Common Cognitive Heuristics and Biases in Judgment: How They Impact the Activities of the Software Development Life Cycle

Timothy Adams

University of South Florida

The Challenging Nature of Software Development

According to Gartner, global spending on Information Technology (IT) is projected to reach $3.7 trillion in 2013. $300 billion of that will be on Enterprise Software. With such a large amount of capital invested in IT, it is distressing that recent surveys suggest that only 12.5% of all software development projects are delivered on time. It is more distressing that these projects typically only deliver between 25% and 75% of their planned functionality. In fact, the Standish Group estimates that in 2009 a full 24% of software projects were either canceled prior to completion or completed but then never used. Other statistics concerning the outcomes of software development projects prove equally grim: 47% of projects experience higher than expected maintenance costs, 80% of development costs are spent identifying and correcting defects, 60% - 80% of software project failures can be attributed to poor requirements gathering, analysis, and management, and 40% of problems are found by end users. Focusing on these negative statistics may neglect the that fact that there are a lot of successful software products in use and also a lot of successful software projects, but it does so to highlight the existence of a real problem in software development. From beginning to end IT has trouble identifying the proper software to build and then building it effectively and efficiently.

There are a number of very real challenges to developing good software: incomplete and changing requirements, evolving technical landscapes, and shifting business environments, not to mention unrealistic deadlines, poor expectation management, and faulty communication. Software development is an essentially complex task. While, as Fred Brooks said, there are no silver bullets to eliminate this complexity, there are some principles and tools to help manage it. A core principle in managing the complexity found in software engineering is intellectual control: the ability to create and maintain a mental model of the entirety of the project in one's mind. Even a simple project has enough details to make this task impossible, which is why a second principle, abstraction, exists. Abstraction allows one to focus only on the aspects of an object that are important to the task at hand, ignoring irrelevant details.

Some useful standard abstractions have been created to benefit the development of software systems with the first such abstraction encountered in a development project being the software development life cycle itself.

The software development life cycle (SDLC) is best thought of as a set of phases defining a process for planning and controlling the creation of a software system. While there are a variety of implementations of the SDLC, they all allocate time to planning and analysis, design and implementation, and testing and verification. By providing a high level view of the activities essential to the project, the SDLC facilitates intellectual control over the development process. Unfortunately, abstracting away the complexity of a process does not remove that complexity, it only defers it until later, and so planning and analysis soon becomes identifying stakeholders and soliciting stake holder needs which leads to assessing those needs for feasibility and testability before prioritizing them and turning them into specifications. Each step leads to another and as the layers of abstraction fall away, the details emerge, and what is left begins to resemble a fundamentally human process. Each phase of the SDLC involves different groups of people and each of these groups has tasks and responsibilities essential to the successful completion of the project. More importantly these tasks and responsibilities require that the members of these groups, in all of their different roles and capacities, exercise judgment and make decisions that will affect the project in its entirety.

Cognitive psychology has developed a body of knowledge about how people process information: how they think and make decisions. Cognitive scientists have found that while people typically make reasonable decisions, there are demonstrable ways that intuitive thinking takes short cuts, leading to predictable errors in judgment, commonly referred to as cognitive biases. These biases are fundamental to the way the brain processes information and therefore are common to everyone. The nefarious quality of them is that they are subconscious: people experience them every day and are generally unaware of it.

Examining the production of software systems through each phase of the software development

life cycle and identifying the actors and activities common to them develops a point of view where the importance of the human element in the process can be understood. While the alarming 24% failure rate for software projects may initially bring to mind broken code and overworked programmers, the statistics actually point to problems throughout the SDLC that are not limited to strict technical factors. Applying some principles from cognitive psychology about how people make judgments and how those judgments can be biased, a model can be developed that explains some of the human factors that contribute to the difficulty in producing software. This model will identify the areas where people are likely to exercise poor judgment and provide a common framework for discussing them. By enabling an informed discussion of this topic among the actors involved in the SDLC some of these pitfalls may be avoided. At the very least, raising the level of discourse on this subject can help create a more realistic set of shared expectations for the outcomes of the activities of the SDLC. Perhaps the discussion of pitfalls and expectations will eventually yield best practices to mitigate some of the negative effects of cognitive biases.

Software impacts everyone, from the executive deciding on a new project to the end user consuming it, from the system analyst collecting a requirement to the quality assurance engineer testing it, and from the architect designing a system to the developer implementing it. Because all of these players also impact the software being produced, each should seek to develop an understanding of the role that heuristics and cognitive biases play in decision making and how that affects the quality of the software being produced. Building an understanding of these principles into the members of this community will allow them to make better decisions both as they become more informed on this topic individually and as they interact with and expect to be evaluated by a peer group that is better informed collectively.

