every genuine *test* of a theory is an attempt to falsify it, or to refute it". Replacing "a theory" with "software" and we see a tester's goal should not be proof of correctness, but lack of proof of incorrectness.

3.1.2 Types of Errors

Before proceeding, a note on terminology is warranted. In the software testing field, various terms are used to denote problems in the software. The two most commonly seen are *bug* and *error*, but *fault* and *failure* are also used. Amongst some, these terms have different meanings, often with only subtle differences. For the purposes of this thesis, I will restrict my own usage to *bug* to mean any type of problem with a piece of software that may be experienced by a user. This section, however, will also use *error*, to be consistent with the usage in [18].

In their book, Kaner, Falk, and Nguyen [18] present a list of software error types. Other systems of classifications exist, such as Beizer's [5] extremely detailed classification system and taxonomy of possible software errors. However, the Kaner, Falk, and Nguyen list is short, fairly comprehensive, and adequate for the purposes of this work. A summary is presented below.

User Interface Errors These can include: the program simply not doing what it is supposed to, unclear commands and navigation within the program, poor program performance, and problems in displaying output.

Error Handling The need to catch errors gracefully is necessary in any software program;

however, unless the error handling code itself is throughly tested before deployment, it may become a source of bugs.

Calculation Errors Programs that perform large amounts of floating point calculations are susceptible to these errors. Large floating point calculations may require rounding or truncation as they are computed. If the calculation is complex enough, these roundings and truncations may compound such that the final answer given is not the correct one, while all intermediate calculations are still valid and correct.

Initial States When a program is run for the first time it may need to initialize all of its variables. If this task is not completed properly the program may behave erratically. However, subsequent runnings may behave normally, making these bugs particularly difficult to find as they may only appear once and never again while the program is running. These bugs may then reappear whenever the software is restarted.

Flow Errors These errors occur when the program calls (executes) the wrong next step, often due to poorly designed control flow logic.

Errors in Handling Data As data are passed between segments of a program, or between different programs, it may become corrupted or incorrectly modified in such a way that when the next segment or program receives it, the data are not what was expected and cause the program to behave incorrectly.

Race Conditions These bugs occur when one event was expected to occur before another. If the actual sequence of events differs from the expected sequence, this may cause errors. These types of bugs are problematic to test for as it is difficult to replicate the exact timings of events in a test environment.

Load Conditions Stressing a program is a common source of errors. If programs run for a very long period of time or at peak performance levels, resources–such as memory or hard drive space–become depleted. Once a resource is no longer available to a program that expects said resource, the program may behave erratically.

Hardware The software/hardware boundary can also be a source of bugs. If hardware is

unavailable or heavily loaded, it may not behave properly, just as software. It would then send messages (or not) to software that is not (or is) expecting them, leading to problems.

Source and Version Control As software is being developed, it is entirely possible that several versions of the source code are in use at any one time. If programmers use outdated versions of source while developing the software, this may reintroduce bugs from the past into the current version of the software.

Documentation If the documentation for a piece of software is incorrect, a tester or user may believe that the software is not performing as it should. This can waste valuable time during the development process and can cause technical support headaches once the software is deployed.

Testing Errors Falsely identifying something as a bug or missing obvious bugs is a concern for every tester. Bugs that are not actually bugs waste the time of the developers in tracking them down. Missing something is worse as bugs become increasingly more expensive to fix the later they are discovered in the development cycle.

3.1.3 Levels of Testing

Testers do their work at several levels, each with a specific goal and focus. Presented below is the common five-level testing model. Others may add levels, remove certain levels I have included, or both. Again, this is by no means a comprehensive summary of all possible levels of software testing.

Unit Testing A software system is built of numerous smaller components interacting with one another to give the user a seamless software experience. These components are sometimes called *modules* or *units*. The goal of unit testing is to test each unit in isolation from others in order to make sure it functions properly. This is work typically done by the programmer who wrote the unit and not the testing team proper. Units will often interact with other units once the software is complete, either providing some form of data to other units, requesting data from other units, or manipulating data for other units. During the unit testing stage, not all these units may have been written yet and thus *scaffolding* is required. Scaffolding consists of code that a programmer is required to write in order to fake the presence

of other units to the unit he is testing. A *harness* is code that tells the unit under test to do something and may provide it with data (it calls the unit under test), while a *stub* is code that provides the unit with dummy data if it requires it during normal operation.

Integration Testing Once the units have been written and individually tested, each must be integrated into the larger software system. During this process, integration testing is performed on multiple units in order to test that each works correctly when interfacing with other units. There are several strategies for integrating units together:

Top-down integration begins with the most visible component of the software system (e.g. a menu) and moves down to smaller, more specialized units. Using the example of the ATM software, we may begin with the main menu of the system and then integrate the units that carry out account transfers, receipt printing, account balancing checking, etc. (all the actions from the "first level" of the menu). Once the account transfers unit is integrated, the units concerning deposits and withdrawals are integrated (the "second level" of the menu), followed by the unit that asks the user how much to deposit, etc. This type of integration allows early versions of the software to be shown to clients/users, as the most visible elements are completed first. The creation of stubs is often necessary for this type of integration.

Bottom-up integration does the opposite of top-down by beginning with the most abstract and specialized elements and working towards integrating them into menus as the last step. As a result, clients/users are unable to see the system in any useful form until all integration is completed. The creation of harnesses is necessary for this type of integration.

Thread integration is done by first integrating all the logical units of a single software action together. In the ATM example, the thread concerning deposits may be thread integrated. Beginning from the menu, the deposit unit is integrated first, followed by the unit that requests the amount to deposit, followed by the unit that opens the ATM's deposit slot, etc. All other actions–such as withdrawals or checking account balances–are ignored until the deposit thread is complete. Although the example just given was a top-down integration, bottom-up thread integration is also possible. Thread integration has the added benefit of allowing parts of the system to be deployed before the entire system is complete.[1]

[1] Although deploying ATM software that can only accept deposits would not be particularly useful to most customers.

System Testing After all units are integrated together, system testing is performed. This may often be what people think of when they hear "software testing". The goal of system testing is to test the entire system as a whole. In addition, the security and performance of the system is also evaluated at this point.

Acceptance Testing Almost identical to system testing, acceptance testing is done in the actual environment the software will be deployed in. This phase of the testing process may also include the clients/users of the system to ensure all expectations are met prior to system deployment.

Regression Testing Software is rarely developed, deployed, and never touched again, as patches, fixes, improvements, and new features are often developed after the system has already been launched. Once these types of changes have been made, regression testing is performed to make sure that anything that has been added to the source code has not introduced new bugs into existing parts of the system.

There are two main approaches to software testing. *White box testing* refers to a style of testing where the tester has in-depth knowledge of the way in which the software was developed, typically by having access to the source code itself. The second approach is known as *black box testing*, in which the tester has no access to the source and no real knowledge of how the software was built. This is the type of testing that this thesis is mainly concerned with. A third form known as *grey box testing* also exists but is usually not considered its own approach. In grey box testing, the tester does not have access to the source code, but may be familiar with programming and the common techniques used therein, and thus develop test cases outside of the realm of black box testing.

In addition, testing can be classified as either being *dynamic* or *static*. Dynamic testing requires the software to be running in order to perform, whereas static testing is done without executing the code. Static tests include walkthroughs of the source code, complexity analysis, syntax checking, etc. All static forms of testing fall under the label of white box testing, as each requires access to the source code of the software. When "testing" is discussed in subsequent chapters it will refer exclusively to black box testing.

3.1.4 White Box Testing

White box tests are created by the tester examining the source code of the software to test possible points of failure, the goal of which is to *cover* the code and to check for certain types of errors. White box testing is a very large field unto itself and beyond the scope of this thesis. A short overview of two common techniques is given below.

Code Coverage

The goal behind code coverage is to test various aspects of the source code to check for the presence of bugs. Various forms of coverage exist, some more stringent than others [5].

Path Coverage Test each logical path through the software and check each for errors. As was discussed earlier, achieving complete path coverage is not feasible due to the resources necessary for test every possible path. Complex states and loops in the code make this form of coverage an impossible and unrealistic goal.

Statement Coverage Cover each statement in the code as part of some test to ensure each is executed properly. Executing each statement just once is not a very thorough test of reliability as many statements often branch into different paths. This makes statement coverage a very weak form of coverage.

Branch Coverage Similar to statement coverage, but each branch statement is checked twice, once when it is evaluated as TRUE and once when it is evaluated FALSE. Imagine you are a tester and the following pseudocode appears in software you are testing:

```
IF (x > 5 OR y < 0 OR z != 2) THEN...
```

Branch coverage would only require this decision statement to be tested when it evaluated to TRUE and FALSE. A test that evaluates this statement to TRUE would be to assign `x > 5`, `y < 0`, and `z != 2`; and a test that evaluates to FALSE would be to assign `x <= 5`, `y >= 0`, `z = 2`. However, this leaves six possible combinations untested:

```
x > 5, y < 0, z == 2
x > 5, y >= 0, z != 2
x > 5, y >= 0, z == 2
x <= 5, y < 0, z != 2
x <= 5, y >= 0, z == 2
x <= 5, y >= 0, z != 2
```

Due to this problem, three more versions of branch coverage exist, each increasingly more stringent. *Condition coverage* tests that each condition is evaluated to TRUE once and FALSE once. In the example above, condition coverage can be achieved in two tests: one with all TRUE evaluations, one with all FALSE evaluations. *Decision/condition coverage* tests each condition to be evaluated as TRUE and again when evaluated as FALSE, and the overall decision statement to be TRUE and again to be FALSE. In the example above, this can again be achieved in just the two tests. Finally, *Multiple condition coverage* requires a test for every combination of TRUE and FALSE evaluations to each condition. If multiple condition coverage were applied to the above example, eight tests would be required, ensuring coverage of all possibilities.

Data-Flow Testing

The manipulation of data in some form is the primary purpose of most software, sometimes making up over half of all statements in the system [5]. Data-flow testing focuses on data and how it is moved and manipulated throughout a software system. Typically in data-flow testing flowgraphs are generated to summarize the movement of a piece of data through the system and certain points are identified in the flow, such as when a data object is defined, destroyed, used in a calculation or a predicate. Certain combinations of actions are searched for, which are typically indicative of bugs. Some examples include:

Destroy-use A data object is destroyed and then used in a calculation or predicate. This is a bug as the data no longer exists and can cause erratic behaviour in the software.

Define-destroy A data object is defined and destroyed but never used for anything. This

may indicate that the data object is a leftover from some previous incarnation of the software and should be eliminated.

Define-define Languages allow data objects to have their values change during execution; however, define-define situations may also indicate the presence of leftover code from a previous version.

Null-destroy Data objects can only be destroyed if they exist. If the software is attempting to destroy a data object that never existed, this may indicate logical problems in the source code.

3.1.5 Black Box Testing

In black box testing, the tester must treat the software as a box that they cannot see into. The tester will provide the software with various forms of input and check the given output against what was expected. If the outputs match, the software is performing as specified. The test cases that are often generated fall into one of two categories: *valid* tests and *invalid* tests. A valid test is one where the input given is something the software should accept and generate a proper output for. If we reuse the calculator example from the beginning of this chapter, a valid test would be $1 + 1$. The program should return 2, and if so, the test case passes. An invalid test case is where the given input is not something the software would accept, such as $10 + 10$, as these are two-digit numbers. The software should inform the user that this input is not valid and cannot be used by the software. Although the software failed to make the calculation, it gracefully handled the invalid input and so this test case passes as well. In black box testing, it is common to use a combination of valid and invalid test cases.

Equivalence Partitioning

Complete testing is impossible for any but the most trivial software systems due to the vast–possibly infinite–number of test cases necessary. To cut down on the number of test cases needed, a tester may use *equivalence partitioning*. All possible inputs that a piece of software can accept is referred to as the *input space*. The goal of equivalence partitioning is to divide the input space into categories, called *equivalence classes*, where all members of that class are essentially considered equivalent to one another. Following the simple calculator example, all positive single-digit numbers could be put into the same

equivalence class. Negative single-digit numbers could make up a second equivalence class. A tester would then create valid and invalid test cases for each class. A valid test case for the all positive single-digit numbers class would be the number 5, while an invalid test could be the number 17. Similar types of tests would be created for each of the other equivalence classes.

Tests for equivalence partitioning typically follow a pattern when being written. For a class that covers a range of values three tests are created: one valid test within the range and two invalid tests outside the range, with one below and one above the limits of the range. For sets of values, two tests are created: one valid test using a value in the set and one invalid test using a value not in the set. For conditions where a value must be of a particular type or format, two tests are created: one valid test that matches the expected type, one invalid test that does not.

Boundary Value Analysis

A variation on equivalence partitioning, *boundary value analysis* focuses on the "edges and corners"–the boundaries–of the program. Boundaries are often said to be one of the most common places for bugs to exist and are of particular interest for testers. In boundary value analysis, the input space is partitioned into equivalence classes as above; however, the values used are those on the boundaries. For example, to test the all positive single-digit number class in the simple calculator example, four tests–two valid and two invalid–would be written. The first valid test would be for the number 0 as it is the smallest of that class and therefore sits on the boundary. The other valid test would be for the number 9 as it is the largest of the class. Invalid tests would be for just outside these two values, namely for the numbers -1 and 10.[2] Similar groups of tests would then be generated for each of the other equivalence classes.

Boundary value analysis not only focuses on inputs but outputs as well. The output space of the program is also partitioned into equivalence classes, and tests are written in such a way as to generate output that is within the class but at the boundaries and output that is just outside the class. This requires a bit of reverse engineering and may be tricky; however, this additional work can often reveal bugs on the edges of outputs that would be difficult to catch otherwise.

[2]The test for -1 is really a valid test as negative numbers are allowed, but it is considered invalid as our focus is on positive numbers at the moment.

Specification Testing

In *specification testing*, a tester compares the software system against all the various documentation that has been generated to ensure that the program meets expectations. These documents often include design specifications, requirements descriptions, manuals, and interface designs. Not only is the functionality of the software being addressed, but also whether it meets quality standards, safety regulations, and any contractual obligations that have been set.

3.2 Thoughts on Test Case Design

In the previous section, a small sample of white and black box testing techniques was presented. These techniques demonstrated some of the ways in which software testers decide how to test software. One problem that was not addressed was how to take those techniques and generate an actual set of test cases for use. The following section presents some advice on the practical creation of good test cases. Like many other sections of this thesis this is by no means comprehensive, but aims to provide good general advice.

3.2.1 What is a Test Case?

From my own experience, a test case can be defined as a documented tool for gathering information about a software system. In [16], Kamde, Nandavadekar, and Pawar outline three factors in test case design: (a) the data components that make up the test case; (b) test case quality factors; and (c) the format of the test case. Documentation also can play a tremendously important role and should be included as well.

3.2.2 Factors in Test Case Design

Data Components

The data components of a test case are determined by using the techniques previously described, namely, white box and black box testing. These techniques will generate what will be used as the inputs during testing. For example, the application of boundary value analysis will reveal the various

boundaries for a section of the software and the inputs that must be tested. These inputs, known as *data components* will then be used to provide the foundation of the test case writing process.

Test Case Quality

Before test cases can be written, a tester must first decide which test cases she will actually use. Several helpful guidelines are provided by Kaner [17] to aid the tester in determining which test cases should be written. Firstly, a test case must (a) find defects and maximize the likelihood and number of bugs found in high-risk or safety-critical areas; (b) provide management with information necessary regarding product maturity, in order to decide if and when the software should be shipped; (c) determine whether the software meets specifications and standards regulations; and (d) assess and insure the quality of the software.

The way this information is provided is important as well, and some test cases will be better at delivering it than others. If there are ever test cases that may overlap in what they test, a tester should always choose the one that is (a) more likely to expose a bug; (b) more likely to expose a significant problem; (c) more representative of the potential actions of the a user; (d) easier to repeat for troubleshooting purposes; and (e) simpler. Test cases should also be independent. If tests are tightly coupled and dependent on one another it can make the testing process longer and more complex. If a test case were to fail, all test cases dependent upon it could not be run, wasting the time and effort of the tester.

Formats

Once the test cases themselves have been decided upon, they must be written up to be of any use. Three common formats exist: (a) sequential; (b) matrix-based; and (c) automated. *Sequential*–or *step-by-step*–test cases follow a "Step 1, Step 2, ..., Step n" style. The test cases used in my experiment (see Chapter 4) follow this format and can be seen in Appendix B. Sequential test cases are excellent for testing GUI-intensive software as they allow for exact descriptions of what it is the tester should be doing, clicking, scrolling, or entering, and for more nuanced evaluation of the software. Style is very important in sequential test cases due to the amount of written descriptions. The language used should be simple and clear, allowing testers to focus on the testing, and not on decoding what the test cases are trying to say. The length of the test case is also important. A very long test case may cause testers to get lost or confused. Very short test cases may not be worth the time necessary to write them in the

Step #	Input 1	Operation	Input 2	Output
1	1	+	1	2
2	2	-	2	0
3	1	$\times$	3	3
4	4	/	2	2
5	2	+	8	10
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3.1: Sample matrix-based test cases for a simple calculator program

proper format. A delicate balance must be struck, with some suggesting a length of 8 to 16 steps as the ideal [16].

Matrix-based test cases are presented as a table, and are particularly useful for running large black box input/output tests. They have been praised by Yamaura [37] for their ability to catch errors in these scenarios. Each row is its own test and provides the tester with inputs and expected outputs. A tester will work her way down the table, inputting each of the inputs provided and checking the output against what was expected. Bugs are logged if the software output differs from what was provided in the matrix. Table 3.1 provides a sample of a matrix-based test case series for the simple, single-digit calculator program described previously.

The final format a test case can take is *automated*, in which suites of test cases are run by a computer to check the software under test. These test cases are less written and more programmed, thus there is little concern for style and presentation, length, or language, as no human will be reading them. Automated test cases can be executed very quickly and in parallel with other testing activities; however, they are often expensive to create, either in monetary cost or in man-hours. Depending on the software being tested, automated test cases can also be extremely difficult to write. Finally, when a suite of automated test cases is run and a bug is detected, subsequent tests may fail due to that initial bug.[3] As a result, those subsequent tests may not even run, or be worthless if they do.

Documentation

In [37], Yamaura presented a compelling argument for thorough documentation of test cases, a position not exactly shared by Kaner, or Kamde, Nandavadekar, and Pawar. While Kaner, or Kamde, Nandavadekar, and Pawar both recommend documentation, Kaner especially makes the case that testers

[3]This is sometimes referred to as a *blocking bug*.

should be free to try ideas on the fly and follow gut feelings, rather than slavishly execute a series of test cases provided to them. However, fully documented test cases allow (a) someone else to run the test cases for you; (b) for checking the status of the current testing cycle; (c) analyses to be run using past test cycle data; and (d) the test cases to be used as templates for new test cases, or reused for similar products. Documentation should also be complete in regards to extra information for the tester. For example, test cases should indicate if the input values the tester is being asked to use are valid, invalid, or boundary values. This type of information may not be normally included in less-documented test cases. Yamaura also found that in certain projects, 21% of the bugs discovered were the result of test cases based on previous test cases, or reused from similar projects.

Yamaura contends the most important use of thoroughly documented test cases is in analyzing the data collected in order to improve the testing process. The test cases–combined with tester bug reports–can be examined to help reveal common types of bugs in the software. If a particular type of bug is common in the software, such as a memory leak, test cases can be added/rewritten in order to focus more attention on this category of bug. Similarly, this type of analysis can be used to determine the units/modules of the software that contain the highest number of reported bugs and allow for refocusing attention on these portions of the software. These types of metrics are also helpful for non-coders and non-testers. Managers need to be aware of the progress being made during the testing cycles and this type of analysis may be helpful to them.

3.2.3 General Guidelines

When creating test cases, a tester must first decide what information they wish to obtain from the test. Different methods produce different information, and different projects will require different information, thus the proper format must be chosen and each of sequential, matrix, and automated test cases have advantages and disadvantages. The format chosen will depend primarily on the project and the resources available to the tester. For sequential and matrix-based test cases, the correct length of each test case must be chosen. Kamde, Nandavadekar, and Pawar suggest a length of 8 to 16 steps for sequential test cases and 10 to 20 steps for matrix test cases. Shorter test cases are preferable to longer ones as they leave less room for error and confusion on the part of the tester executing them. An increase in test case density (i.e. the number of test cases in a test cycle) is better than an increase in overall length of individual test cases. Longer test cases also make it more difficult to locate a bug once

it has been detected. Whatever length is used, it should be fairly uniform across all test cases. This will result in more accurate statistics from any analysis that is performed. For example, if tests are of similar length and it was found that it took a tester a week to run half the test cases, it would be reasonable to estimate that another week would be necessary to run the rest. If the test cases are of wildly different lengths, this same metric may become useless. Kaner also proposed that as test cycles are complete, any tests that continually pass should be collapsed together as the chance of them failing decreases as the test cycles progress. Language used in the test cases should be clear and easy to understand, and tests should be written to be as independent from one another as possible. Once a testing cycle is complete, the data should be used to perform analyses as previously described. Most importantly, there is no test case writing silver bullet as different projects, companies, and development teams each call for different approaches to test case design.

3.3 Psychology & Software

One aspect that is lacking from the field of software testing is the application of psychology. As was noted earlier, psychologists have described numerous ways the mind can be tricked or become confused. A very light version of these ideas has made it into test case design, usually as heuristics or practical advice, such as not to make test cases too long. A direct application of psychology to software testing is missing from the literature, although a spattering of inroads have been made to the more general field of software development.

One of the first attempts at this was made by Stacy and MacMillian [35] in which several problems were presented stemming from cognitive biases and poor mental representations during software development. First, the *representativeness bias*, in which the probability of an event is judged by evaluating how similar the event is to instances stored in memory. A series of fair coin tosses resulting in `heads`, `tails`, `tails`, `heads`, `tails`, has equal probability to one of `heads`, `heads`, `heads`, `heads`, `heads`. The first series "looks more random" and a person may therefore believe that it is more likely to occur than a series of straight `heads`. These types of biases may cause a developer to misjudge aspects of the code they are working on. Representativeness bias may cause certain portions of code to appear more salient than others. Stacy and MacMillian proposed an example in which a developer fond of using very long variable names in code will find any other portions of code with very long variables names–even if

not written by the developer–to be much more salient and easier to recall. A second source of possible errors comes from the *availability bias* in which people tend to estimate the frequency at which an event may occur based on how easy or difficult it is to remember or perceive specific instances of such an event. A developer may be more aware of their own contributions to the software and may misjudge the frequency of use or importance of those contributions during development. Finally, the *confirmation bias*–the tendency to seek evidence that supports our ideas–is described as another source of potential problems. For example, if a programmer is checking to make sure an instance of a particular class has been initialized they may only think to check instances of that particular class (supporting evidence), but would not check for uninitialized instances of an unknown class (refuting evidence). This type of behaviour could have a significant impact on the development process as certain scenarios may be left unchecked by the programmer. Software testing and debugging could also suffer as a tester may tend to show that software works and is correct due to confirmation bias, even though their goal is quite the opposite. Although their work was not tested empirically, Stacy and MacMillian serve as an excellent starting point for future work.

Aranda and Easterbrook [1] provide an example of an experimental study combining psychology and software development. Their work examined the application of the anchoring and adjustment effect (see Section 2.3) to estimations of effort (i.e. man-hours) required to complete software development tasks. In the anchoring and adjustment effect, if a person is provided with an anchor (an answer) to a question, their response will be biased in favour of the given answer, regardless of whether they feel that answer is correct or not. During software development, effort estimation has been a particularly difficult nut to crack. Aranda and Easterbrook point out that initial estimates of effort can be off by a factor of four, and even more detailed estimates produced later on in the development process can still be off by as much as half. Although numerous formal techniques exist, Aranda and Easterbrook claim the most common is the *expert-based method* in which technical experts produce effort estimations based on their past experiences. To test whether anchoring and adjustment manifests during effort estimation, Aranda and Easterbrook performed the following experiment. Participants were provided with a specification of a fictional software project including all necessary details for effort estimation. Participants were asked to estimate the length of time in months they thought it would take a team to build the software, how much they believed their estimate were off by, and justifications for each. The specifications provided only differed in that one group was told that a manager on the project believed

that development would take 2 months, another group was told that the manager believed it would take 20 months, and a third was given no estimate by the manager. It was found that a participant's effort estimate was biased towards the anchor provided. The larger the anchor, the higher the average estimate provided by the participant when compared with the low anchor and no anchor conditions. The effect was rather large as demonstrated by the fact that the the best-case estimate for the 20 month anchor was still much longer than the worst-case estimate for the two month anchor, and this may lead to a significant source of errors during the planning stage of software development.

In a similar experimental vein, Guilford, Rugg, and Scott [12] presented an application of the peak-end effect (see Section 2.4) to software usability. Poorly designed software can be unpleasant–even "painful"–for users thus it is entirely possible that a user's experience of a piece of software could be influenced by something like the peak-end effect. A pilot study was conducted to determine if peak-end effect could be used as a predictor of a user's experience. Participants were given a software package to use and asked to rate their enjoyment (or lack thereof) throughout use. The peak-end effect was found to be a predictor of overall enjoyment, albeit a rather weak one. Followup experiments were conducted to further investigate the effect and during these experiments it was found that the peak-end effect was a poor predictor when applied to unpleasant feelings. However, if pleasant feelings were used, peak-end effect became a much greater predictor of a participant's feelings regarding the usability of the software, affecting up to 50 percent of participants.

Chapter 4

The Experiments

Is this, like, some psychological experiment?

—SUBJECT IN MY STUDY

It is an accepted fact that software can never be exhaustively tested. Imagine being asked to test every combination of numbers and operations on a basic calculator program. The task quickly becomes overwhelming and virtually impossible in any reasonable amount of time, and this is a very simple piece of software. To combat this, software testers have developed various techniques to uncover bugs in software without having to perform exhaustive testing. It requires a balancing act between the hunt for bugs and the resources–time, money, people–used to find them.

Automated testing techniques do exist and work tremendously well when they can be applied; however, their use is often restricted to straightforward types of tests and can fail to find more nuanced errors, for which an observer is more suited. Using these automated techniques for testing more complicated forms of software can be prohibitively expensive–in both time and money–and it is often necessary to go beyond these limited techniques, thus human software testers play an integral role in the process but come with their own set of problems. The field of psychology has revealed that the human mind–although capable of wondrous feats–is subject to numerous cognitive biases that hinder reasoning, attention, memory, and decision making, sometimes causing us to make very odd choices or miss very obvious events (see Chapter 2).

The goal of these experiments was to examine the impact of cognitive biases on human software testers. Specifically, I planned to examine two sources of error: the peak-end effect and the subject-expectancy effect. The discovery of the peak-end effect was one of the results of research conducted by Kahneman [8] (see Section 2.4). It was found that human memory of past experiences is based largely on two specific moments of the experience: the peak moment of intensity in relation to the end moment of the experience. People will remember an experience as being more pleasurable if the peak moment of intensity of a negative experience comes well before the end moment of the experience. This effect was retested to see if it applied to people using software by Guilford, Rugg, and Scott [12] where it was found to hold more for pleasurable experiences rather than negative ones. Experiment I was designed to examine whether the effect could impact the decision-making and judgement abilities of a software tester.

The subject-expectancy effect is a variant of the observer-expectancy effect, a common issue for any type of experimental research. Rather than the experimenter influencing the subject to change their behaviour, the subject-expectancy effect causes behavioural changes in the subject based on the results they believe the experimenter wishes to see [26, 27, 28] (see Section 2.5). Additionally, change blindness–a phenomenon in which a person observing a changing scene or image fails to notice the changes (see Section 2.1)– was a possible minor effect to examine and Experiment II was created to study the impact of these two effects on software testers. The goal of these experiments was to determine (a) in what ways do these biases affect a tester and how great are the impacts; and (b) how can we avoid these biases when using human software testers.

4.1 Participants

Participants were drawn from two sources: Ryerson computer science graduate students, and professional software developers/testers that I knew from when I worked as a software tester. The graduate students were recruited via an announcement I made during the weekly Computer Science Graduate Seminar class which all computer science graduate students participate in, while the professionals were contacted directly via email. A total of 16 individuals (3 female) participated in my study, including 11 Ryerson graduate students and 5 software professionals.

One major requirement for these experiments was that they emulate the software testing process

as closely as possible. Participants were told that they would be helping test mathematical software that I had written for a paper unrelated to my thesis. They were also told that the test cases to be used during their participation were generated by an automated test case generating algorithm, which was my actual thesis project. The need for this deception is explained in Section 4.2.

Once an individual expressed interest in participating in the study, I provided them with a detailed overview[1] of the research, the tasks involved, the time commitment required, the risks and benefits, confidentially, and location of the study (see Section A.1). To be accepted into the study the individual was required to have a solid understanding of English and be familiar with basic computer science/software engineering terminology. As every person who participated was either a graduate student in computer science or a software professional, they readily met these criteria. At this point, if the individual wished to participate, I scheduled a session for them to come in and run through the study.

Once the participant arrived at their scheduled time, they were provided with my contact information if they had any questions after their study session was complete. Participants were run through the basics of what the study would entail, given ample opportunity to ask questions, asked to sign a consent form, and to complete a general information questionnaire. For their participation, subjects were not compensated financially in any way and this was made clear during the recruitment phase; however, refreshments were provided during the sessions.

4.2 Deception

This study required a certain degree of deception due to the nature of the cognitive biases being examined. Participants could not be told what was actually being studied as this may have affected their behaviour. Instead, they were told that I was working on a mathematical method for automated test case generation and the study itself was to help in the evaluation of that automated system. If the participants knew ahead of time the purpose of the study was to evaluate whether a software tester is susceptible to cognitive biases their behaviour may have changed to avoid these pitfalls. It was also entirely possible that if participants knew the true purpose of the study they may have altered their behaviour to specifically fall into those cognitive traps, as they may have thought that this is what

[1] This description was of the fake study described above.

was expected of them. Additionally, in Experiment II, Version I, participants were told that the various steps of the software system had certain levels of bugs. These ratings did not always correctly reflect the true "bugginess" of the software. This deception was needed to examine the impact of having certain expectations when testing software (see Section 4.3.3).

Once the study was completed, all graduate students were debriefed and informed of the real intent and goals of the study via email (the content of the debriefing script can be read in Section A.2). The software professionals were debriefed after they had completed their individual sessions as there was no chance that they would reveal the actual purpose of the study to other participants who had yet to have completed their session. The two biases that were examined were described and explanations given regarding how they were tested for. The need for deception was also explained and participants were given the opportunity to ask any questions the may have had regarding the study and both its fake and true goals. It was reiterated that their individual performances were not strictly being evaluated, but rather the study would examine the group as a whole.[2]

4.3 Design

4.3.1 General Experiment Information

Each participant was given the same set of instructions and explanations to prepare them for the experiments. They were also provided with the following materials:

- A set of test cases. In some companies/testing scenarios, test case design is done by a senior member of the test team while the actual testing is carried out by the less experienced members. I choose to emulate this in my experiments and the test cases themselves were written purely in a sequential, black box, specification testing style.
- A type/severity matrix, or bug matrix, so participants knew what type and severity to assign to any bugs encountered (see Table 4.1).
- A number of blank bug reporting forms, to report any bugs they encountered during the experiments (see Figure A.2).

[2]Interestingly, no one complained about the use of deception once it was revealed to them. One participant even jokingly asked if they were in fact a subject in a psychological study (see quote that opens this chapter). The only comment of any significance regarding the deception was a compliment I received from one participant on my ability to lie convincingly to them regarding all aspects of the study, even though most of those aspects were untrue.

Severity	Application Failure	Functionality	Cosmetic
High	◦ Application freezes/crashes	◦ Calculation error ◦ Disabled field	◦ Gibberish text on screen ◦ Large display problems with the application that make it unusable
Medium	◦ N/A	◦ Truncation/rounding errors ◦ Drop boxes have too few/many items ◦ Text field too short for input ◦ Extra unused fields present ◦ Submit form button is missing a confirmation dialog ◦ Submit form button has unnecessary confirmation dialog	◦ Colours are incorrect on page ◦ Order of fields incorrect
Low	◦ N/A	◦ Field should have a default value but doesn't (applies to check boxes, text fields, radio buttons, etc.) ◦ Field has a default value set but shouldn't	◦ Field looks too short (but still accepts input) ◦ Label is misspelled, mislabeled, or label is missing entirely ◦ Button or field is in the wrong location

Table 4.1: Bug Matrix; describing types of bugs and their respective severities

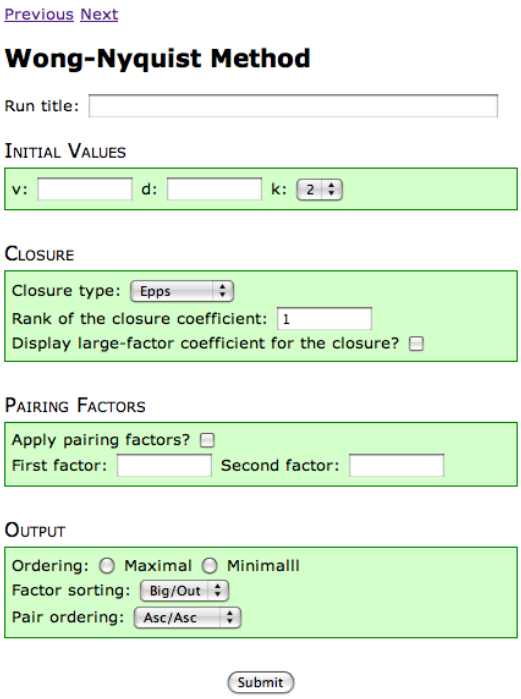
4.3.2 Experiment I

To examine whether a software tester's judgement and decision making abilities are affected by peak-end effect study participants were given the task of testing two versions of a mathematical calculator program which was purported to calculate complex mathematical formulas. The program exclusively consisted of forms made up of text fields, radio buttons, drop-down lists and checkboxes. Each version was designed so that the number of fields to be filled in, the number of keystrokes, and the number of clicks were as similar as possible. The first version of the software was called the *Wong-Nyquist Method* and the second the *Williamsburg Method*. The names were meaningless and chosen to give the software a more realistic look of being mathematical in nature, and both were full of simulated mathematical terminology (see Figure 4.1 for the main screens of each of the two versions). Participants were told that they were not required to understand the mathematics being used in the software, a perfectly plausible scenario as it is often the case that a software tester does not fully understand the software under test.

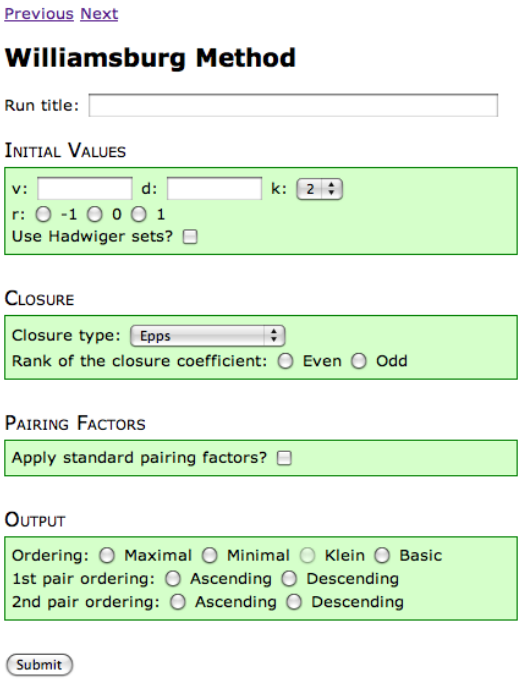
Participants were given a set of test cases to use when testing each version. They were told that each was specifically generated by the automated test case generator for this particular run of the experiment–part of the deception. In reality, each participant received the same set of test cases (see Appendix B). The test cases were simple and focused primarily on the output generated by the program for a given input. There were 6 of these input/output test cases, plus a single test case to check GUI layout, for a total of 14 test cases across both versions of the software. See Figure 4.2 for an example of the output generated by Test 2 from Appendix B.1.

To simulate the increasing and decreasing levels of negative affect in traditional peak-end effect experiments a number of bugs were planted in each version on purpose.[3] Both versions included bugs with all three severities described by the Bug Matrix (Table 4.1) used in this experiment. Each had two low severity bugs, two medium severity bugs, and one high severity bug. In the Wong-Nyquist Method, participants would encounter bugs that progressively increased in severity, starting with low severity bugs and ending with a high severity bug. In the Williamsburg Method, the opposite progression was encountered by participants, starting with a high severity bug and ending with low severity bugs. See Table 4.2 for a breakdown of when bugs were encountered during testing and their

[3]This was done as an analog to pain. Unfortunately, it was not a true analogy as the pain present in previous peak-end effect studies was a continuous experience but finding bugs was not.