Common Cognitive Heuristics and Biased Judgments

Up through the 1970's it was a commonly held view among social scientists that human nature is rational. People were generally assumed to make sound decisions based on logic and reasoning. However, in 1974, two researchers, Amos Tversky, and Daniel Kahneman published an article, "Judgment Under Uncertainty: Heuristics and Biases", that explored the nature of intuitive thinking. They found that when people were asked to assess the likelihood of uncertain events, rather than applying purely logic based mathematical reasoning, they instead turned to a limited number of heuristics. These heuristics are the short-cuts of intuitive thinking: the simple rules people use to solve problems and perform tasks that are substituted for the more complex problem. Because they are short-cuts, these heuristics provide adequate, but often flawed, answers to the original problem, resulting in biased judgments.

The following example will help illustrate the point. Suppose one tosses a fair coin. It is common knowledge that there are two possible outcomes, heads (H) and tails (T), and that each is equally likely. Now, suppose one tosses that coin repeatedly, say one hundred times in a row, and records the results of each try. It is again common knowledge that this will result in approximately the same number of heads and tails, about fifty-fifty. Finally, consider the following three sequences of six coin flips:

H – H – H – T – T – T
T – T – T – T – T – T
H – T – H – H – T – H

Without calculating the actual probability, which one seems more likely to occur? Most of the respondents in this experiment chose the third sequence (H-T-H-H-T-H) when, in reality, each of these sequences is equally probable. In fact, since each coin toss is independent of the last, any sequence of six coin tosses is as likely as any other. So, why does it seem like this last sequence is more probable?

The answer is that when asked about these probabilities people apply a heuristic known as *representativeness*. The process works as follows: since considering the probability of the sequences is complex enough that it requires the application of some logic and reasoning, one does not have an

immediate answer to the question. What one does have is an intuitive thought process that is very good at answering questions very quickly. In this instance, the intuitive system immediately knows that coin flips are random and that each outcome is equally likely, and so it considers the options given in relation to these facts. In the example above, the first sequence (H-H-H-T-T-T) has an equal number of heads and tails but is ordered in a way that does not appear random. The second sequence (T-T-T-T-T-T) has neither an equal number of outcomes nor a random order. However, the third sequence (H-T-H-H-T-H) satisfies both criteria: an equal number of heads and tails arranged in a way that generally agrees with what one thinks of as random. Consequently, while the logical mind is taking a moment to consider the question, the intuitive thought process has already decided that option three is the answer because it is the sequence that is most representative of the qualities that one expects to see from a large number of coin tosses.

Unfortunately the intuitive answer is wrong in this instance. While the representativeness heuristic can be helpful, it leads to a number of cognitive biases. The above example demonstrates something called the *law of small numbers*, where one expects the global attributes of a system to be represented locally, regardless of the size of the sample being considered. This has been shown to lead to an overvaluing of the content of a message in relation to the information about its validity. In short the law of small numbers leads people to accept the information reported to them as more accurate than the supporting evidence may indicate. Representativeness causes other biases as well, but at their core, they occur when people substitute an estimate of how representative or typical of what they expected a result is for an estimate of the actual likelihood of that result, and they typically occur when people are asked to assess how likely it is that an object or event belongs to a given class of objects or events.

Of course, people are required to make many other types of judgments as well. Sometimes one needs to estimate the frequency of an event, perhaps the number of shark attacks per year. In the absence of having memorized that statistic, one will need to estimate it. Experiments have shown that when

estimating the frequency of an event, people start by creating a mental list of all the instances of that event that they can remember. Their final estimates will be based on how easily those instances are remembered. This is termed the *availability* heuristic. It works well in the general sense since events that occur more frequently will tend to have more instances that are easily remembered. However, it has an associated bias. Remember, one is substituting how easy it is to recall events for an estimate of the frequency of those events. There is no intuitive accounting of the reasons that the events may be easy to recall. Returning to the example of shark attacks, while they occur relatively infrequently, they tend to be memorable because they are widely, and often sensationally, reported. So, had an attack occurred recently or perhaps at a local beach, it would be easily recalled, causing one to overestimate the occurrence of all attacks. This is the fundamental nature of biases of availability: events that are vivid, emotional, particularly salient, or recent are easier to recall, causing people to overstate the frequency of their occurrence.