(a) Wong-Nyquist Method main screen



(b) Williamsburg Method main screen

Figure 4.1: Main screens of the "mathematical software" under test

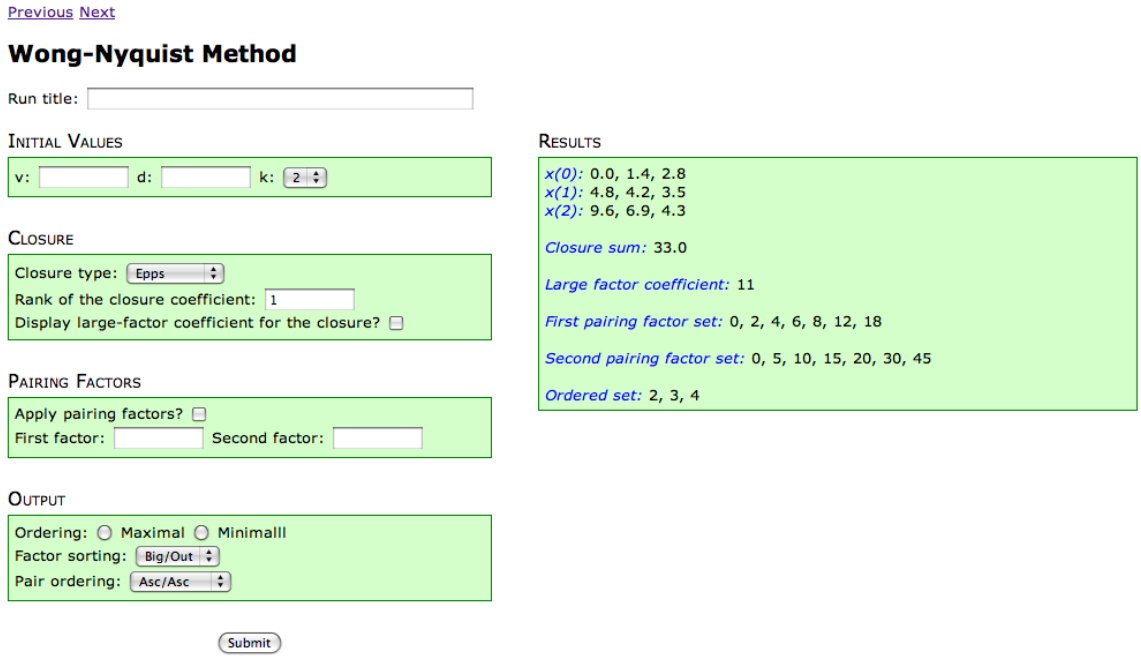


Figure 4.2: Wong-Nyquist Method sample output for Test 2 of the provided test cases

	Wong-Nyquist bugs		*Williamsburg bugs*	
Test Number	**Severity**	**Type**	**Severity**	**Type**
1	No bug	–	High	Functionality
2	Low	Cosmetic	Medium	Functionality
3	No bug	–	Medium	Functionality
4	Low	Functionality	Low	Cosmetic
5	Medium	Functionality	No bug	–
6	Medium	Functionality	Low	Cosmetic
7	High	Functionality	No bug	–

Table 4.2: Bug severities and types encountered by participants correctly testing the Wong-Nyquist and Williamsburg Methods

types and severities.[4] Five bugs were planted in each of the two versions of the software to be found by the participants. Detailed descriptions of all ten bugs are provided below.

[4]Although I designed the test cases in such a way as to lead the participants to the bugs, their behaviour certainly did not follow this trail of bread crumbs and would often deviate from the path I had laid down. As a result, this table only describes when I planned on them locating each of the listed bugs, not when they most often found them. Also, the design of the test cases were such that during the Williamsburg Method testing, participants should have encountered two bugs during the second test. This includes the bug described as being in the second test, and the bug being described as being in the third. However, many participants caught the third bug during the third test, so it is listed here as such.

Wong-Nyquist Bugs

1. To be encountered during the second test case. Participants were asked to select a checkbox labeled "Minimal"; however, the label on the screen read "Minimalll" (misspelled with three Ls). Severity/Type: Low/Cosmetic.

2. To be encountered during the fourth test case. According to the instructions during this test case, the checkbox labeled "Display large-factor coefficient for the closure?" should have been checked by default; however was unchecked by default. Before this test case, no mention of the default status of this checkbox was made. Severity/Type: Low/Functionality.

3. To be encountered during the fifth test case. In the *Closure* section of the form, there was a drop-down list labeled "Closure type", from which the participant was told to choose "Reverse k-aligned". This option was missing. Severity/Type: Medium/Functionality.

4. To be encountered during the sixth test case. The participant was instructed to enter the value "8.41" into the *Initial Values* field labeled "v:". However, the input length of this field was restricted and would not accept more than three characters. This way the participant could only enter "8.4" before being unable to enter anything else. Severity/Type: Medium/Functionality.

5. To be encountered during the seventh test case. After the participant had entered all the data for this test case and submitted it for calculation, the results that were returned were not those listed in the test case. Severity/Type: High/Functionality.

Williamsburg Bugs

1. To be encountered during the first test case. Participants were asked to click a radio button labeled "Klein" in the *Output* section of the form. The button was disabled and therefore could not be clicked successfully. Severity/Type: High/Functionality.

2. To be encountered during the second test case. The test case stated that upon clicking the *Submit* button at the bottom of the form, a dialog box would appear asking if the values that had been entered are correct. However, no dialog box appeared in the Williamsburg Method (one did appear for the Wong-Nyquist Method). Severity/Type: Medium/Functionality.

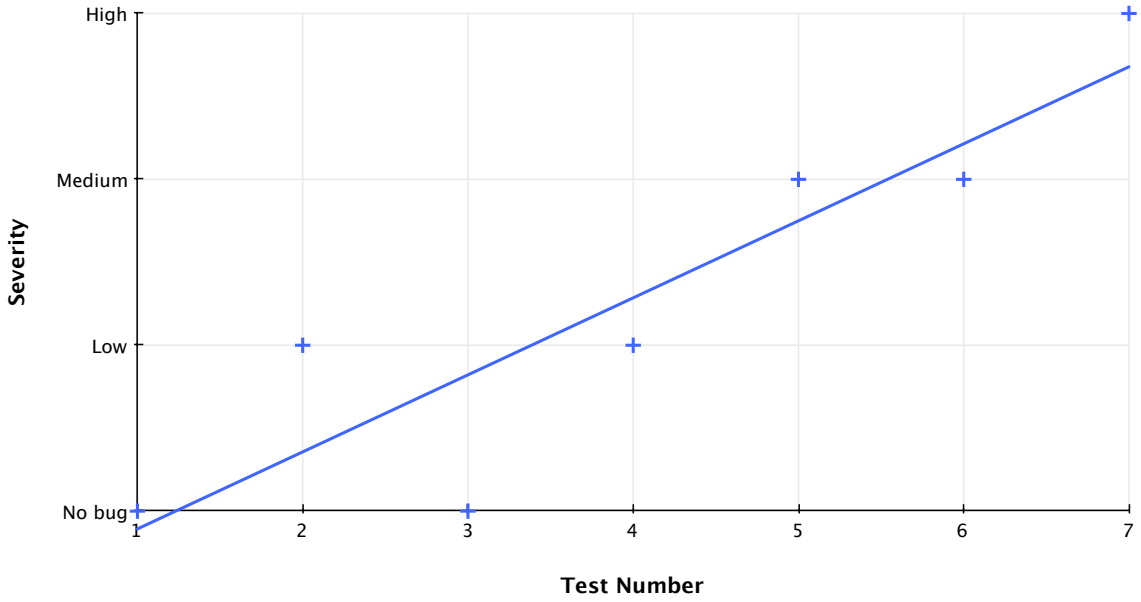
3. To be encountered during the second/third test case. A field in the *Results* section marked "Closure sum" should have been an integer but appeared as a float. Severity/Type: Medium/Functionality.

4. To be encountered during the fourth test case. In the *Closure* section of the form, there was a drop-down list labeled "Closure type", from which the participant was told to select "DeLounge" which was misspelled as either "SeLounge" or "De3Lounge". Severity/Type: Low/Cosmetic.

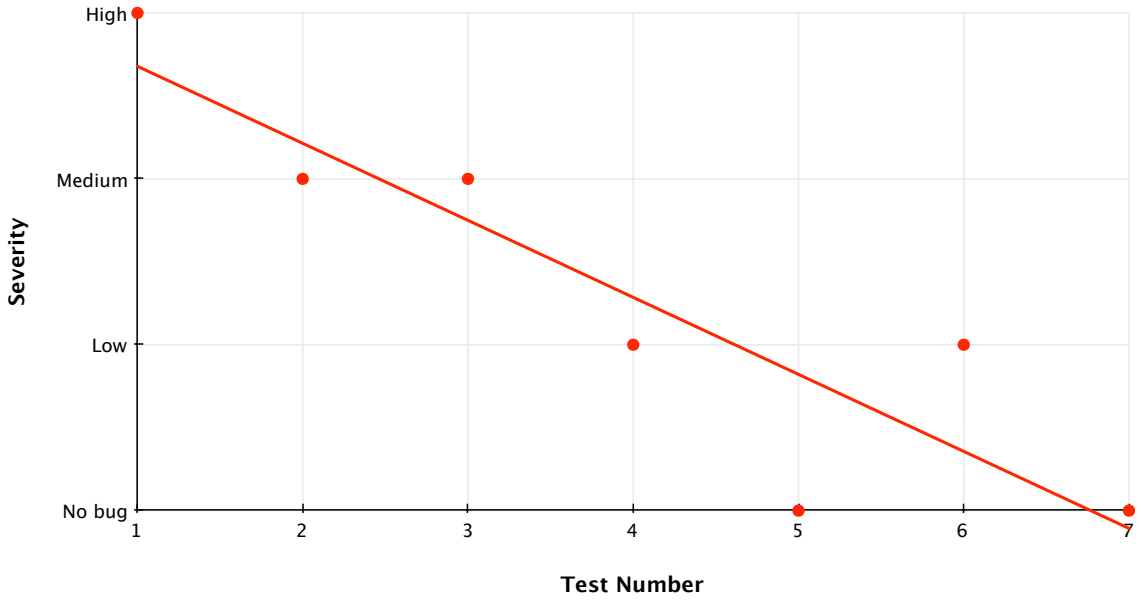
5. To be encountered during the sixth test case. While checking the elements of the interface, participants were told that the *Submit* button at the bottom of the screen should be centred in relation to the form above, however it was actually left-aligned. Severity/Type: Low/Cosmetic.

As one can see by examining the descriptions, types, and severities listed above, the experiences of testing each of the two methods was quite different. During the Wong-Nyquist test a participant would encounter bugs that were progressively increasing in their severity. In contrast, the Williamsburg testing started off with a very serious, high severity bug, followed by bugs of decreasing severity. Figures 4.3a and 4.3b offer graphical representations of the two experiences.

The goal of this experiment was to simulate the pain experienced by participants in the original trials that discovered the peak-end effect (see Section 2.4). This could not be reproduced here exactly as pain is a continuous measure whereas finding bugs in a piece of software is a discrete measure. Nevertheless, the overall experiences of the Wong-Nyquist and Williamsburg Methods simulated the fundamental concepts of import: peaks and ends. In testing the Wong-Nyquist Method, participants experienced the peak moment of intensity ("pain") during their final test in the form of a high severity bug, thus making the peak moment occur at the end of the experience. In testing the Williamsburg Method, participants encountered the high severity bug during the very first test, thus making the peak moment occur very early in the experience. After this initial high severity bug participants' experiences of the test were much more favourable as the bugs encountered were of decreasing severity. If the peak-end effect is experienced by software testers during their testing duties, it would manifest as a tester believing that the Williamsburg version of the software was *better* than the Wong-Nyquist version, even though in actuality they had the same level of bugs present.



(a) Bug severity experience for the Wong-Nyquist Method



(b) Bug severity experience for the Williamsburg Method

Figure 4.3: Graphical representation of the bug severities experiences for the Wong-Nyquist and Williamsburg versions

4.3.3 Experiment II

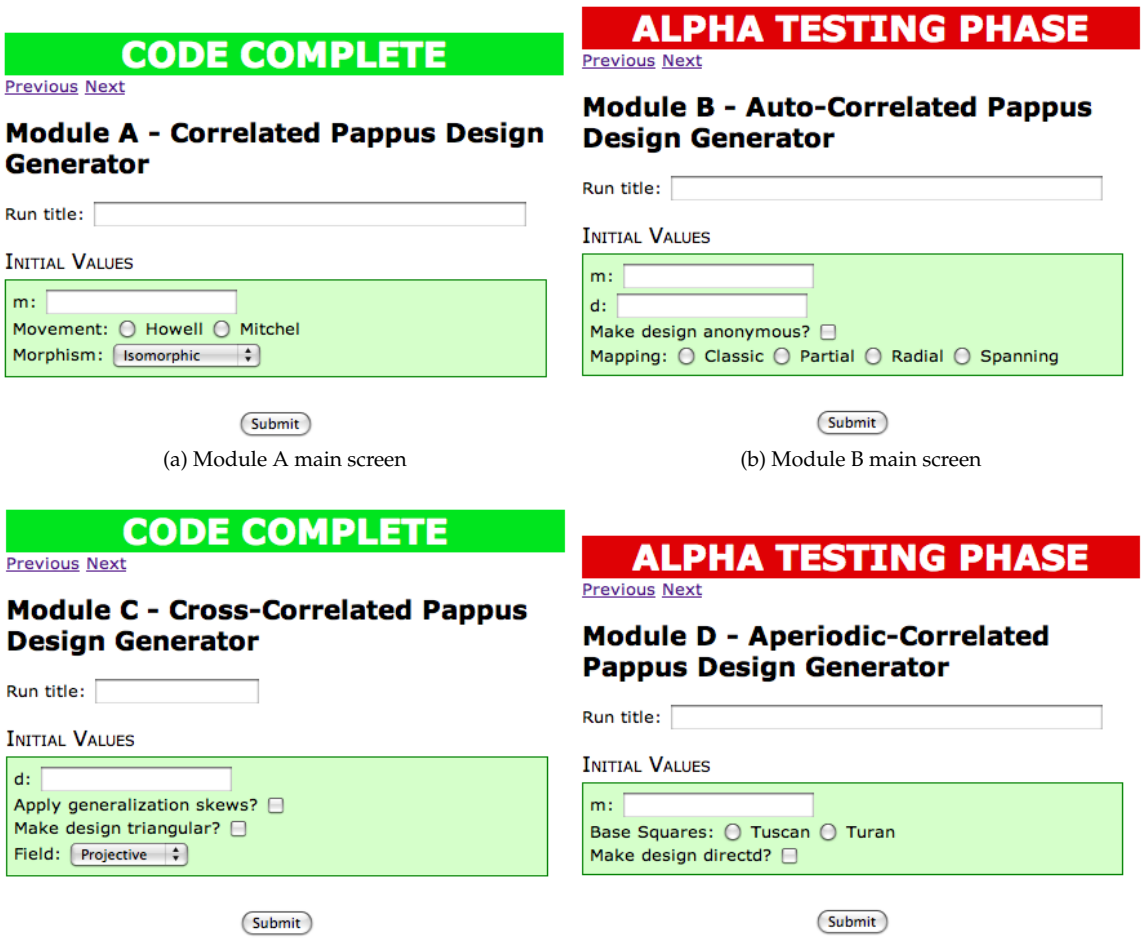
The second experiment focused on the subject-expectancy effect, produced by direct instructions and cues, typically occurring when a subject in an experiment is made aware of the behaviour that they should be displaying during the experiment and then displaying this behaviour, either consciously or not. To see whether participants could be influenced in this way Experiment II was designed and two variants–consisting of essentially the same tasks–were administered to participants. Participants were given mathematical software–similar to that in Experiment I–and test cases to test it. They were told to follow the instructions in the test cases exactly and report any bugs encountered. Both versions had planted bugs for the participants to find.

The initial version of this experiment consisted of four modules of fake mathematical software. Each module consisted of a mix of text fields, drop-down lists, checkboxes, and radio buttons, for the participants to interact with. Each of the four modules were given a status description that told the participant how far along in the development process each module was. Two modules were listed as being "Code Complete" which the participants were told meant that the developer of the software believed that the code had been finished for the module and it had been properly tested with no outstanding issues. The remaining two modules were listed as being in "Alpha Testing Phase" and were very early on in the development cycle and as a result contained many bugs. See Figure 4.4. These fake status descriptions were used to suggest to the participant (the subject) how we believed the testing would play out (the expectancy). One module of each of the two statuses actually contained bugs while the other two did not, regardless of whether or not this agreed with the given statuses. See Table 4.3 for a summary of the modules and their statuses, given and real.

The idea was participants' behaviour would be modified by these statuses, as the status given for each module would set up expectations in the participant, and then the bugs–or lack thereof–would either meet those expectations or not. The goal was to determine if a tester can change their reporting

Module	Status Given	Were bugs present?
A	Code Complete	No
B	Alpha Testing Phase	No
C	Code Complete	Yes
D	Alpha Testing Phase	Yes

Table 4.3: Modules for the initial version of Experiment II



(a) Module A main screen

(b) Module B main screen

(c) Module C main screen

(d) Module D main screen

Figure 4.4: Main screens of the modules for the initial version of Experiment II

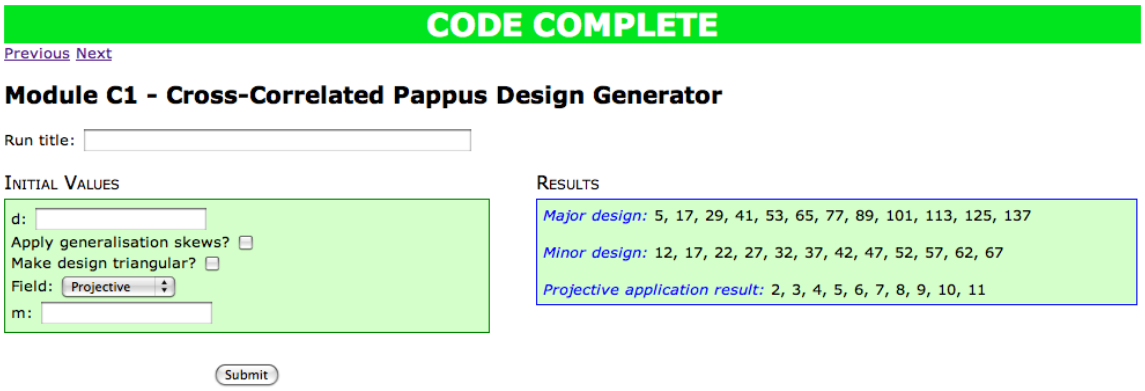
behaviour due to such statuses. For example, in Module A participants would expect to not find bugs in the software and that expectation would have been meet. However, in Module C participants would expect to see no bugs but would actually encounter bugs and the behaviour that followed was of interest. Would the participant ignore these bugs because they believed that the software was complete and tested, so these bugs must be OK or perhaps "not count"? Or would they act appropriately and report the bugs anyway? In Module B, the given status would give the participant the expectation that bugs would be found, but no bugs were actually present in this module. Would the participant therefore make up bugs or nitpick details of the module in order to log something, as this was the expected behaviour for that module? Before actual participants were brought in to run the experiment a short

series of preliminary runs of the experiment were conducted. During these runs it became clear that participants paid no attention whatsoever to the statuses given and simply tested the software as per the test cases, and reported any bugs found. This initial version was deemed unsuccessful and a new version was designed.

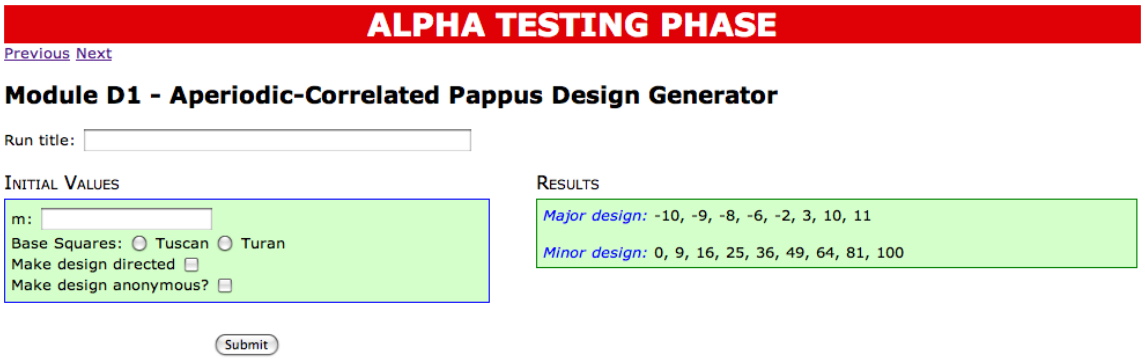
Version I

Participants were asked to test modules of fake mathematical software as before. Four modules were created, each with a status as before ("Code Complete" or "Alpha Testing Phase") and a number of planted bugs. Every module included bugs regardless of the status assigned to it. Each of the modules contained a spelling error, an extra field that was never referenced in the test cases, and the border of either the *Initial Values* or *Results* sections was the wrong colour. Participants either encountered modules C1 and D1 (see Figure 4.5) or modules C2 and D2 (see Figure 4.6). The order that either pair of modules was encountered was counterbalanced across participants (i.e. a quarter of participants saw C1 then D1, a quarter saw D1 then D2, a quarter saw C2 then D2, and a quarter saw D2 then C2). Participants were told that although the modules were related they were independent of one another.

Module C1 was labeled as being "Code Complete" yet contained three bugs. A checkbox was labeled as "Apply generalisation skews?"; however, in the test cases this was labeled as being "Apply generalization skews?" (the former was spelled with an S, the latter with a Z). The textfield labeled "m:" at the bottom of the *Initial Values* section was never mentioned in any test case and was never used during testing. When the *Results* section was displayed, the border colour around the box was not green as it had been previously, but was now blue. Module D1 was labeled as being "Alpha Testing Phase" and contained four bugs. The largest bug was a calculation error, included to act as a sort of proof that this module did indeed deserve the "Alpha Testing Phase" label. A checkbox was labelled as "Make design directed" on the application, but as "Make design directed?" (with the question mark) in the test cases). The checkbox labelled "Make design anonymous?" was never referenced in the test cases during testing. Finally, the border of the *Initial Values* section was blue, rather than the normal green. Module C2 was labeled as being "Code Complete" and contained three bugs. These bugs were identical to those in Module D2, minus the calculation error. This bug was removed to make the module seem to be more correct than it actually was. Module D2 was labelled as being "Alpha Testing Phase" and contained the same bugs as Module C1. In addition to those bugs, a calculation error bug



(a) Module C1 main screen

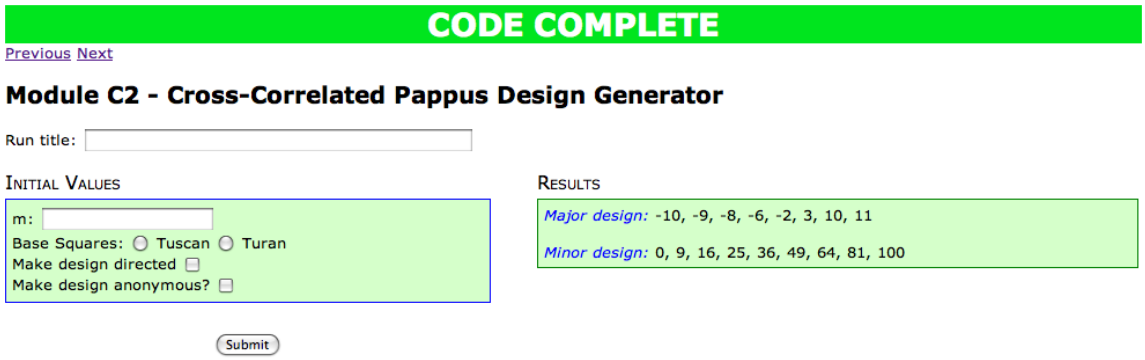


(b) Module D1 main screen

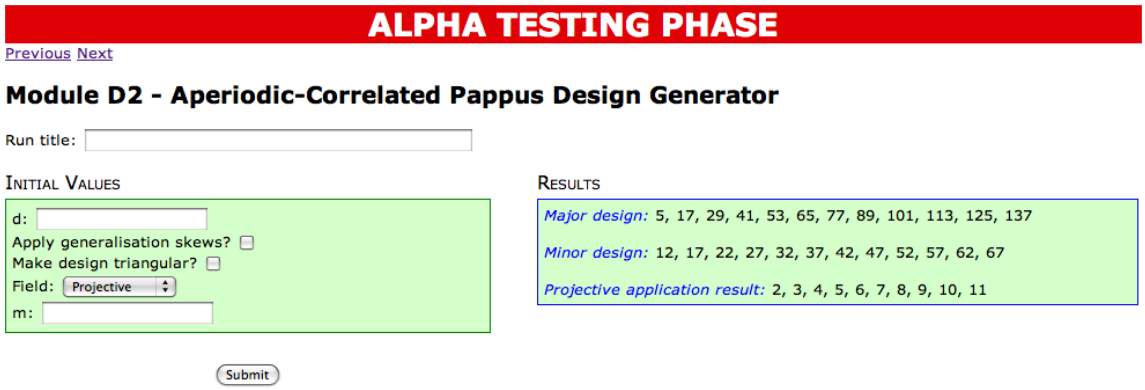
Figure 4.5: Main screens with results displayed of modules C1 & D1 for version I of Experiment II

was added in, similarly to module D1, in order to make this module seem buggier.

With these changes the goal of the experiment changed slightly. Originally, the goal was to see if status labels alone could change participant behaviour. If a participant was presented with a module that contained no bugs but was labelled as being very early on in the development cycle, would they pick at details in the application in order to be able to log something, thus meeting their expected behaviour? Conversely, if presented with a module labelled as being complete and requiring no further work, but that contained many bugs, would the participant ignore these bugs as this might be the behaviour expected of them? In this new version of the experiment the question became more focused. Two of the four modules created for the initial version of the experiment were removed. Module A was removed as it was one that would properly meet a participant's expectations and therefore offered no new insights. Module B was also removed as the software was too limited for a participant to



(a) Module C2 main screen



(b) Module D2 main screen

Figure 4.6: Main screens with results displayed of modules C2 & D2 for version I of Experiment II

"make up" or nitpick what they were testing. The goal of this experiment had now become whether a participant would log bugs that were present in a module that they were told had been fully and completely tested but actually contained bugs (module C) and if their behaviour was at all influenced by the presence of a module (Module D) which properly met their expectations. Additionally, these changes allowed for an examination of the role that change blindness plays in software testing. Would participants be able to notice the changes that would now appear between the modules?

Version II

While a majority of participants completed Version I of Experiment II, a small number of changes were introduced into a second version that was given to the last few participants. The reasons for this are discussed in Section 5.2. In module C1, the checkbox labeled "Make design triangular?" was changed

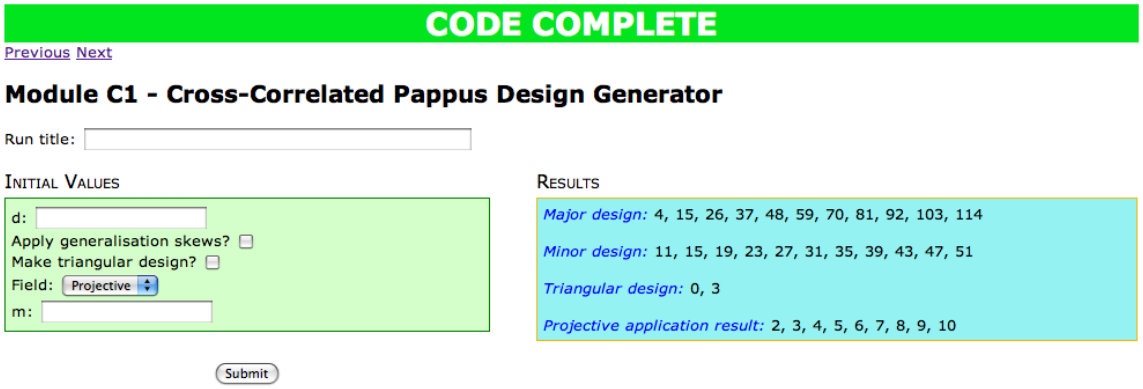


Figure 4.7: Main screen and results of module C1 from Version II of Experiment II

to "Make triangular design?", with the accompanying test case unchanged. Also, the background and border of the *Results* section was changed to be light blue and orange, respectively (see Figure 4.7). In Module D1, the drop-down list labelled "Base Squares:" was changed to "Square Bases:", with the accompanying test case unchanged. Also, the background and border of the *Results* section was changed to be red and blue, respectively. Module C2 had identical changes made as in D1[5] except rather than the *Results* section having the background and border changed, the *Initial Values* section was changed in C2. Module D2[6] had the colour change applied to the *Initial Values* section as opposed to the *Results* section.

4.3.4 Experiment Procedure

All participant sessions took place in the George Vari Engineering and Computing Centre, 245 Church Street, room ENG 232. This room is set up as a usability testing lab, with an inner isolation room, inside the room proper. The inner room also contained a large window so that participant actions could be observed from the outer room. Participants were seated in the inner isolation room while I observed their actions from the outer room. The only people present during the study were the participant and myself. One of the main goals was to recreate the software testing process as closely as was allowable. Participants were given a computer, the various materials listed above, and asked to test the software simply by following the given instructions and test cases. This black box style of testing is common in

[5]Both were identical in Version I but D1 contains an added calculation error not present in C2. The same is true in Version II.
[6]As with C2, D2 is identical to C1 with an added calculation error.

certain companies and for certain types of software. Best practices encourage that the person who wrote the test cases does not actually execute them. Participants were given the opportunity to ask whatever questions they felt were necessary in order to complete the tests; however, the majority required no additional instructions beyond what was initially provided.

Once they arrived in the room participants were given a description of the project, a consent form to sign, and their initial instructions for Experiment I. They would then begin to test either the Wong-Nyquist Method or the Williamsburg Method, half of the participants were assigned to begin with the former, the other half with the latter (i.e. participants were counterbalanced). I left the inner room at this stage and observed from the outer room and would only reenter if the participant needed clarification or they had completed this portion of the experiment. Once this first testing phase was complete I would came back into the inner room, take the booklet of test cases that the participant had just completed, and give them the other test booklet. The participant would then continue testing until they had completed the second booklet of tests and thus completing the testing for both the Wong-Nyquist and Williamsburg Methods. At this point in the experiment the participants were given a short break and refreshments. I would also ask them to complete a general information questionnaire during this time (see Figure A.1) while I would step out of the inner room for several minutes to allow them time to rest and complete the form. The break was included to allow the participant to solidify their memories of the past experience, a requirement for the peak-end effect (as it is a memory-based effect). Similar breaks were used in the original peak-end effect studies.

Once the break had concluded I returned to the inner room and conducted a brief interview with the participant. Each participant was asked to compare the two versions of software they had just tested and judge if one of the versions was better than the other. It was explained to them that for my thesis work I would only have the time and resources to fully develop one of the two versions and I was using the testing process to decide which version to continue developing. Their input–as they had just tested the software–was highly valuable to me. Once the participant made a choice (Wong-Nyquist, Williamsburg, or neither) I would ask follow up questions to determine why they had chosen the way they had. I then explained the second half of the session to the participant and provided them with the test cases required for Experiment II. I would again leave the inner room and only return if they had a question or once the they had completed testing. Once the testing for Experiment II was complete participants would be debriefed and were free to leave.

Chapter 5

The Results

However beautiful the strategy, you should occasionally look at the results

—WINSTON CHURCHILL

5.1 Experiment I

A summary of all 16 participants' results is provided in Table 5.1. The first column assigns each participant a number which will be used to refer to specific participants throughout this chapter. Columns two and three summarize the results for the Wong-Nyquist Method, and columns four and five summarize the results for the Williamsburg Method. The columns labelled *Bugs* list the total number of bugs that the participant logged for that version of the software. Had the participant logged only the bugs that were planted in the software and nothing else, they should have found a total of five bugs for each version. The columns labelled *Score* list the "severity score" assigned by the participant to each of the Wong-Nyquist and Williamsburg versions. These were obtained by assigning values to each level of bug severity and totaling the severities the participant assigned to each bug they reported. Low severity bugs were assigned a score of 1, medium severity bugs were assigned a score of 2, and high severity bugs were assigned a score of 3. Each of the Wong-Nyquist and Williamsburg versions contained two low severity bugs, two medium severity bugs, and one high severity bug. Therefore,

Participant	*Wong-Nyquist*		*Williamsburg*		Score Diff.	Bug Diff.	Choice
	Score	Bugs	Score	Bugs			
1	7	3	10	5	3	2	Williamsburg
2	8	3	11	5	3	2	Neither
3	10	4	20	9	10	5	Wong-Nyquist
4	10	4	8	5	-2	1	Williamsburg
5	10	5	15	7	5	2	Neither
6	12	6	16	8	4	2	Williamsburg
7	12	5	18	8	6	3	Wong-Nyquist
8	7	4	14	9	7	5	Wong-Nyquist
9	9	4	10	7	1	3	Williamsburg
10	13	7	14	6	1	-1	Williamsburg
11	10	5	10	5	0	0	Neither
12	13	6	15	7	2	1	Williamsburg
13	8	5	9	5	1	0	Williamsburg
14	11	5	14	7	3	2	Wong-Nyquist
15	14	6	11	5	-3	-1	Wong-Nyquist
16	8	4	14	6	6	2	Wong-Nyquist

Table 5.1: Participant severity scores and number of bugs logged for Experiment I

had a participant logged those five bugs and nothing else a severity score of 9 ($1 + 1 + 2 + 2 + 3 = 9$) would be obtained. The columns *Score Diff.* and *Bug Diff.* list the difference between the severity scores and bug totals for each participant, obtained by subtracting the respective Wong-Nyquist scores from their Williamsburg counterparts. If a participant found each version to be completely equal in its "bugginess" the differences should both be 0. If the values in these columns is positive, that means the participant found Williamsburg to be more buggy. If the values in these columns is negative, the participant found Wong-Nyquist to be more buggy. The final column labelled *Choice* lists each participant's response when asked which version they believed was more buggy. This question was asked after the short break during the testing session and the participant did not have access to any of the bug reports they had logged, thus their answer was purely from memory.

The first thing that should be noticed is that only one participant (participant #11) completed this experiment as was intended, meaning that they logged just the five planted bugs for each version and assigned the intended severities to each. Only one other participant (participant #13) was close to following the intended behaviour, while all others significantly deviated from what was intended and expected. This alone is a very important result. Given the same instructions, same materials, same environment, etc. participants showed tremendous differences in performance. Previous experience

may explain the performance of participant #11, who had exposure to development and testing in both school course work and industry; however, participant #13 had no comparable experience. The role of experience in participant performance is discussed in Section 5.4.

The primary goal of this experiment was to examine the peak-end effect and its impact on software testing. To do this I compared a participant's choice of which of the versions was rated as better, to their perceived experience testing those versions as was detailed by their bug reports. For the peak-end effect to hold a participant should have chosen the Williamsburg version of the software as the better version and their experience of the Williamsburg version must have been equal to–or worse–than their experience of the Wong-Nyquist version. These experiences were measured by the severity scores and the differences between them as detailed in Table 5.1. If a participant experienced the Wong-Nyquist and Williamsburg versions of the software to be equal, their severity score would have been zero. If the participant experienced the Williamsburg version as being a poorer experience than the Wong-Nyquist version, their severity score would be positive. Finally, if the participant experienced the Wong-Nyquist version to be poorer than the Williamsburg version, their severity score would be negative.

Before any analyses could be performed to check for the peak-end effect holding several participants' data had to be removed. Participants #3, 7, 8, and 16 each had significant inconsistencies and anomalies in their severity scores. These participants' perceived experiences of the two versions deviated from each other–and the indented experience–to such a degree that they left the realm where the peak-end effect could possible apply. Their experiences were such that one version of the software was so overwhelmingly bad, it would make no sense to chose it as being better than the other. For example, participant #3 assigned a severity score of 20 to the Williamsburg version, twice his assigned severity score to the Wong-Nyquist version and more than twice the intended severity score. Upon examination of his bug reports it was found he consistently logged the same bug multiple times, despite being instructed to only log a bug once. His experience was so different from the intended experience that a direct comparison would be very difficult to accomplish. Participants #7, 8, and 16 had similarly deviant experiences. The participants who logged a significant difference between the two methods could not be expected to show the peak-end effect and a cutoff was introduced to select participants for removal. A participant was removed from the analysis if their severity score difference was greater than ± 5, meaning that their experience of one of the two versions deviated by more than half of what was intended and was therefore deemed unusable. In essence, their experience no longer had the in-

creasing or decreasing unpleasantness for peak-end effect to be a factor but was something else entirely. The bug reports for each of the four removed participants were examined and it became evident that they were not following the instructions that were given to them. Common errors included logging the same bug multiple times and ignoring the provided rubric and making up their own severities for bugs.