In addition to representativeness and availability, one last heuristic needs to be discussed. It is one of the most robust and reliable phenomena in experimental psychology: when people are given a problem to solve and an initial estimate for that solution, their final answers will be close to the original estimate. This is called the *anchoring* effect and it has been demonstrated experimentally across a wide variety of domains. In one example, participants in a study were shown two equations:

```
8 x 7 x 6 x 5 x 4 x 3 x 2 x 1
1 x 2 x 3 x 4 x 5 x 6 x 7 x 8
```

When asked to estimate the solutions to these problems, the answer to the first equation was consistently estimated higher than the answer to the second. Although the product of each of these series of numbers is obviously the same, when not given time to calculate them in their entirety participants are left guessing based on their initial estimates. Since the top equation starts with the higher numbers, subsequent estimates are higher. The bottom equation starts with lower number and the resultant estimates reflect

that. As with the other heuristics, anchoring helps people simplify complex problems. When given an initial estimate, the intuitive system uses this as a starting point from which to find a solution. If this initial estimate is reasonable, the final estimate is likely to be good, since people tend to stay close to the initial estimates. Unfortunately, if this initial estimate is flawed, then the final solution is also going to be flawed due to inadequate adjustment. This is termed *adjustment bias*.

Looking at these heuristics and the way they affect judgments, it is clear that people are not quite as rational as initially thought. The cognitive process has some short cuts built into it that lead to cognitive biases that are related to the ways that humans store and use information. Many cognitive scientists believe that people create mental models to solve problems and understand situations and that cognitive biases derive from the way those models are developed and used. Examining the mental models created during the software development process to manage its complexity, and considering the heuristics applied when making certain judgments will develop a model for how these biases affect software development projects. Table 1 describes some common biases.

Table 1: Heuristics and Biases

Heuristic & Biases	Description
Representativeness	Estimating the odds of an event by how representative it is of a typical event in its class.
Law of Small Numbers	Overvaluing the content of a message without regard to the evidence supporting it.
Availability	Estimating the frequency of an event by how "available" that event is in memory.
Anchoring	Estimating starting from an initial value.
Adjustment Bias	Insufficient movement away from the anchor when revising estimates.
Planning Fallacy	Estimating plans close to best case scenarios without considering the outcomes of similar plans.
Overconfidence	Confidence in an outcome is higher than objective circumstances warrant.
Confirmation Bias	Predilection to confirm ideas rather than try to disprove them.

The Software Development Life Cycle

The software development life cycle (SDLC) is a series of steps, or phases, that provide a model for managing the development of a piece of software. It is the first step towards delivering a working system that is, theoretically at least, on time, on budget, and in agreement with customer expectations.

There are a variety of specific methods employed to develop software, and each models the SDLC in a way that reflects the particular values of the given method. Still, regardless of the specific implementation, all models agree on some of the general activities involved in developing a piece of software: requirements gathering, design, implementation, testing, and deployment.

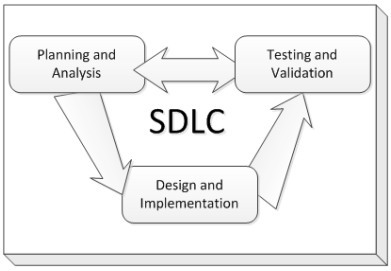

Fig. 1 – The Software Development Life Cycle

Figure 1 illustrates a high level view of the development process prior to deployment. It partitions the SDLC into three phases: planning and analysis, design and implementation, and testing and validation. Planning and analysis precede design and implementation, but coincide with some testing and validation activities. Design and implementation precede testing and validation, although in truth testing is involved here as well. With the popularity and effectiveness of incremental development, the entire process can be repeated again after testing and validation verify the correctness of the current increment.

A software development project starts with planning and analysis. The primary purpose of this phase is to produce and agree to a specification for the system that is to be built. This typically starts with system analysts identifying and gathering all of the stakeholders in the project. These stakeholders will then be asked what they require from the proposed system. Requirements will be of two types: functional requirements that define the inputs and outputs of the system, and quality requirements that define the aspects of the system related to the level of service it should provide. Once the requirements are identified they are reviewed for feasibility and testability. The requirements that are considered to be realistic are then grouped together in a way that defines the actual system to be built. This is the specification, and once it is complete, and all parties agree to it, the next phase of the SDLC can begin.

Following the agreement to the specification by the project stakeholders, design and implementation can begin. This phase of the SDLC is where most of the technical work is done. Design can be broken down into two parts. The first is a high level design. Here systems architects will review the specification to determine the scope of the project. Requirements will be grouped together by common features and functions. These groups begin to define the modules for the system. The number of development increments needed to complete the system the contents of those increments will be planned.

Once the overall design of the system is established, work on the details will begin. Software engineers will look at the abstract modules that are needed, and begin planning their implementations. These modules can be implemented in different ways. Some may be purchased from vendors. Others might be developed from scratch. Still other modules might already exist and be available for integration. All of these options will be considered before arriving at a final solution.