Participants #2, 5, and 11 were also removed from analysis. These were the participants who when asked to chose which version was better replied that neither was. If a participant chose neither, it may seem said participant was unaffected by the peak-end effect and should be held up as bastion of rationality. This would especially be the case for participant #11, the only person to complete the experiment as was intended. However, participants #2, 5, and 11 made the right choice for the wrong reason. During the interview portion each of these participants were asked why they chose neither version. All three responded that they did not feel they possessed enough information about how the software was developed, how it worked, and what exactly it was supposed to do, in order to make an educated judgement of which should be chosen for continued development (rather than making a judgement based on the tests they had just conducted). Essentially, these three participants chose neither as a form of abstaining, not because this was their actual choice.

The remaining participant data used for analysis can be found in Table 5.2. These data reveal that of the nine remaining participants, seven chose the Williamsburg version as better. Each made that choice even after reporting that the Williamsburg version had a higher total severity score, had more bugs, or both.[1] The remaining two participants–#14 and 15–chose the Wong-Nyquist version over the Williamsburg version. Participant #14 seemed completely unaffected by the peak-end effect. He found the Williamsburg version to be worse and chose Wong-Nyquist. Participant #15 presented with a much different behaviour. He found the Wong-Nyquist version to contain more bugs, yet still chose it as better. When asked to explain his decision during the interview participant #15 explained that he preferred the Wong-Nyquist version because it had a nicer beginning and that his decision was purely based on how each of the two versions began and what expectations those beginnings created for him. Therefore, out of the eight participants that performed the experiment with no inherent expectations seven chose the Williamsburg version, despite assigning it a higher severity score and logging more

[1]Participant #2 had a negative severity score due to erroneously logging the high severity bug in Williamsburg as a low severity bug. The participant caught this mistake during the experiment session but did not update their already logged bug. This was left unchanged in the table.

Participant	*Wong-Nyquist*		*Williamsburg*		Score Diff.	Bug Diff.	Choice
	Score	**Bugs**	**Score**	**Bugs**			
1	7	3	10	5	3	2	Williamsburg
4	10	4	8	5	-2	1	Williamsburg
6	12	6	16	8	4	2	Williamsburg
9	9	4	10	7	1	3	Williamsburg
10	13	7	14	6	1	-1	Williamsburg
12	13	6	15	7	2	1	Williamsburg
13	8	5	9	5	1	0	Williamsburg
14	11	5	14	7	3	2	Wong-Nyquist
15	14	6	11	5	-3	-1	Wong-Nyquist

Table 5.2: Participant severity scores and number of bugs logged for Experiment I, with certain participant data removed

bugs for it. As participants were also counterbalanced—half tested the Wong-Nyquist version first, half tested Williamsburg version first—it can also be said that the order in which the versions were ended had no impact on the results.

In order to make sure that the observed results were not random but actually due to the peak-end effect a statistical analysis was performed. A binomial distribution was obtained, which is commonly used to analyze independent success/fail experiments in which each trial is independently successful with probability p. The probability of obtaining r successes in n trials is

$$P(r) = \binom{n}{r}(p)^r(1-p)^{n-r} \tag{5.1}$$

The following values were used: $n = 8$, the total number of trials; $r = 1$, representing participant #14, the only participant choosing Wong-Nyquist; $p = 0.50$, representing an even choice between the Wong-Nyquist version or the Williamsburg version.

$$\begin{aligned} P(r) &= \binom{8}{1}(0.50)^1(1-0.50)^{8-1} \\ &= (8)(0.50)(0.50)^7 \\ &= 0.03125 \end{aligned} \tag{5.2}$$

As equation (5.2) shows, if there was no population difference in the preference for the two versions, then the probability of observing this result is 3.125%. Overall, participants found the Williamsburg ver-

sion to be worse by a significant margin and therefore it may be argued that the value for p should be less than 0.5, which would lower the value of $P(r)$ even further; however, it was felt that putting a numerical value on this effect would be difficult to do and the result is convincing nonetheless. Therefore, the pattern observed from the experiment–of participants choosing the Williamsburg version over the Wong-Nyquist version despite having rated the Williamsburg version as being of poorer quality–was most likely due to the peak-end effect and was not a random occurrence.

5.2 Experiment II

Experiment II did not mirror the rousing success of Experiment I. The goal of Experiment II was to determine if a tester's actions during testing could be influenced by statuses provided to them of the software under test. These statuses were shown to each participant at the top of every module they were testing (see Figure 4.5 for an example), as well as provided verbally when the experiment was explained to them. A summary of all 16 participants' results is provided in Table 5.3. The first column assigns each participant a number as in the previous section. Columns two to four summarize whether or not the participant found the planted bugs for whichever version of Module C they were testing, either Module C1 or Module C2. Columns five to eight summarize whether or not the participant found the planted bugs for whichever version of Module D they were testing, either Module D1 or Module D2. These bugs are summarized in Section 4.3.3.

It became evident from examining the table that participants were unaffected by the statuses provided. In almost every case a participant's ability to locate bugs of one type in one module of the software was mirrored in the other module, and there was no evidence to suggest that a participant was affected by the purported software status. Several participants mentioned after their sessions were complete that they paid no particular attention to the statuses and treated each module simply as a piece of software that needed testing. This is actually a positive result for software testing in general.

However, two interesting patterns did emerge from this experiment. Firstly, participants seemed to have trouble detecting spelling errors. It should be noted that the test cases provided to participants included the correct spellings. Experiment II provided each participant with two spelling error bugs to find. Of the 32 ($16 \times 2 = 32$) spelling errors, a total of only 7 were detected, or about 22%. Additionally, only participants #10 and 13 detected both of their spelling errors, all other participants that caught a

Participant	*Module Cx*			*Module Dx*			
	Spelling	**Extra Field**	**Border**	**Error**	**Spelling**	**Extra Field**	**Border**
1	No	Yes	No	Yes	No	Yes	No
2	No	Yes	No	Yes	No	Yes	No
3	No	Yes	No	Yes	No	Yes	No
4	No	No	No	Yes	No	Yes	No
5	No	Yes	No	Yes	No	Yes	No
6	Yes	No	No	Yes	No	Yes	No
7	No	Yes	No	Yes	No	Yes	No
8	Yes	Yes	No	Yes	No	Yes	No
9	No	Yes	No	Yes	No	Yes	No
10	Yes	Yes	No	Yes	Yes	Yes	No
11	No	Yes	No	Yes	No	Yes	No
12	No	Yes	No	Yes	No	Yes	No
13	Yes	Yes	No	Yes	Yes	Yes	No
14	No	Yes	No	Yes	Yes	No	No
15	No	No	No	Yes	No	No	No
16	No	Yes	No	Yes	No	No	No

Table 5.3: Participant bug detection for Experiment II, Version I

spelling error only caught one of the two. If the spelling errors from Experiment I are included, there were 4 potential spelling errors for participants to detect for a total of 64 spelling errors. Of these only 31 were detected in total, or about 48%. Again, only participants #10 and 13 were able to detect all four spelling errors. Version II of Experiment II was administered to participants #14, 15, and 16. This version included several added bugs to be detected. These additional planted bugs are described in Section 4.3.3. A summary of the results of these additional bugs is provided in Table 5.4. Although the participant pool for Version II was very small it was found that word order–an extension of spelling errors–was also a problem.

Secondly, participants also had difficulty detecting colour changes in the borders and backgrounds of the sections of the forms. As Tables 5.3 and 5.4 show, all sixteen participants missed logging the change in border colour and only one logged a bug for a change in the background, a rather significant change (see Figure 4.7 for an example). Although the information was not gathered during the experiment, it is highly unlikely that all participants in the experiment were colourblind. The border colour itself was never explicitly described in the test cases; however, the coloured section boxes were described win the following manner: "In the green Initial Values section, there is a text field...", in every test case for both experiments. That participants did not log the rather drastic colour change of back-

Participant	*Module Cx*		*Module Dx*	
	Word Order	**Background**	**Word Order**	**Background**
14	Yes	No	Yes	No
15	No	No	No	No
16	No	Yes	No	Yes

Table 5.4: Participant bug detection for the extra bugs in Experiment II, Version II

ground of the coloured sections is more troubling. Did the participants simply decide not to log the difference although the coloured section boxes are always described as being "green" in the test cases, or did the participants simply not notice the difference?

If the former, this raises the issue of participants no longer paying attention to/reading test cases of a set, repetitive form, as were the test cases in this experiment. Perhaps participants "zone out" certain details over a period of time. These details are perhaps those that are consistently stated in the test cases, but as no bugs have occurred and the participant becomes accustomed to these and eventually starts ignoring the relevant parts in the test cases, forgetting to always check for them when testing. The latter perhaps unveils some possible form of change blindness at work, where the participant is simply unaware that a change has occurred. As was described earlier change blindness can be very extreme at times, thus it is entirely possible that the participant was unaware a change had even occurred. Both raise troubling questions and should be investigated further.

5.3 The "Reverse k-aligned" Mistake

During Experiment I an interesting effect was discovered in the Wong-Nyquist portion of the testing. The fourth test case asked participants to select "k-aligned" from a drop-down list labelled *Closure Type* and participants would then continue with the test case without issue. In the subsequent test case, participants were asked to select from the same drop-down list, this time selecting "reverse k-aligned"; however, this option was missing from the drop-down list as it was one of the five planted bugs in the Wong-Nyquist version.

During the planning stages I envisioned that the typical user would to stop at this point and log the bug as a missing option in a drop-down list. The user would then stop executing that test case–as nothing more could be done–and move on to the next test. This is standard practice amongst software testers and participants were instructed to do so during the experiment. Of the first 13 participants

only 7 caught the bug, logged it, and moved on to the next test case; however, when it came time to choose "reverse k-aligned" from the drop-down list, the 6 remaining participants all chose "k-aligned" without logging a difference and continued executing the test case. As the participants had not selected the correct inputs, the software was not able to generate the expected outcome. This would then result in these participants logging a calculation error bug, believing the software had generated incorrect output for correct input. Of the six participants who made this error only one decided to repeat the test case in order to verify the inputs he had provided. On his second go-around, the participant noticed the problem and was able to log it as intended. A small change was made to Experiment I at the same time Version II of the Experiment II was implemented. The test case that instructed participants to choose "reverse k-aligned" for the *Closure Type* was changed to "French"–a drastic difference from the "k-aligned" that participants previously selected–and all three participants that completed this version had no trouble logging the bug when presented in this way.

I believe this error occurred due to some form of *difference blindness*. Participants simply did not notice a difference between what the test case said ("reverse k-aligned") and what was presented to them on the screen ("k-aligned") and chose one that most closely resembled their target. Perhaps the participant simply honed in on the "k-aligned" portion of "reverse k-aligned", recognizing it as being something they had just seen and thought this must be what was being asked of them. This suggests the troubling thought that participants are not reading test cases in detail but rather ferreting out keywords when executing test cases.

5.4 The Role of Experience

Earlier in this chapter the topic of participant experience was broached. An examination of Table 5.1 revealed that only a single participant performed as was intended, while all other participants deviated in some way, demonstrating wide differences in performance given the same tasks, instructions, materials, etc. Perhaps the past experiences of the participants influenced their behaviour, with more experienced testers deviating to a smaller degree from what was expected than their less experienced counterparts.

The results of the 16 participants were reexamined for the role experience played in their performance. Table 5.5 provides a summary of each participant's experience with software testing based on

Participant	Status	*Past Experience*			Exp. Score
		Testing Courses	Development Exp.	Testing Exp.	
1	1st MSc	No	Yes	No	2
2	2nd MSc	No	Yes	No	2
3	2nd MSc	No	No	No	0
4	2nd MSc	Yes	No	No	1
5	SW Test	Yes	No	Yes	4
6	2nd MSc	Yes	No	No	1
7	2nd MSc	No	No	No	0
8	SW Dev	Yes	Yes	Yes	6
9	2nd MSc	No	Yes	Yes	5
10	1st MSc	Yes	No	Yes	4
11	1st MSc	Yes	Yes	No	3
12	2nd MSc	No	No	No	0
13	SW Dev	No	Yes	Yes	5
14	SW Dev	No	Yes	No	2
15	2nd MSc	No	Yes	No	2
16	2nd MSc	No	No	No	0

Table 5.5: Past experience levels of each participant

the answers they provided on the General Information Questionnaire (see Figure A.1) administered during the experiment. The column labelled *Status* represents the participant's current status with "1st MSc" representing a first year MSc graduate student in computer science; "2nd MSc" representing a second year MSc graduate student in computer science; "SW Dev" representing a professional software developer; and "SW Test" representing a professional software tester. Additionally, the column labelled *Experience Score* provides a quantification of their experience. To keep things simple and consistent with previous quantifications, a simple scale from 1 to 6 was used. If the participant had taken software testing courses in the past, this was assigned a score of 1. If the participant had professional software development experience (past or current), this was assigned a score of 2. If the participant had professional testing experience (past or current), this was assigned a score of 3. All participants had taken software development courses in the past.

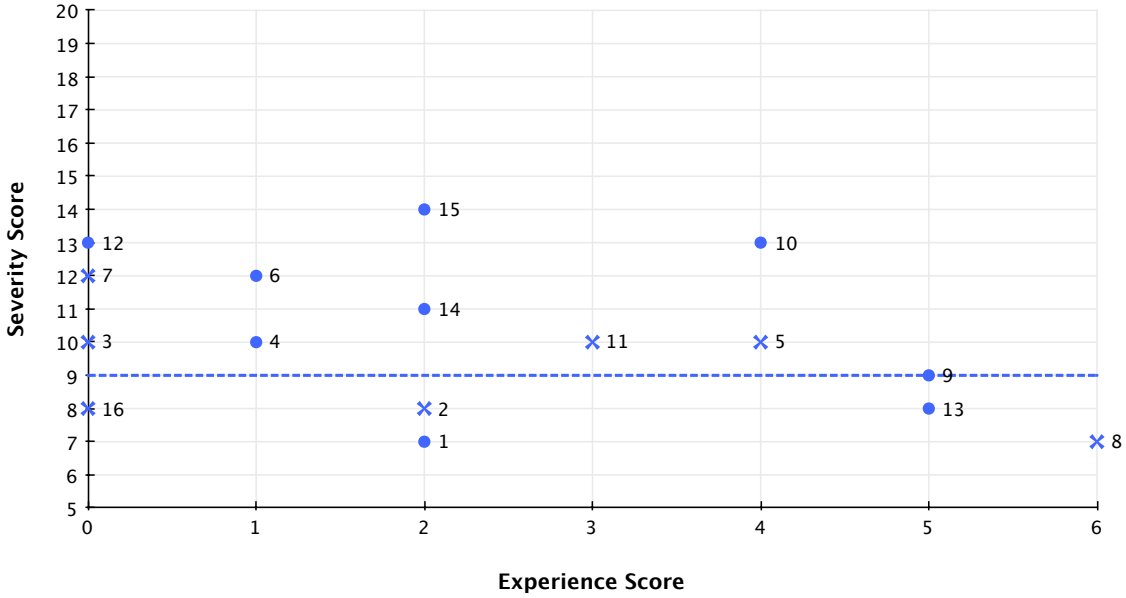
As revealed by Table 5.5 there was a wide range of previous experience amongst participants, with a mean of 2.312. Table 5.6 examines the role of experience in bug reporting, showing each participant's experience score and the number of bugs and severity score assigned for each of the two versions of the mathematical software under test. This information is presented graphically in Figures 5.1 and 5.2. Figure 5.1 presents the experience level of each participant against the severity score they provided for

Participant	Exp. Score	*Wong-Nyquist*		*Williamsburg*	
		Score	**Bugs**	**Score**	**Bugs**
1	2	7	3	10	5
2	2	8	3	11	5
3	0	10	4	20	9
4	1	10	4	8	5
5	4	10	5	15	7
6	1	12	6	16	8
7	0	12	5	18	8
8	6	7	4	14	9
9	5	9	4	10	7
10	4	13	7	14	6
11	3	10	5	10	5
12	0	13	6	15	7
13	5	8	5	8	5
14	2	11	5	14	7
15	2	14	6	11	5
16	0	8	4	14	6

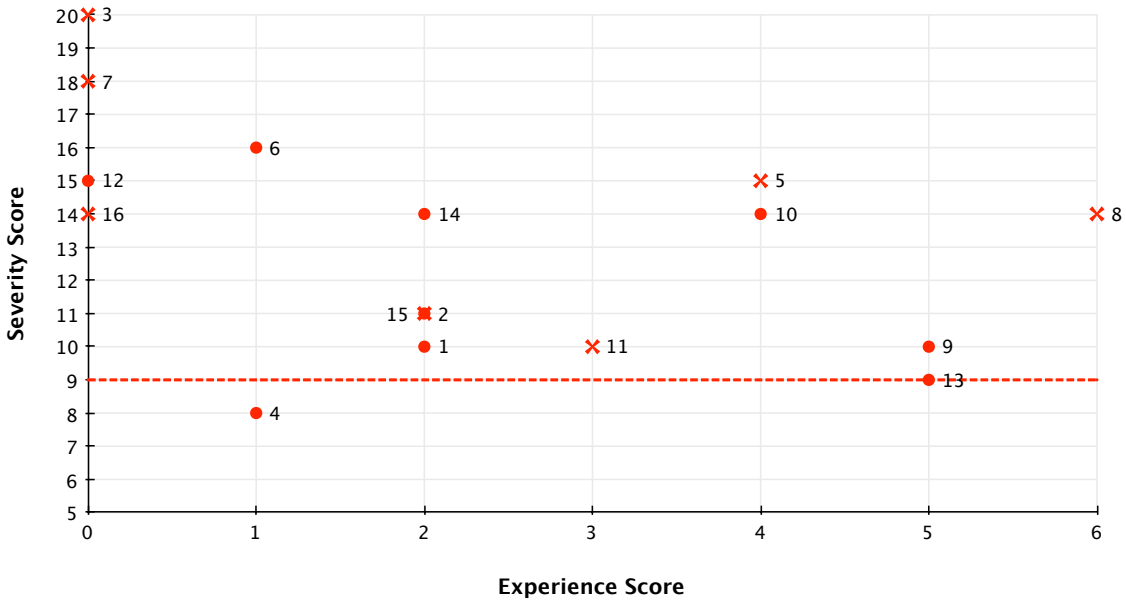
Table 5.6: Participant experience and bugs reported for Wong-Nyquist and Williamsburg Methods

each of the Wong-Nyquist and Williamsburg versions. The dotted line present in both graphs represents the severity score each participant should have logged if they had completed the study as was intended (severity score of 9). Participants represented by a circle on the graph were those whose data was used for the peak-end effect analysis, while the participants represented by an X were those whose data was removed from analysis. This was added to graphically distinguish the two groups. Note that in Figure 5.1b, participants #2 and 15 share the same point (experience score 2, severity score 11) and are represented by an X and circle, respectively. Figure 5.2 presents the experience level of each participant against the number of bugs they logged. The dotted line here represents the intended number of bugs to have been logged (five bugs logged). Again, participants are represented by a circle if their data was kept for peak-end effect analysis or an X if it was removed from analysis. Note that in Figure 5.2a, participants #1 and 2 share the same point (experience score 2, bugs logged 3) and participants #3 and 16 share the same point (experience score 0, bugs logged 4). In Figure 5.2b, participants #1, 2, and 15 all share the same point (experience score 2, bugs reported 5). Of these, participants #1 and 15 are represented by circles, and participants #2, 3, and 15 are represented by Xs.