Finally, with design complete, implementation begins. Here programmers assemble the software based on all of the previous design work. This can mean coding new software modules, adapting existing modules, consuming external services, or integrating the commercial off-the-shelf software (COTS). These programmers will also be unit testing their work as they go along, making sure all of the methods function as expected.

While testing has been listed last and as its own phase, testing should be an integral part of the whole process. It can start the same time that the requirements are gathered as a way to validate that all requirements are testable and correct. Test scripts can be written throughout the design phase to verify the correctness of the design. As mentioned, code should be unit tested throughout implementation. So, this final standalone testing phase will be dedicated to quality assurance engineers running integration tests, ensuring that all the modules work well together, and functional system tests, ensuring that the end product does what it is expected to do.

While this has only been a general overview of some of the activities that occur in the development of a software system, it should illustrate just how many people are involved in creating software, and how dependent on one another they all are. In describing the general work flow, what happens now, what happens next, it provides a context to understand the importance each step and evaluate the dependencies that exist throughout the cycle. From this abstract model, the complexities and individual activities of each of these phases can be explored.

Table 2: Software Development Life Cycle (SDLC)

Phase	Description	Actors	Activities
Planning and Analysis	The project is defined and agreed to.	Systems analysts, project stakeholders, experts	Requirements gathering and production of a specification. Estimation of effort.
Design and Implementation	The project is designed per the specification and coded.	Systems architects, software engineers, software developers	System architecture and identification of modules. Strategy to implement modules. Coding and unit testing.
Testing and Validation	The correctness of the project is verified.	Quality assurance analysts, software testers.	Systems integration testing. Functional systems testing.

Requirements Gathering

As the primary goal of planning and analysis, it has been argued that information requirements determination is the most important part of systems development. Since accurately knowing what the users need to do and what their expectations are is a key determinant to system success, it is easy to see why. Regardless of anything else: how well the system works, how quickly it was deployed, or how under budget it came in, if it does not meet the needs of the users then it will not be used, and if it is not used then it has to be considered a failure.

The general methodology systems analysts follow to determine user needs involves three phases. First, information is gathered, either through observation of the business process, the prior knowledge and experience of the analyst, or interviews with stakeholders and system users. Next, the results of this information gathering are summarized in a document or set of documents. Used to verify the correctness

of the requirements, these document are revised and refined until they can be agreed to as final. In the last step, the stakeholders sign off on the documents. This process is often iterative, requiring several cycles of information gathering, documenting, and refinement before the final agreement.

During requirements gathering, the system analyst must solicit requirements from the users of the system. A typical enterprise software system will have different classes of users that each need to interact with the system in specific ways. These classes will contain a lot of different individual users. As an exhaustive investigation of each user's needs would be impossible, the analyst will need to choose subsets of the users of each class to act as proxies for the whole of the class. Two critical decisions that need to be made here are how many users to interview, and which users those should be. These type of decisions are susceptible to biases of representativeness. Where the analyst really needs to collect requirements typical of the user base, these will not always equate to the requirements of the typical user. Relying on personal judgment, the analyst deciding to interview users is likely to form a mental model of the user she considers typical of the class of users she is targeting. The analyst is also likely to overestimate the validity of the requirements solicited from that person, considering them to be more representative of the entire user base than may be merited. For example, a more advanced user may not need a specific helper function to look up codes relevant to the task they are performing, where a less experienced user will absolutely need that function. So, biases of representativeness can lead to incomplete requirements through improper sampling of the user base, and over reliance on the information gathered from that sample.

While accurately sampling the user base is a prerequisite to developing a complete set of requirements, it is only the first step. Soliciting good requirements from that sample presents its own challenge. To create a useful system, the analyst needs to document all the steps necessary to complete the business processes the system needs to perform. The more routine these steps are and the more frequently they occur, the more important they are going to be. Unfortunately, human memory is not

well suited to remembering a single routine occurrence within a set of routine occurrences. Through a process known as automaticity, habitual tasks can become automatic. Individual perform them with almost no thought or effort given to their actions. This poses an obvious initial obstacle for the user being interviewed about the typical activities they perform. Overcoming that obstacle is made more difficult by biases of availability. When trying to make a list of the most frequent, and subsequently most important, actions, the user is going to substitute ease of recall for actual frequency, overstating the frequency and importance of those actions that have been complete recently or that are memorable for reasons other than being common. So, where the systems analyst is looking to create a list of the most common, frequent, important requirements, the user is likely to omit some very common tasks, and to overstate the frequency and importance of very recent and novel tasks.