As can be gleaned from these two sets of graphs, participants tended to log more bugs overall (typically over-logging in the Williamsburg method) and as a result assigned higher severity scores overall.

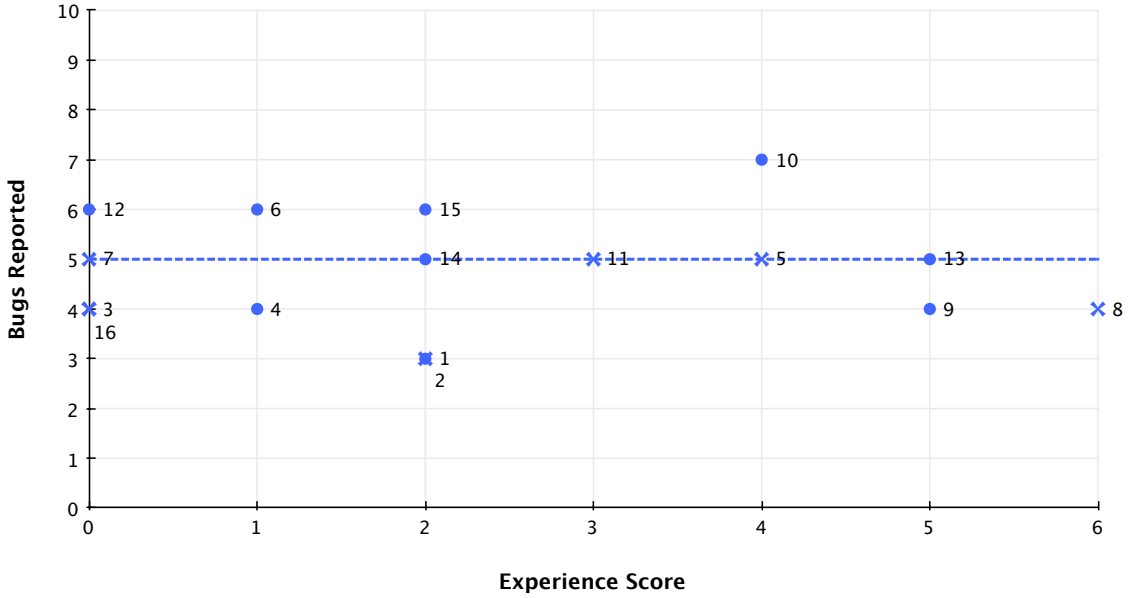


(a) Experience vs. severity score for the Wong-Nyquist Method

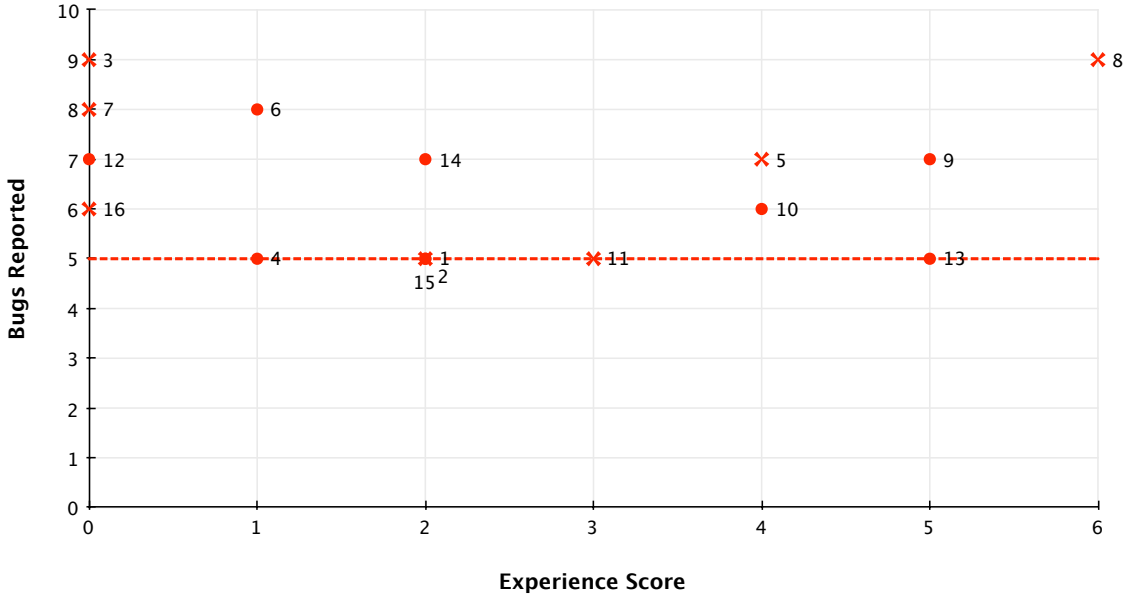


(b) Experience vs. severity score for the Williamsburg Method

Figure 5.1: Graphical representation of experience vs. severity score of bugs logged



(a) Experience vs. bugs reported for the Wong-Nyquist Method



(b) Experience vs. bugs reported for the Williamsburg Method

Figure 5.2: Graphical representation of experience vs. number of bugs logged

Participant	Exp. Score	*Wong-Nyquist*		*Williamsburg*	
		Score Dev.	**Bugs Dev.**	**Score Dev.**	**Bugs Dev.**
1	2	2	2	1	0
2	2	1	2	2	0
3	0	1	1	11	4
4	1	1	1	1	0
5	4	1	0	6	2
6	1	3	1	7	3
7	0	3	0	9	3
8	6	2	1	5	4
9	5	0	1	1	2
10	4	4	2	5	1
11	3	1	0	1	0
12	0	4	1	6	2
13	5	1	0	0	0
14	2	2	0	5	2
15	2	5	1	2	0
16	0	1	1	5	1

Table 5.7: Participant experience and how much deviation was reported in scores and bugs logged

Participants with no past testing or development experience whatsoever tended to (sometimes greatly) over-report bugs in the Williamsburg version, leading to grossly inflated severity scores. In order to determine if there was a correlation between experience level and performance a Pearson product-moment correlation coefficient was calculated, which is often used to described the linear dependence of two variables. To determine if there was a correlation participant experience scores were compared with their individual deviations from the intended severity scores and number of bugs logged for each of the two versions. Table 5.7 presents a summary of this data. The columns labelled *Score Dev.* represent the absolute value of the difference between what the participant assigned and what was expected. The columns labelled *Bugs Dev.* represent the absolute value of the difference between the number of bugs logged by the participant and the intended number to be logged.

Pearson coefficients were calculated for experience score versus each of the four deviant values. A Pearson coefficient can be between -1 and $+1$, inclusively. A coefficient approaching $+1$ denotes a high positive correlation (as one variable increases, so does the second), while a coefficient approaching -1 denotes a high negative correlation (as one variable increases, the second decreases). A coefficient near 0 denotes no correlation. The Pearson coefficients for experience in bug reporting were found to be -0.064 and -0.057 for Wong-Nyquist and Williamsburg, respectively, suggesting no correlation

whatsoever. Interestingly, for the role of experience in correctly assigning severity scores, Pearson coefficients for the Wong-Nyquist and Williamsburg versions were found to be -0.237 and -0.469, respectively. This suggests an existing–albeit rather weak–negative correlation between experience and the correct assignment of severity scores, suggesting that as a participant's experience level increases they may be more likely to properly assign severity scores.

This implies that experience played no factor in a participant being able to actually report bugs, but did seem to play a role in correctly assigning severity scores. Perhaps more experienced testers properly follow the instructions/materials provided to them and assign the correct severities. However, these correlations are rather weak and thus this conclusion may be ill-considered at this point.

5.5 Potential Weaknesses of these Experiments

Firstly, it may be argued that the participants in this study were not all professional software testers and were therefore not representative of how true software testers would behave; however, I believe that this is not a very important concern. I have personally spent eight months testing software professionally. When I was hired I had no formal education or training in software testing, yet was able to perform my duties very well. The type of software testing described here–black box testing where the tester has little knowledge of the software–is a common practice in various companies. It is also not a terribly difficult practice as it only requires the tester to be able to follow a set of written instructions and be able to notice inconsistencies on the screen. It can be argued that using inexperienced testers can even be beneficial as they may serve as an analog for real-world users. However, a thorough investigation of the various findings presented here, done with professional software testers, would be a very worthwhile and important next step.

One other aspect of the experiment worth addressing is the sample size used. Due to the need for participants to be knowledgeable about software and its development and testing, it was difficult to secure a large pool of participants. The search was restricted to students in Ryerson's MSc Computer Science program and professional software developers/testers. As the experiments required a significant length of time to complete and I did not possess the resources to compensate participants for their time, it was difficult to secure a large pool of participants. Also, a number of participant results had to be eliminated from the final analysis, further limiting sample size. Any further work, or reproduction

of these results, should require a much larger sample size.

When an initial version of this work was presented at the weekly Computer Science Graduate Student Seminar the issue of motivation was raised. It was argued that participants were not motivated to perform at their peak level, but if motivated by a prize (i.e. money) or the threat of losing their job their performance may have been "better". Aside from my financial inability to give out prize money–or the logistical and ethical issues with trying to fire students–the argument of motivation playing a factor is fallacious in my mind. Firstly, participants were volunteers and in no way forced or coerced into participating. Potential participants were told during recruitment that there was absolutely no incentive to participate in this study. Therefore, they made the decision to join the study for whatever reason was their own. Secondly, participants were aware that this study would greatly impact my thesis work. I believe this acted as a source of motivation to perform well during the study and I certainly do not believe any participant would try to sabotage the study through poor performance, willfully or otherwise. Thirdly, if we assume that participants were not motivated to perform at their absolute peaks, is this really much different from an industry testing setting? Can we honestly assume that the employees in a company perform to their absolute fullest potential at all times? No, of course not. People are people and thus have both greatly productive days and days where they just want to go home, sit with their significant other and watch *American Idol*. While the motivation provided by a monetary incentive may keep an employee coming to work every day, it is no guarantee of stellar productivity. Many people also dislike their jobs and are perhaps even less motivated to do it well than the participants in my study. Finally, the goal was to test for biases, which should be generic and not motivational. Thus, I believe that motivation, positive or negative, plays little part in these results.

Chapter 6

The Conclusions

This is the end
My only friend, the end
Of our elaborate plans, the end
Of everything that stands, the end
No safety or surprise, the end

—"THE END" BY THE DOORS

6.1 Summary

The human mind is a wondrous thing, allowing us to land on the moon, visit the bottoms of the oceans, and invent bacon sandwiches; all truly amazing feats. However, it is also littered with cognitive biases that warp our perceptions, twist our reasoning, and muddle our logic. A selection of such biases was presented and the question was posed as to how said biases impact the software tester and the software testing process. Through a pair of experiments several interesting effects were discovered. In Experiment I, participants showed a susceptibility to the peak-end effect; their judgements of the software they tested was influenced by which version had the nicer ending. This demonstrated the rather worrisome potential problem of testers' judgement being very easy to influence and manipulate as it seems

to be based on the order in which bugs were encountered. Although participants were not influenced by the given statuses in Experiment II, a number of other potential problems were discovered. Participants displayed difficulty in detecting spelling errors and (at times dramatic) changes in the colour of elements on the screen. A final potential problem was the appearance of a type of "difference blindness" amongst participants, leading to them being unable to notice or discern differences in what was presented in the test cases and what was on the screen. This issue may have also been responsible for the above mentioned difficulty with spelling errors. This thesis has proposed the idea that cognitive biases impact software testers and may hinder the testing process, and demonstrated several of these potential problems experimentally. Further and deeper investigation is require to fully comprehend the extent to which cognitive biases warp the test process and the need for guidelines to be developed to combat this problem.

6.2 Guidelines

The cognitive biases examined in this thesis represent many pitfalls for the modern software tester. As a result, the process of testing should be reworked and changes must be implemented to compensate for said pitfalls. However, it may be said that software is not so bad as to warrant such a reworking. Bugs are not a constant part of the end user's life, are they? For some software this is true, for others it really is not. But the main reason for reexamining and repairing the current software testing framework is not for the direct benefit of the end user, but rather for the companies creating the software. The further along the software development process marches, the more costly bug fixes become. A problem that is detected in the conceptual stage of the process may be fixed by a few strokes of a pen, but if the problem is detected after the software has already launched the costs become staggering. The bug must be first located and patched (no small feat), the patch must be tested to ensure the problem has been rectified, full regression testing must be performed to check the patch did not break a different section of the software, and the patch must finally be deployed to end users. As a result, it should be the goal of testers and the companies they work for to find and eliminate software bugs as early in the process as possible; hence, the need for the reworking I am proposing.

Below is a small collection of suggestions to serve as possible guidelines to improve the testing process. As with everything in this thesis, these guidelines are not exhaustive and may not be applica-

ble/acceptable to every form of testing, project, team, or company.

Change Blindness The effects of change blindness pose a significant problem in the area of software testing. The menus and screens of applications are constantly being drawn, erased, and redrawn as the application is used. Each of these refreshes act as artificial saccades, introducing possible points of failure. To avoid this, test cases must be written so that testers constantly and consistently recheck screen elements after each major refresh. Although these added tests may reduce bugs that slip through due to change blindness they may require a level of additional cost that becomes unfeasible for applications with constant changes to screen elements (e.g. video games.)

Inattentional Blindness As an extension of change blindness, inattentional blindness also poses a challenge during software testing. Testers are constantly required to focus their attention on checking for a single event which may lead them missing other readily apparent bugs due to inattentional blindness. A possible solution is that while a tester is focusing on checking for an event to occur they may quickly glance at other screen elements to check that no major bugs are slipping past their watchful gaze. However, this may not be feasible as the tester may require their full and complete attention to check for that single event they are testing for. Another possible solution is pair testing, where one tester checks for whatever needs to be checked while the second attacks as a guard against inattentional blindness and is constantly checking all other screen elements. Again, this solution may pose too high of a cost for certain types of software applications.

Anchoring and Adjustment Effect Research into this bias has shown that any value provided, even a blatantly random or unrelated value, may be used as an anchor and that this effect can be applied to aspects of the software development process, such as in effort estimation. To avoid this trap, any form of effort estimation (e.g. time necessary to complete a testing cycle) should be done without any anchors given. Management personnel should avoid questions such as "Will it take more or less than two weeks to finish this testing cycle" as this provides the tester whom is answering with an anchor. If the cycle will take less than two weeks, the tester may overestimate due to the anchor and say a period of time longer than what is necessary. This may lead to wasted resources as testing may be complete well

before the next phase is ready begin. If the cycle will take more than two weeks, the tester whom is replying may underestimate the length of time necessary, resulting in scheduling conflicts, useless timetables, and wasted resources. Management and testers must be aware of the potential problems caused by suggesting anchors and avoid their use as much as is reasonable.

Peak-End Effect This effect was shown to have a strong influence on the judgement of a tester through the results of Experiment I. It was found that a tester's judgement regarding the quality of the software they had tested could easily be influenced by the order in which they encountered the bugs in the program. To prevent this effect influencing real-world testers engaged in testing two similar applications or modules, no manager should ask for subjective quality assessments from their testers and no tester should be willing to provide them. A possible exception to this would be projects in which there is only a single, unique module or program being tested. However, the argument could be made that although a tester may not be able to compare the experience of two similar applications, as was done in Experiment I, they may consciously or unconsciously compare the application under test to a similar application that was previously tested. All quality assessments made by testers and management should be done with objective analysis of the bugs currently remaining in the system, and other similar means. In essence, human judgement of quality should be minimized to avoid a potentially biased opinion.

Expectancy Effects Expectancy effects were largely proven not have a significant impact on a tester's perceptions of the application. If such effects are of concern, many established experimental protocols exist to minimize or eliminate these, such as double-blind trials.

Spelling Mistakes These were tricky bugs to catch, as was demonstrated in Section 5.2. Testers may have trouble in noticing the sometimes subtle differences when spelling mistakes appear on screen. Further, if the tester fails to notice the spelling mistake the first time they check, they may assume that the spelling is fine afterwards and not bother to check spelling in future test cycles. As a result, checking for spelling errors should become a primary task during the testing process. Spelling mistakes that are caught after the product has been release are embarrassing for the companies that allowed them. How can customers trust

your software if you cannot even catch a spelling mistake?

To combat this, checking for spelling must become an integral part of the process. During the test cycle a pair of testers should be selected to verify all written elements in menus, GUIs, etc. The two testers should work independently to act as checks for one another. This should be done as a separate test during the cycle and should only be carried out near the completion of the software. Done too soon and the effort may be wasted as new spelling mistakes may be introduced as other, non-spelling-related bugs are fixed. Another possible solution is the use of editors–similar to the publishing industry–or at least incorporating some of their proofreading techniques into the testing process.

Difference Blindness Experiment II also demonstrated the potential problem of difference blindness, where testers did not notice differences between what they were told to click and what was displayed on the screen. This was well documented in Section 5.3 as the "reverse k-aligned mistake". Participants were told to choose "reverse k-aligned" from a drop-down list; however, as one of the five bugs planted in this version of the experiment, this option was missing. A number of participants simply choose the closest option available, namely "k-aligned". This resulted in a calculation and a misreported bug.

In order to stop the "reverse k-aligned mistake", testers must always go back and recheck inputs if incorrect output was generated. If a tester believes they have entered the correct test values, but the output does not match what was expected, the test should be repeated. A tester's reflex may be to immediately log the bug as soon as incorrect output is found; however, this behaviour may be partially responsible for the "reverse k-aligned mistake". Once incorrect output is found, a tester should repeat the test as many times as necessary until they can say with certainty which input behaviour generated the erroneous output. This will decrease the number of false positives that are a consequence of difference blindness.

6.3 Future Work

The first step in any future work examining the affect of cognitive biases on software testers is the assembly of a larger group with more experience in software testing. The subject pool and sample

size used here was adequate for the purposes of this proof-of-concept study but should be greatly expanded in further studies. The cooperation of a corporation with a dedicated, internal testing team would be best. Video game companies would serve as an ideal partner as their internal testers would be highly skilled in both advanced testing techniques and would possess highly developed hand-eye coordination, allowing them to quickly execute most tests.