Both of these problems assume that all of the requirements are known in advance, that the user base knows all of the functionality required from the new system, and that the problem is simply in figuring out what those are. However, in reality, requirements are emergent. They are never entirely known at the beginning. New requirements are added as the development process progresses, and previously established requirements are subject to change. In managing these changes, the systems analyst is subject to the anchoring effect and adjustment bias. While anchoring was presented mathematically before, it is not limited to numeric estimates. The existing requirements documents or systems specifications can act as an initial estimate of the final system. When presented with changing requirements, the analyst is going to return to this initial estimate and adjust it to meet the new need. This will not be problematic for small changes, but if the initial understanding was wrong, or the new requirement requires a significant change, there is a danger that the analyst will not adjust far enough from that initial requirement to sufficiently capture the nature of the requested changes. The more the system needs to be changed, the more likely it is that the revised requirements will not incorporate the new functionality simply due to adjustment bias, and anchoring on the original view of the system.

As the first step in the development of a new system, the requirements gathered and the specification produced from them are the foundation for the project. If they are incomplete then the system will not meet the users' needs. Even though changes to the requirements should be expected, knowing that the original requirements create anchors, it is essential that the initial specifications are as correct as possible.

Estimating Effort

A complete specification will determine the scope of the project, and from this the amount of effort required to build the system can be estimated. An accurate estimation of effort is essential to the efficient allocation of resources to the project and drives the budget and delivery date. So, to keep the system on time and on budget, these estimates need to be right.

Expert estimation is a common method employed to determine software development effort. A broad definition for expert estimation is any estimate performed by someone who has considerable knowledge of the domain, where a portion of the estimate is based on the experience of the estimator. This does not preclude the use of tools or measures but does require the expert to exercise judgment in weighing the information derived from them.

While expert estimation can be quite accurate, studies show that a handful of cognitive biases have the potential to make it inaccurate. The first problem with any estimation is something called the *planning fallacy*. This term was coined by Kahneman and Tversky to describe plans or forecasts that are unrealistically close to best-case scenarios and that could have been improved by considering the outcomes of similar cases. The planning fallacy occurs in every domain, causing people to be overly optimistic about the likelihood of success of the projects they are planning and the amount of effort they will require. Applied to the development of software systems, the planning fallacy leads to underestimating the amount of effort, time, and resources required to complete a project.

In the domain of software development, the effects of the planning fallacy have been demonstrated to increase with the amount of control the individual making the estimate has over the project. So, a developer responsible for implementing a software module will be more optimistic in his estimate of time than a project manager estimating the amount of time it will require for that developer to implement the module. This appears to be related to the mental models that each individual constructs when making the estimate. Relying on memory and experience, the individual responsible for the task is likely to take an "inside" perspective, concentrating on case specific planning and a neglecting other data, such as completion times of similar projects. The individual with less direct control will consider other similar tasks that have been completed and model his estimate based on how much effort those required. This latter model is not without its own drawback since identifying similar projects is a judgment vulnerable to the availability bias. The list of similar projects may only reflect the most recent projects with which that person is familiar, neglecting some more appropriate yet less easily remembered past events.

Additional research into the planning fallacy shows that time estimates are related to the stimuli they are estimating, with the duration of short stimuli being overestimated, and the duration of long stimuli being underestimated. Framing this in terms of software development, long stimuli can be considered equivalent to complex modules of code while short stimuli might be the functions that compose that module. Evidence is emerging that unpacking larger projects, breaking them down into smaller parts, through a process known as segmentation will change the overall estimate. Breaking the tasks down introduces an overestimation of individual task length, which when summed, leads to a larger estimate that that given for the project as a whole.

In addition to predicting the amount of effort required to develop a software project, the accuracy of that prediction must be assessed. Here the *overconfidence effect*, where people are more confident in their judgments than the objective accuracy of those judgments merits, is introduced. Studies show that

the amount of confidence that experts assign to their estimates is directly related to the amount of time spent on those estimates but, unfortunately, rather unrelated to their actual accuracy. In fact, the research demonstrated that there is not much correlation between one's ability in a subject and one's ability to estimate it, nor is there much correlation between one's ability to estimate something and one's ability to judge the accuracy of that estimate. These appear to be three distinct things: knowledge of a subject, the ability to estimate a subject, and the ability to determination the accuracy of estimates.

An important part of project planning is projecting a time line for a project and determining the budget. Both of these tasks rely not only on having an accurate specification to scope the project, but also on accurately determining the amount of effort that will be required to realize that specification. These estimates of effort are sensitive to cognitive biases, and likely to be quite different depending on who is performing the estimate, the amount of control he has over the artifact he is estimating, and the level of detail at which that artifact is presented. Not only that, but the estimator is likely to be much more confident in his estimates than accuracy merits. Overly optimistic schedules or insufficient resource allocation will lead to missed release dates and missed budget figures, likely ending with a compromised product and unhappy client.

Reusing Software Artifacts

An important part of the design and implementation process is identifying common areas of functionality in the specification and building them as modules. This helps control the complexity of the software by building the system, piece by piece, out of components. A strength of this design philosophy is that components from other systems can be re-used when they provide the same or similar functionality to the module identified in the specification. This should make development more efficient since existing code that has already been verified can be integrated into the new systems. Unfortunately, not every common component can be re-used as it currently exists. Often a module has functionality that is close

enough to the identified functionality that it can be used but only after some modifications. While using existing code in these circumstances has advantages, deciding how to re-use it requires making some judgments that are susceptible to anchoring and adjustment biases.

Since every software artifact is developed in a specific context, the proper re-use of that artifact will require examining it for correctness, or how well the existing object meets the current requirements. There are two ways for the existing object to be incorrect in the new environment. It can omit behavior specified by the new requirements, or it can contain additional behavior not included in the new requirements. While missing behavior should always be added, handling extraneous behavior requires more nuance. Any behavior that hinders the new system for any reason should be excluded, but behaviors that do not interfere with the new system and could potentially benefit it might legitimately be included. Either way, the anchor will be the original software artifact, and adjustment bias will result in a final software artifact that does not reflect the current set of requirements either by omitting required functionality or including extra functionality.

Experiments which examined both experienced and novice programmers showed that each group exhibited anchoring and adjustment bias to some extent when given a set of requirements and an existing piece of code that left out important requirements but included extraneous functionality. Novice developers exhibited the most adjustment bias by neglecting to add all of the necessary code to their final module and also including the extraneous functionality. The experienced programmers fared better by including all of the new requirements in their final design. However, they still included the extraneous functionality. How much of this was intentional, extra functionality they actively decided to include to benefit to the new system, is still unclear. Regardless, even new code that is included intentionally because it is assumed to add value to the final product, is a departure from the specification agreed to by the client. Therefore it needs to be acknowledged and presented to the client as such.

System design determines how the specification will be realized. This determines what the end

product will look like and how it will behave. While the point of design is to identify and plan how to implement system functions, seemingly simple decisions such as which component to start from can impact if those functions are fully included as agreed to in the final system.

Testing and Debugging

Testing is not just something that happens after all the programming is completed. Different types of tests are performed throughout the development cycle. User acceptance tests check the software against the original set of requirements to determine if it is complete per the specification. Integration tests verify that a particular module works properly when deployed with the rest of the system. Unit tests verify the correctness of functions and procedures at the code level. One proven problem with human cognition is the proclivity to search for evidence that confirms theories, rather than for evidence that refutes them. The term *confirmation bias* was coined by Peter Wason to describe this tendency.

The point of software testing is to find the defects in a system before they are propagated to the next phase of the software development life cycle. To find these defects testers should always be trying to fail the code. Unfortunately, it has been established that even experienced software engineering professionals are four times more likely to choose positive tests, designed to show that the code works, over negative tests, design to make the code fail. This is problematic since defects introduced into code have been directly correlated with the levels of confirmation bias in software developers, and defects introduced into production have been directly correlated with the levels of confirmation bias in software testers.

Studies looking at the roles that experience, education, and software development methods have on confirmation bias in software developers and testers indicate that experience in software development has little effect on the levels of confirmation bias displayed. Experienced developers are just as likely to design positive tests as neophytes. However, individuals with experience in software development that

are currently inactive in the field, for instance those who have moved on to research or management, seem to perform better on confirmation bias tests than individuals with experience in software development that are currently active in software development projects.

Cognitive scientists believe that confirmation bias is related to the way that humans construct models to examine information. It is believed that people do not evaluate propositions by applying algorithms to them and analyzing the results, but by constructing mental representations of the propositions and then examining the representations. For a positive test this is fairly simple: create a mental model where the expected inputs produce the expected outputs. Negative tests are more complex, typically starting with a positive model and then producing an inverse proposition. Additional models for expected and unexpected inputs and expected and unexpected outcomes also need to be produced. The more complicated the logic proposition, the more of these models that need to be created and compared to evaluate it. This is relevant to an activity closely related to testing: debugging.