As was uncovered during Experiment II, participants seemed to have had trouble logging defects related to changes in border and background colour. Whether this was a result of the participant becoming accustomed to the test cases and forgetting to check for certain details; being blind to changes; or some other reason, this problem requires further study. The ability of software testers to detect inconsistencies and changes between similar screens of the software under test is a very important ability and of vital importance to the testing process. It is exactly these unforeseen, test-less bugs that are the most difficult to predict and catch. It is therefore central that software testers have a keen ability to notice out-of-place changes and unexpected bugs when executing their testing duties.

Appendix A

Extra Materials

A.1 Recruitment Information

I am a graduate student at Ryerson University and am conducting research into software testing. An overview of the research is provided below to help you decide whether or not you would be willing to participate in the study. Your participation is completely voluntary and you are under no obligation to participate. Your choice to participate, or not, will not influence any future relations with myself or with Ryerson University.

The purpose of this study is to examine existing techniques for testing software of complex sets of user input. Your status as either a graduate student in computer science or as a software professional qualifies you for participation in this research.

The study requires you to participate in a testing session on campus. During this session you will be asked to test several pieces of mathematical software that have been developed. The testing session is expected to take approximately one (1) hour but this may vary based on your experience level with software testing. Formal knowledge of software testing techniques is not necessary as complete instructions and test cases will be provided to you.

During the session, you will test pieces of mathematical software according to the provided test cases and report any and all bugs encountered. After you have completed testing the software, you will be asked a short series of oral questions regarding your impressions of the software. Due to the nature and research objectives of this study a degree of incomplete disclosure is required. A full debriefing

will be provided immediately after the study is completed.

The study will be conducted at Ryerson University, in the George Vari Engineering and Computing Centre, 245 Church Street, in room ENG 232. The only people present during the study will be you, the researcher, and possibly the researchers thesis supervisor.

Your individual results are confidential. Only the investigator knows the results of the sessions and any identifying information linking you to those results. No personal or identifying information will be disclosed as a result of publishing or presentation.

Your participation is voluntary and you may, if you wish, withdraw your participation at anytime during the course of this study. Your choice to participate, or not, will not influence any future relations with the researcher or with Ryerson University.

The study begins Thursday, February 10th 2010 and ends Friday March 5th 2010, including Reading Week (for Ryerson students) and weekends. If you're willing to participate, please let me know a date and time that you're free. If you have any questions feel free to ask them at any time.

Thank you very much,

Pete Wegier, Principle Investigator
Email: pwegier@scs.ryerson.ca
Mobile phone: 416-904-1527
Department of Computer Science, Ryerson University

A.2 Debriefing Script

Investigator

Pete Wegier, Department of Computer Science, Ryerson University

Stated Purpose of the Study

"This study attempts to examine existing techniques for testing software of complex sets of user input." To achieve this, you participated in a testing session during which you were asked to test several pieces of mathematical software and log any defects that were found. You were also asked your opinions on the quality of the software in order to determine which version should be further developed.

Actual Purpose of the Study

The study was actually designed to examine the impact of cognitive biases of software testers during the testing process. Specifically, the influence of *peak-end effect* and *subject-expectancy effect/priming* on a testers impressions of the software they are testing was examined.

Peak-end Effect

The peak-end effect was discovered during research into how people remember physical pain. It was found that if the most painful moment ("peak") of an experience, such as a medical procedure, occurs before the final moment ("end"), the experience will be remembered more favourably compared to if the peak was the final moment of the experience. It was found that people would even choose to **experience** more overall physical pain as long as it improves the overall **memory** of the experience. The effect was retested with the medical procedure being replaced by the use of a new software system. Here the peak-end effect was found to hold for pleasurable experiences rather than painful ones.

During the testing session, you were asked to test two versions of a piece of mathematical software. For all intents and purposes, these two versions were actually identical. To test each version as per the instructions provided, you should have made the same number of keystrokes, the same number of mouse clicks, etc.

The two versions also contained an identical set of bugs. These bugs were identical in severity, but the order in which you encountered them varied. In one version, the bugs became progressively more severe (thus making the "peak" negative experience to occur in the final moment of the experience), and in the other the bugs became progressively less severe (thus making the "peak" negative experience occur at the beginning of the experience). Your choice of one version over the other was therefore influenced by this peak-end effect, rather than a rational choice between the two versions.

Subject-Expectancy Effect

This is a variant of the *observer-expectancy effect* or *experimenter effect*, which is a common issue for any type of experimental research. Rather than the experimenter influencing the subject to change their behaviour, subject-expectancy effect causes behavioural changes in the subject based on the results they believe the experimenter wishes to see. During the second half of the session, you were asked to test a piece of software with multiple functions. You were either told one of the following scenarios:

- You were asked to test four small functions of an application and were told that two of the functions had a LOW rating of bugs while the other two had a HIGH rating of bugs. In reality, one of the LOW rated functions had a many bugs present and one of the HIGH rated functions had no bugs present. The goal of this session was to determine if your bug reporting was influenced by these ratings.
- You were asked to test two small functions and were told that one was believed to contain no bugs and had been in use for a period of time by the investigator, while the other contained many bugs as it was just recently completed.

Two questions were to be answered:

1. Would subjects ignore (and not report) obvious bugs because they were expecting to find none?
2. Would subjects report bugs when none were present simply because they expecting many bugs?

Justification for Incomplete Disclosure

The reason you were not told the true nature of the research was due to the nature of the cognitive biases being tested. The purpose was to see if a tester is subject to these biases. Knowing that such

biases exist ahead of time may have radically altered your behaviour to avoid them.

It is also entirely possible that if you knew of these biases, you may have altered your behaviour in order to fall prey to these biases, because that is what you think was expected of you. To avoid this from occurring, you were told that the study was to examine testing techniques for mathematical software.

Please keep in mind that individual performance is not being evaluated in this study, but rather the focus is on the performance of the entire group as a whole. Also, your individual results are confidential. Only the investigator knows the results of the sessions and any identifying information linking you to those results. No personal or identifying information will be disclosed as a result of public or private presentation.

Data and Findings

As a participant in this study, you have the right to request, without consequence to you, complete withdrawal of your data from the sample. You may make such a request up to the point of formal dissemination of the findings, which for this study is expected to be 01 May 2010. If you choose to remove your data from the sample, please send a written request to the investigator, Pete Wegier, by emailing pwegier@scs.ryerson.ca.

Questions

Questions can be directed to the investigator, Pete Wegier, by emailing them to pwegier@scs.ryerson.ca. If you wish to contact the Ryerson Ethics board, you may direct your communication to:

Alex Karabanow

Office of the Vice President, Research and Innovation

Ryerson University, 350 Victoria Street, Room YDI 1154

Toronto, Ontario, Canada M5B 2K3

Phone: (416) 979-5000 Ext. 7112, Fax: (416) 979-5336

Email: alex.karabanow@ryerson.ca

A.3 Experimental Materials Provided

In this section, the following materials are available: (a) the General Information Questionnaire given to all participants; (b) the type/severity Bug Matrix used by participants when logging bugs; and (c) the Bug Report Form used by participants to report the bugs they discovered.

The Bug Matrix was based on a similar matrix I used as a professional software tester, slightly modified to fit these experiments. The bugs themselves are different, but the types and severity ratings are unchanged. Many companies use similar types and severities in their own test teams.

The Bug Report Form is also similar to ones used in industry; however, this version is highly stripped-down in order to make it simple and easy to fill out by participants. Industry bug reporting forms are typically much more complex as a much higher level of detail is required. This same level of detail was unnecessary here and was thusly removed.

RYERSON UNIVERSITY

GENERAL INFORMATION QUESTIONNAIRE

STUDY ID: _____

CURRENT STATUS:

☐ - 1st year Masters Student ☐ - 2nd year Masters Student

☐ - Fulltime Software Developer ☐ - Fulltime Software Tester

☐ - Other, please describe:

_____

_____

LEVEL OF EDUCTATION & EXPERIENCE (check all that apply):

☐ - One or more courses in software development

☐ - One or more courses in software testing

☐ - Professional software development experience, if so please describe:

_____

☐ - Professional software testing experience, if so please describe:

_____

☐ - Other relavent education and/or experience, please describe:

_____

_____

Figure A.1: General Information Questionnaire; regarding basic information of each participant

Severity	Application Failure	Functionality	Cosmetic
High	◦ Application freezes/crashes	◦ Calculation error ◦ Disabled field	◦ Gibberish text on screen ◦ Large display problems with the application that make it unusable
Medium	◦ N/A	◦ Truncation/rounding errors ◦ Drop boxes have too few/many items ◦ Text field too short for input ◦ Extra unused fields present ◦ Submit form button is missing a confirmation dialog ◦ Submit form button has unnecessary confirmation dialog	◦ Colours are incorrect on page ◦ Order of fields incorrect
Low	◦ N/A	◦ Field should have a default value but doesn't (applies to check boxes, text fields, radio buttons, etc.) ◦ Field has a default value set but shouldn't	◦ Field looks too short (but still accepts input) ◦ Label is misspelled, mislabeled, or label is missing entirely ◦ Button or field is in the wrong location

Table A.1: Bug Matrix; describing types of bugs and their respective severities

RYERSON UNIVERSITY

BUG REPORT FORM

WHERE:

Module: _____

Test: _____

Step: _____

TYPE: ☐ - Application Failure ☐ - Functionality ☐ - Cosmetic

SEVERITY: ☐ - High ☐ - Medium ☐ - Low

OBSERVATION (Briefly explain your actions and their result):

(Continue on the back if necessary)

Figure A.2: Bug Report; for reporting of any encountered bugs

Appendix B

Test Cases

This appendix provides the reader with all the test cases that were given to participants during the experiment. This includes the test cases for (a) the Wong-Nyquist Method; (b) the Williamsburg Method; and (c) Modules C1, D1, C2, and D2. Not included are the test cases for Modules A, B, C, and D, as these were for the initial version of Experiment II which was not officially run. Additionally, these test cases were very similar to the test cases for the included modules.

The test cases were designed as black box test cases, a common form of testing used in many companies and for many different applications. The test cases did not obviously use any form of equivalence partitioning or boundary-value analysis (see Section 3.1.5) as these are techniques used to create test cases, not run them. Participants were also not aware of what the equivalence classes or boundaries of the application were as they were not familiar with the (fake) math used in the applications.

B.1 Wong-Nyquist Method Test Cases

Test 1

Steps

Check that the following fields and types are correct and present:

1. The title, *Wong-Nyquist Method* is at the top left of the screen, just below two links labeled *Previous* and *Next*.
2. Below the title is a text field labeled *Run title*.
3. In the green Initial Values section, there is a text field labeled v, a text field labeled d, and a drop box labeled k.
4. In the green Closure section, there is a drop box labeled *Closure type*, a text field labeled *Rank of the closure coefficient* with a default value of one (1), and a check box labeled *Display large-factor coefficient for the closure?*.
5. In the green Pairing Factors section, there is a check box labeled *Apply pairing factors?*, and two text fields labeled *First factor* and *Second factor*.
6. In the green Output section, there is a set of radio buttons labeled *Ordering*. There is also one drop box labeled *Factor sorting* and another labeled *Pair ordering*.
7. Below the Output section is a single button labeled *Submit*. It is centred in relation to the middle of the Output section.

Expected Output

None

Test 2

Steps

1. For the Run title enter **Test 2**
2. In the Initial Values section:
   (a) For **v** enter **1.6**
   (b) For **d** enter **2.1**
   (c) For **k** choose **3**
3. In the Closure section:
   (a) For **Closure type** choose **Epps**
   (b) The **Display large-factor coefficient for the closure?** check box is to be checked
4. In the Pairing Factors section:
   (a) Check the **Apply pairing factors?** check box
   (b) For **First factor** enter **2**
   (c) For **Second factor** enter **5**
5. In the Output section:
   (a) For **Ordering** choose **Minimal**
   (b) For **Factor sorting** choose **Big/Out**
   (c) For **Pair ordering** choose **Asc/Asc**
6. Click the *Submit* button
7. A popup box will appear asking *Are you sure these values are correct?*
8. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 1.4, 2.8

x(1): 4.8, 4.2, 3.5

x(2): 9.6, 6.9, 4.3

Closure sum: 33.0

Large factor coefficient: 11

First pairing factor set: 0, 2, 4, 6, 8, 12, 18

Second pairing factor set: 0, 5, 10, 15, 20, 30, 45

Ordered set: 4, 3, 2

Test 3

Steps

1. For the Run title enter **Test 3**
2. In the Initial Values section:
   (a) For **v** enter **2.5**
   (b) For **d** enter **4.99**
   (c) For **k** choose **2**
3. In the Closure section:
   (a) For **Closure type** choose **k-aligned**
   (b) The **Display large-factor coefficient for the closure?** check box is to be checked
4. In the Output section:
   (a) For **Ordering** choose **Maximal**
   (b) For **Factor sorting** choose **Big/Out**
   (c) For **Pair ordering** choose **Asc/Asc**
5. Click the *Submit* button
6. A popup box will appear asking *Are you sure these values are correct?*
7. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 2.2

x(1): 25.3, 26.1

Closure sum: 26.5

Large factor coefficient: 106

Ordered set: 2, 26

Test 4

Steps

1. For the Run title enter **Test 4**
2. In the Initial Values section:
   (a) For **v** enter **5.7**
   (b) For **d** enter **8.0**
   (c) For **k** choose **4**
3. In the Closure section:
   (a) For **Closure type** choose **k-aligned**
   (b) Make sure the **Display large-factor coefficient for the closure?** check box is checked. By default this check box should always be checked
4. In the Output section:
   (a) For **Ordering** choose **Maximal**
   (b) For **Factor sorting** choose **Big/In**
   (c) For **Pair ordering** choose **Desc/Desc**
5. Click the *Submit* button
6. A popup box will appear asking *Are you sure these values are correct?*
7. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 5.0, 9.9, 14.9

x(1): 64.4, 66.7, 68.9, 71.2

x(2): 128.8, 128.4, 127.9, 127.5

x(3): 193.1, 190.0, 187.0, 183.9

Closure sum: 389.5

Large factor coefficient: 6232

Ordered set: 14, 71, 127, 183

Test 5

Steps

1. For the Run title enter **Test 5**
2. In the Initial Values section:
   (a) For **v** enter **7.0**
   (b) For **d** enter **11.0**
   (c) For **k** choose **3**
3. In the Closure section:
   (a) For **Closure type** choose **reverse k-aligned**
   (b) The **Display large-factor coefficient for the closure?** check box is to be checked
4. In the Pairing Factors section:
   (a) Check the **Apply pairing factors?** check box
   (b) For **First factor** enter **4**
   (c) For **Second factor** enter **7**
5. In the Output section:
   (a) For **Ordering** choose **Minimal**
   (b) For **Factor sorting** choose **Big/Out**
   (c) For **Pair ordering** choose **Desc/Asc**
6. Click the *Submit* button
7. A popup box will appear asking *Are you sure these values are correct?*
8. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 5.9, 12.0

x(1): 21.4, 25.8, 29.0

x(2): 24.2, 24.4, 24.6

Closure sum: 57.0

Large factor coefficient: 5337

First pairing factor set: 981, 971, 961, 586, 506, 474, 48, 44, 0

Second pairing factor set: 0, 2, 4, 47, 75, 90, 169, 178, 722

Ordered set: 542, 43, 7

Test 6

Steps

1. For the Run title enter **Test 6**
2. In the Initial Values section:
   (a) For **v** enter **8.41**
   (b) For **d** enter **0.5**
   (c) For **k** choose **4**
3. In the Closure section:
   (a) For **Closure type** choose **Epps**
   (b) The **Display large-factor coefficient for the closure?** check box is to be checked
4. In the Pairing Factors section:
   (a) Check the **Apply pairing factors?** check box
   (b) For **First factor** enter **6**
   (c) For **Second factor** enter **11**
5. In the Output section:
   (a) For **Ordering** choose **Minimal**
   (b) For **Factor sorting** choose **Big/In**
   (c) For **Pair ordering** choose **Asc/Desc**
6. Click the *Submit* button
7. A popup box will appear asking *Are you sure these values are correct?*
8. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 7.3, 14.6, 22.0

x(1): 0.6, 5.3, 9.9, 14.5

x(2): 1.3, 3.2, 5.2, 7.1

x(3): 1.9, 1.2, 0.4, -0.3

Closure sum: 88.0

Large factor coefficient: 22

First pairing factor set: 0, 0, 0, 0, 6, 6, 6, 18, 30, 30, 42, 42, 54, 84, 84, 126

Second pairing factor set: 231, 154, 154, 99, 77, 77, 55, 55, 33, 11, 11, 11, 0, 0, 0, 0