Typically the job of a tester is to identify a failure in the software. When a failure is discovered, it is the job of a developer to examine it and find the cause in the code, a process that is also subject to confirmation bias. A defect does not always manifest itself in the system at the same point as it was introduced into the code. Starting from where the problem appears in the system, the developer assigned to the defect will need to form and test a theories of what happened in the code to cause it. The further from the problematic code the defect manifests, the more mental models the developer will need to create to track it down and verify it. With increasing distance between the flaw in the code and the expression of that flaw in the system, the less rigorous test cases introduced by confirmation bias increase the chance that the defect will not be properly diagnosed and adequately corrected.

While confirmation bias is a significant problem in software testing, it is not the only one. The availability bias also plays a large role in testing. Where confirmation bias addresses the quality of the test cases being produced, the availability bias affects the appropriateness of those tests. Recall that in

estimating the frequency of events, ease of recall will determine estimates of frequency. For a tester, the way she relates to the code will determine the parts she uses most frequently and consequently what comes to mind when determining what to test. Thorough testing should try to fail all the code, but the availability bias can lead to parts of the system not being thoroughly tested due to inadequate coverage of test cases.

Testing should ensure that the code that exists functions correctly. To do this, testers should always be trying to fail all of the code. However, confirmation biases and availability biases can lead to a system that is insufficiently and inadequately tested by having too many test cases written to demonstrate that the code works on too little of the code. For developers unit testing their code this means more errors making it into the code. For testers looking at this system this means more errors being deployed to end users. Both of these things negatively impact the quality of the final product.

Revising Project Status

In order to deliver a product on time, the process that creates that product must be managed. While estimating the amount of effort required to develop a system happens early in the software development life cycle, that estimate must be revisited and revised throughout the process in order to gauge progress, to set milestones, and to manage expectations. Project managers are responsible for assessing and reporting the status of a project. Since software development is a complicated task, it is difficult to control in the best of circumstances, having accurate information about the state of the project is essential to keeping everything running on time.

Unfortunately, project status reports demonstrate bias approximately 60% of the time. These biased reports skew two to one in favor of overly optimistic reports, where the status given is more favorable than indicated by the actual state of the project. Estimates indicate that only about 15% of projects with biased status reports are eventually completed on time, and that none of the optimistically

biased reports are among them. The biased projects that are completed on time are exclusively limited to those with pessimistic status reports where the project fell behind enough to hit that deadline. In light of this, it might appear that the amount of risk associated with the project would affect status reports, but biasing occurs equally across both high and low risk projects.

Some cognitive biases can help explain why this occurs. First, project managers tend to anchor on the initial project estimates. They are reluctant to change those estimates when reporting the status of a project at a later time. So, the less accurate those initial estimates prove to be, the less accurately project status will be reported in the future. Second, the planning fallacy can lead to project status reporting that is based on the best case scenario at the time of the report, not taking in the reality of the project and the difficulties it is facing.

Accurate project plans are essential to managing those projects and creating realistic expectations in the stakeholders. Technical challenges and changes are already likely to occur, making planning difficult. Managing these complexities is more difficult when optimistic status reporting creates unrealistic expectations and deadlines.

Recommendations

For as much as software systems can seem like they are all about coding and computers, systems of ones and zeros, it takes the coordinated efforts of a diverse group of people to bring them into existence. It has been said, software is of the people, by the people, and for the people, and delivering good software is about more than just writing good code. It is just as much about making good decisions. As table 3 illustrates, there are some ways that the human brain processes information that make this intrinsically challenging.

Table 3: Biases In Activities

Heuristic/Bias	Requirements Gathering	Estimating Effort	Re-Using Software Artifacts	Testing and Debugging	Revising Project Status

Representativeness	x				
Availability	x	x		x	
Anchoring	x		x		x
Adjustment Bias	x		x		x
Planning Fallacy		x			x
Overconfidence		x			
Confirmation Bias				x	

While these issues are unavoidable, they are predictable and, therefore, manageable. There is no evidence that simply being aware of the biases of representativeness, or availability, or the anchoring effect has any ability to mitigate their effects. To the contrary, there seems to be ample evidence that awareness does nothing to lessen the impacts of these biases. However, understanding these biases: how they work and what they do, it is possible to recognize the situations in which they occur and to develop strategies to combat them.

The first strategy is general: always prefer empirical investigation to intuition when the option is available. The foundations of representativeness and availability biases lie in estimating likelihoods and relying on memory. Whenever one has access to measurements, or tools to perform measurements using them will avoid the need to estimate, thus avoiding the problem. There are many tools available to manage the production of software and the management of software projects. While most can be helpful, there are some critical processes where using tools can provide substantial benefits. The first is traceability.