Ordered set: 21, 14, 7, 0

Test 7

Steps

1. For the Run title enter **Test 7**
2. In the Initial Values section:
   (a) For **v** enter **9.6**
   (b) For **d** enter **1.23**
   (c) For **k** choose **3**
3. In the Closure section:
   (a) For **Closure type** choose **Epps**
   (b) The **Display large-factor coefficient for the closure?** check box is to be checked
4. In the Output section:
   (a) For **Ordering** choose **Maximal**
   (b) For **Factor sorting** choose **Big/In**
   (c) For **Pair ordering** choose **Desc/Desc**
5. Click the *Submit* button
6. A popup box will appear asking *Are you sure these values are correct?*
7. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 8.6, 17.2

x(1): 1.6, 8.2, 14.8

x(2): 3.2, 7.8, 12.4

Closure sum: 70.0

Large factor coefficient: 23

Ordered set: 12, 14, 17

B.2 Williamsburg Method Test Cases

Test 1

Steps

1. For the Run title enter **Test 1**
2. In the Initial Values section:
   (a) For **v** enter **3.0**
   (b) For **d** enter **3.01**
   (c) For **k** choose **3**
   (d) For **r** choose **1**
3. In the Closure section:
   (a) For **Closure type** choose **Epps**
   (b) For **Rank of the closure coefficient** choose **Even**
4. In the Pairing Factors section:
   (a) Check the **Apply standard pairing factors?** check box
5. In the Output section:
   (a) For **Ordering** choose **Klein**
   (b) For **1st pair ordering** choose **Ascending**
   (c) For **2nd pair ordering** choose **Ascending**
6. Click the *Submit* button
7. A popup box will appear asking *Are you sure these values are correct?*
8. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 2.6, 5.2

x(1): 9.4, 12.1, 14.7

x(2): 18.9, 21.5, 24.1

Closure sum: 105.0

First pairing factor set: 0, 4, 10, 18, 24, 28, 36, 42, 48

Second pairing factor set: 0, 8, 20, 36, 48, 56, 72, 84, 96

Ordered set: 24, 14, 5

Test 2

Steps

1. For the Run title enter **Test 2**
2. In the Initial Values section:
   (a) For **v** enter **5.99**
   (b) For **d** enter **2.3**
   (c) For **k** choose **2**
   (d) For **r** choose **1**
3. In the Closure section:
   (a) For **Closure type** choose **k-aligned**
   (b) For **Rank of the closure coefficient** choose **Odd**
4. In the Pairing Factors section:
   (a) Check the **Apply standard pairing factors?** check box
5. In the Output section:
   (a) For **Ordering** choose **Maximal**
   (b) For **1st pair ordering** choose **Ascending**
   (c) For **2nd pair ordering** choose **Ascending**
6. Click the *Submit* button
7. A popup box will appear asking *Are you sure these values are correct?*
8. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics. Confirm that only the values of $x(n)$ for some n are floats. All other numbers should be integers.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 5.2

x(1): 5.7, 10.9

Closure sum: 10

First pairing factor set: 0, 10, 10, 20

Second pairing factor set: 0, 20, 20, 40

Ordered set: 5, 10

Test 3

Steps

1. For the Run title enter **Test 3**
2. In the Initial Values section:
   (a) For **v** enter **7.01**
   (b) For **d** enter **1.01**
   (c) For **k** choose **3**
   (d) For **r** choose **1**
3. In the Closure section:
   (a) For **Closure type** choose **reverse k-aligned**
   (b) For **Rank of the closure coefficient** choose **Even**
4. In the Output section:
   (a) For **Ordering** choose **Basic**
   (b) For **1st pair ordering** choose **Descending**
   (c) For **2nd pair ordering** choose **Ascending**
5. Click the *Submit* button
6. A popup box will appear asking *Are you sure these values are correct?*
7. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics. Confirm that only the values of $x(n)$ for some n are floats. All other numbers should be integers.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 6.1, 12.2

x(1): 1.4, 7.5, 13.6

x(2): 2.8, 8.9, 15.0

Closure sum: 0

Ordered set: 12, 13, 15

Test 4

Steps

1. For the Run title enter **Test 4**
2. In the Initial Values section:
   (a) For **v** enter **9.90**
   (b) For **d** enter **0.1**
   (c) For **k** choose **3**
   (d) For **r** choose **1**
3. In the Closure section:
   (a) For **Closure type** choose **DeLounge**
   (b) For **Rank of the closure coefficient** choose **Odd**
4. In the Pairing Factors section:
   (a) Check the **Apply standard pairing factors?** check box
5. In the Output section:
   (a) For **Ordering** choose **Minimal**
   (b) For **1st pair ordering** choose **Descending**
   (c) For **2nd pair ordering** choose **Descending**
6. Click the *Submit* button
7. A popup box will appear asking *Are you sure these values are correct?*
8. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics. Confirm that only the values of $x(n)$ for some n are floats. All other numbers should be integers.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 8.6, 17.2

x(1): 0.4, 9.0, 17.6

x(2): 0.8, 9.4, 18.0

Closure sum: 8

First pairing factor set: 36, 34, 34, 18, 18, 16, 0, 0, 0

Second pairing factor set: 72, 68, 68, 36, 36, 32, 0, 0, 0

Ordered set: 18, 17, 17

Test 5

Steps

1. For the Run title enter **Test 5**
2. In the Initial Values section:
   (a) For **v** enter **5.44**
   (b) For **d** enter **3.33**
   (c) For **k** choose **4**
   (d) For **r** choose **1**
3. In the Closure section:
   (a) For **Closure type** choose **Logarithmic**
   (b) For **Rank of the closure coefficient** choose **Even**
4. In the Output section:
   (a) For **Ordering** choose **Maximal**
   (b) For **1st pair ordering** choose **Descending**
   (c) For **2nd pair ordering** choose **Descending**
5. Click the *Submit* button
6. A popup box will appear asking *Are you sure these values are correct?*
7. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics. Confirm that only the values of $x(n)$ for some n are floats. All other numbers should be integers.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 4.7, 9.5, 14.2

x(1): 11.5, 16.2, 21.0, 25.7

x(2): 22.9, 27.7, 32.4, 37.2

x(3): 34.4, 39.2, 43.9, 48.6

Closure sum: (blank)

Ordered set: 14, 25, 37, 48

Test 6

Steps

Check that the following fields and types are correct and present:

1. The title, *Williamsburg Method* is at the top left of the screen, just below two links labeled *Previous* and *Next*.
2. Below the title is a text field labeled *Run title*
3. In the green Initial Values section, there is a text field labeled v, a text field labeled d, and a drop box labeled k. There is also a set of radio buttons labeled r, with each radio button being labeled *-1*, *0*, *1*, from left to right. Finally there is a check box labeled *Use Hadwiger sets?*
4. In the green Closure section, there is a drop box labeled *Closure type*, and a set of radio buttons labeled *Rank of the closure coefficient*. The radio buttons are labeled *Even* and *Odd*, from left to right.
5. In the green Pairing Factors section there is a check box labeled *Apply standard pairing factors?*.
6. In the green Output section, there are three sets of radio buttons. The first set is labeled *Ordering* and has the choices *Maximal*, *Minimal*, *Klein*, and *Basic*, from left to right. The second set is labeled *1st pair ordering* and has the choices *Ascending* and *Descending*, from left to right. The third set is labeled *2nd pair ordering* and has the choices *Ascending* and *Descending*, from left to right.
7. Below the Output section is a single button labeled *Submit*. It is centered in relation to the middle of the Output section.

Expected Output

None

Test 7

Steps

1. For the Run title enter **Test 7**
2. In the Initial Values section:
   (a) For **v** enter **7.0**
   (b) For **d** enter **5.0**
   (c) For **k** choose **2**
   (d) For **r** choose **1**
3. In the Closure section:
   (a) For **Closure type** choose **Epps**
   (b) For **Rank of the closure coefficient** choose **Odd**
4. In the Pairing Factors section:
   (a) Check the **Apply standard pairing factors?** check box
5. In the Output section:
   (a) For **Ordering** choose **Maximal**
   (b) For **1st pair ordering** choose **Ascending**
   (c) For **2nd pair ordering** choose **Ascending**
6. Click the *Submit* button
7. A popup box will appear asking *Are you sure these values are correct?*
8. Click *OK*

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics. Confirm that only the values of $x(n)$ for some n are floats. All other numbers should be integers.

Check the expected output below against that which appeared on the screen:

x(0): 0.0, 6.1

x(1): 25.4, 31.5

Closure sum: 62

First pairing factor set: 0, 12, 50, 62

Second pairing factor set: 0, 24, 100, 124

Ordered set: 6, 31

B.3 Module C1 Test Cases

Test 1

Steps

Check that the following fields and types are correct and present:

1. The title, *Module C1 - Cross-Correlated Pappus Design Generator* is at the top left of the screen, just below two links labeled *Previous* and *Next*.
2. Below the title is a text field labeled *Run title*
3. In the green Initial Values section, there is a text field labeled *d*; a check box labeled *Apply generalization skews?*; a check box labeled *Make design triangular?*; and a drop box labeled *Field*
4. Below the Initial Values section is a single button labeled *Submit*. It is centered in relation to the middle of the Initial Values section.

Expected Output

None

Test 2

Steps

1. For the Run title enter **Test 2**
2. In the Initial Values section:
   (a) For **d** enter **3**
   (b) Check the **Apply generalization skews?** check box
   (c) Check the **Make design triangular?** check box
   (d) For **Field** choose **Projective**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: 14, 36, 47, 69, 80, 102, 113

Minor design: 11, 14, 20, 23, 29, 32, 38

Triangular design: 0, 2

Projective application result: 3, 4, 5, 6, 8, 10

Test 3

Steps

1. For the Run title enter **Test 3**
2. In the Initial Values section:
   (a) For **d** enter **6**
   (b) For **Field** choose **Near Normal**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: 6, 18, 30, 42, 54, 66, 78, 90, 102, 114, 126, 138

Minor design: 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78

Near Normal application result: 12

Test 4

Steps

1. For the Run title enter **Test 4**
2. In the Initial Values section:
   (a) For **d** enter **12**
   (b) Check the **Apply generalization skews?** check box
   (c) For **Field** choose **Quasi**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: 28, 60, 108, 124, 156, 172, 220, 252

Minor design: 16, 40, 76, 88, 112, 124, 160, 184

Quasi application result: -4, -1, 0, 2, 10

B.4 Module D1 Test Cases

Test 1

Steps

Check that the following fields and types are correct and present:

1. The title, *Module D1 - Aperiodic-Correlated Pappus Design Generator* is at the top left of the screen, just below two links labeled *Previous* and *Next*.
2. Below the title is a text field labeled *Run title*
3. In the green Initial Values section, there is a text field labeled *m*; a drop box labeled *Base Squares*; and a check box labeled *Make design directed?*.
4. Below the Initial Values section is a single button labeled *Submit*. It is centered in relation to the middle of the Initial Values section.

Expected Output

None

Test 2

Steps

1. For the Run title enter **Test 2**
2. In the Initial Values section:
   (a) For **m** enter **5**
   (b) For **Base Squares** choose **Turan**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: -4, 0, 2, 3

Minor design: 0, 3, 5, 7

Test 3

Steps

1. For the Run title enter **Test 3**
2. In the Initial Values section:
   (a) For **m** enter **7**
   (b) For **Base Squares** choose **Tuscan**
   (c) Check the **Make design directed?** check box
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: -6, -2, 0, 5

Minor design: 0, 2, 3, 4, 5, 6

Test 4

Steps

1. For the Run title enter **Test 4**
2. In the Initial Values section:
   (a) For **m** enter **12**
   (b) For **Base Squares** choose **reverse k-aligned**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: -13, -9, -6, -5, -1, 0, 2, 4, 13, 19

Minor design: 0, 1

B.5 Module C2 Test Cases

Test 1

Steps

Check that the following fields and types are correct and present:

1. The title, *Module C2 - Cross-Correlated Pappus Design Generator* is at the top left of the screen, just below two links labeled *Previous* and *Next*.
2. Below the title is a text field labeled *Run title*
3. In the green Initial Values section, there is a text field labeled *m*; a drop box labeled *Base Squares*; and a check box labeled *Make design directed?*.
4. Below the Initial Values section is a single button labeled *Submit*. It is centered in relation to the middle of the Initial Values section.

Expected Output

None

Test 2

Steps

1. For the Run title enter **Test 2**
2. In the Initial Values section:
   (a) For **m** enter **5**
   (b) For **Base Squares** choose **Turan**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: -3, 0, 4, 5

Minor design: 0, 4, 9, 16

Test 3

Steps

1. For the Run title enter **Test 3**
2. In the Initial Values section:
   (a) For **m** enter **7**
   (b) For **Base Squares** choose **Tuscan**
   (c) Check the **Make design directed?** check box
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: -6, -2, 0, 5

Minor design: 0, 2, 3, 4, 5, 6

Test 4

Steps

1. For the Run title enter **Test 4**
2. In the Initial Values section:
   (a) For **m** enter **12**
   (b) For **Base Squares** choose **reverse k-aligned**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a green box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: -13, -9, -6, -5, -1, 0, 2, 4, 13, 19

Minor design: 0, 1

B.6 Module D2 Test Cases

Test 1

Steps

Check that the following fields and types are correct and present:

1. The title, *Module D2 - Aperiodic-Correlated Pappus Design Generator* is at the top left of the screen, just below two links labeled *Previous* and *Next*.
2. Below the title is a text field labeled *Run title*
3. In the green Initial Values section, there is a text field labeled *d*; a check box labeled *Apply generalization skews?*; a check box labeled *Make design triangular?*; and a drop box labeled *Field*
4. Below the Initial Values section is a single button labeled *Submit*. It is centered in relation to the middle of the Initial Values section.

Expected Output

None

Test 2

Steps

1. For the Run title enter **Test 2**
2. In the Initial Values section:
   (a) For **d** enter **3**
   (b) Check the **Apply generalization skews?** check box
   (c) Check the **Make design triangular?** check box
   (d) For **Field** choose **Projective**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: 15, 35, 48, 68, 81, 101, 114

Minor design: 11, 15, 25, 31, 38, 42

Triangular design: 0, 3

Projective application result: 2, 4, 5, 6, 7, 11

Test 3

Steps

1. For the Run title enter **Test 3**
2. In the Initial Values section:
   (a) For **d** enter **6**
   (b) For **Field** choose **Near Normal**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: 6, 18, 30, 42, 54, 66, 78, 90, 102, 114, 126, 138

Minor design: 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78

Near Normal application result: 12

Test 4

Steps

1. For the Run title enter **Test 4**
2. In the Initial Values section:
   (a) For **d** enter **12**
   (b) Check the **Apply generalization skews?** check box
   (c) For **Field** choose **Quasi**
3. Click the *Submit* button

Expected Output

The form will be reset. To the right, a box will appear labeled Results. Within the results box, each result label is blue and in italics.

Check the expected output below against that which appeared on the screen:

Major design: 28, 60, 108, 124, 156, 172, 220, 252

Minor design: 16, 40, 76, 88, 112, 124, 160, 184

Quasi application result: -4, -1, 0, 2, 10

References

[1] J. Aranda and S. Easterbrook. Anchoring and adjustment in software estimation. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 346–355, New York, NY, USA, 2005. ACM.

[2] D. Ariely. *Predictabley Irrational: The Hidden Forces that Shape Our Decisions*. HarperCollins, first revised and expanded edition, 2009.

[3] D. Ariely, G. Loewenstein, and D. Prelec. Coherent arbitrariness: Stable demand curves without stable preferences. *Quarterly Journal of Economics*, 118(1):73, 2003.

[4] F. Bacon. *Novum Organum*. London:Longmans, 1858. Translated from the Latin by James Spedding et al. Originally publised in *Instauratio Magna* in 1620.

[5] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, second edition, 1990.

[6] P. Blecha. *Taboo Tunes: A History of Banned Bands & Censored Songs*. Backbeat Books, San Francisco, 2004.

[7] E. Davis. *Led Zeppelin IV*. In D. Barker, editor, *33 1/3 Greatest Hits*, volume 1, pages 201–213. Continuum International Publishing Group, 2005.

[8] B. L. Fredrickson. Extracting meaning from past affective experiences: The importance of peaks, ends, and specific emotions. *Cognition Emotion*, 14(4):577 – 606, 2000.

[9] B. L. Fredrickson and D. Kahneman. Duration neglect in retrospective evaluations of affective episodes. *Journal of Personality and Social Psychology*, 65(1):45–55, 1993.

[10] A. Gill. Led zeppelin: A complete guide to the band's studio albums. The Independent, December 7, 2007.

[11] J. Grimes. On the failure to detect changes in scences across saccades. *Perception (Vancouver Studies in Cognitive Science)*, 2:89–109, 1996.

[12] S. Guilford, G. Rugg, and N. Scott. Pleasure and pain: Perceptual bias and its implications for software engineering. *IEEE Software*, 19(3):63–69, 2002.

[13] D. Kahneman. The riddle of experience vs. memory. Presented at TED2010, Long Beach, California, 2010.

[14] D. Kahneman, B. L. Fredrickson, C. A. Schreiber, and D. A. Redelmeier. When more pain is preferred to less: Adding a better end. *Psychological Science*, 4(6):401–405, November 1993.

[15] D. Kahneman, P. P. Wakker, and R. Sarin. Back to bentham? explorations of experienced utility. *Quarterly Journal of Economics*, 112(2):375–405, 1997.

[16] P. Kamde, V. Nandavadekar, and R. Pawar. Value of test cases in software testing. In *Management of Innovation and Technology, 2006 IEEE International Conference on*, volume 2, pages 668 –672, 21-23 2006.

[17] C. Kaner. What is a good test case? In *Software Testing Analysis Review Conference (STAR) East*, Orlando, FL, May 12-16, 2003 2003.

[18] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software*. Wiley, second edition, 1999.

[19] D. Levin and D. Simons. Failure to detect changes to attended objects in motion pictures. *Psychonomic Bulletin and Review*, 4(4):501–506, 1997.

[20] A. Mack and I. Rock. *Inattentional Blindness*. Bradford Books Series in Cognitive Psychology. MIT Press, 1998.

[21] U. Neisser and R. Becklen. Selective looking: Attending to visually specified events. *Cognitive Psychology*, 7(4):480 – 494, 1975.

[22] J. K. O'Regan, R. A. Rensink, and J. J. Clark. Change-blindness as a result of "mudsplashes". *Nature*, 398(6722):34–34, 1999.

[23] K. R. Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, fifth edition, 1989.

[24] D. A. Redelmeier and D. Kahneman. Patients' memories of painful medical treatments: real-time and retrospective evaluations of two minimally invasive procedures. *Pain*, 66(1):3 – 8, 1996.

[25] D. A. Redelmeier, J. Katz, and D. Kahneman. Memories of colonoscopy: a randomized trial. *Pain*, 104(1-2):187 – 194, 2003.

[26] R. Rosenthal. *Experimenter Effects in Behavioral Research*. Century Psychology Series. Irvington Publishers, enlarged edition, 1976.

[27] R. Rosenthal and R. L. Rosnow, editors. *Artifact in behavioral research*. Academic Press, 1969.

[28] R. Rosenthal and R. L. Rosnow. *The Volunteer Subject*. Wiley series on personality processes. Wiley, 1975.

[29] K. Shadwick. *Led Zeppelin: The story of a band and their music 1969-1980*. Backbeat Books, 2005.

[30] M. Shermer. Turn me on, dead man. *Scientific American*, 292(5):37, May 2005.

[31] D. J. Simons and C. F. Chabris. Gorillas in our midst: Sustained inattentional blindness for dynamic events. *Perception*, 28:1059–1074, 1999.

[32] D. J. Simons, S. L. Franconeri, and R. R. L. Change blindness in the absence of a visual disruption. *Perception*, 29(10):1143–1154, 2000.

[33] D. J. Simons and D. T. Levin. Change blindness. *Trends in Cognitive Sciences*, 1(7):261 – 267, 1997.

[34] D. J. Simons and D. T. Levin. Failure to detect changes to people during a real-world interaction. *Psychonomic Bulletin and Review*, 5:644–649, 1998.

[35] W. Stacy and J. MacMillian. Cognitive bias in software engineering. *Commun. ACM*, 38(6):57–63, 1995.

[36] A. Tversky and D. Kahneman. Judgment under uncertainty: Heuristics and biases. *Science*, 185(4157):1124–1131, 1974.

[37] T. Yamaura. How to design practical test cases. *IEEE Software*, 15(6):30–36, Nov/Dec 1998.