Traceability is a way to describe and follow the life of a software artifact throughout the SDLC. At the simplest level, traceability ties a requirement to a design artifact. High end traceability practices will include documentation of the decisions made in implementing the code, from requirement through design to implementation, testing, and deployment. High end traceability has been demonstrated to reduce the adverse effects of anchoring and adjustment biases when re-using software artifacts and when making changes to existing artifacts. Tools that allow for high end traceability, having extensive documentation tied directly the software artifacts in an easily accessible way, will result in more accurate

code.

Project velocity, much as the name suggests, is a measure of the amount of work being done over a period of time. It is a particularly useful concept in incremental development as it allows a team to measure the amount of work done over the span of an increment. Many tools monitor project velocity by producing burn down charts. These charts plot the amount of work remaining to be done (y-axis) versus time (x-axis). Day zero starts with the estimated amount of work for the entire increment. Each successive day the work remaining will decrease, as work is completed, or increase, as tasks are broken down and re-estimated. The results are charted and compared to an ideal work line. The biggest benefit of burn down charts are that they provide objective measures of development progress and estimated completion time which can offset optimistic project status reporting. Another benefit is that they provide feedback on the quality of initial estimates of effort. If the development team is always behind the ideal work line, but otherwise maintaining a consistent rate of work, it is an indicator that the initial estimates were too optimistic. Actual project velocity can be used to calculate an efficiency factor in order to correct for these optimistic estimates.

Unit testing can also benefit from the use of a tool. Test suites allow developers to write scripts to verify the code. These scripts can be saved and run whenever the code is changed in order to regression test it. While this is valuable in itself, most test suites will generate test stubs from existing code and provide reports indicating how much of the code is covered by test cases. Both of these options can combat availability biases and ensure adequate test coverage. Also, testing tools allow developers to practice test driven development. In test driven development, the unit tests for the code are written before the actual code itself. Under this method, developers write the test cases, run the tests so that they fail, and then write the code that satisfies the test cases until everything passes. Developing this way certainly ensures complete test coverage, but it can also help avoid confirmation bias, since the process starts with failing tests.

Not every problem can be solved with a tool. There is no way to automate the interview process when collecting new requirements for a project. However, some prompting techniques have been developed to avoid common cognitive biases. Three general techniques for avoiding cognitive biases including availability and representativeness are scenario building, generating counter arguments, and directed questions. Examples for each of these are provided in table 4. Including these general questions in interviews with users will produce better requirements.

Table 4: Requirements Prompting Techniques

Technique	Example
Scenario Building	Describe the most unusual customer you ever had. How did you respond in that situation?
Generating Counterargument	Why might the system not work as well as you think it will?
Directed Questions	For each of the last four times this happened, what was the customer's response? How many people did you ask? Do you think if you asked any other people about this problem that they would answer differently? Over the years, how often has this occurred? How many people does this usually affect?

When estimating effort several best practices can be employed. First, breaking projects down into their constituent parts, estimating effort for each part, and summing those efforts will produce a larger estimate for the project as a whole. This process of segmentation should be employed to combat the planning fallacy. Second, experts should be chosen based on domain expertise and a demonstrated record of accurate estimations. Neither one by itself is sufficient. Third, whenever possible have someone not responsible for the actual work estimate the amount of time it will take. This will avoid estimating based on an inside model and produce more objective estimates. One method for estimating the time it will take to complete a task that is commonly used by agile development teams is called planning poker. There is a meeting where each task is discussed. Team members write down their estimates for the task secretly, then all show them at the same time. This allows for collaborative time estimation while avoiding anchoring on the first estimates that are given.

Finally, while experience was not shown to affect the levels of confirmation bias displayed by

software developers and testers, training in logic was found to reduce them. To this end, box specifications and design strategies could be employed. Applying black box, state box, and clear box structures requires the creation of different mental models of the system that would help reduce confirmation bias in testing.

The best practices recommended in this section have been summarized in table 5. This list is not meant to be exhaustive as there are many other practices that could be implemented to help reduce the effects of cognitive biases on the software development process, and these are simply some of the results of the current research. They should be tried where appropriate, and used as a basis for the development of other practices that would benefit software development.

Table 5: Recommended Best Practices

Recommendation	Mitigating Effect
Prefer Empirical Investigation to Intuition	Actual statistics avoid the fallacies of estimating frequencies or likelihoods.
Traceability	Form more appropriate anchors to code and requirements. Adjust better from anchors to modify code more appropriately.
Project Velocity/Burn Down Charts	Objective measure of project progress. Predicted completion times based on measured rate of work done. Efficiency factor to offset overly optimistic estimates of time and effort.
Automated Unit Testing/Test Driven Development	Empirically ensure complete test coverage. Avoid confirmation bias by failing all tests first.
Scenario Building, Generating Counterarguments, Directed Questions	Avoid representativeness and availability biases in requirements gathering.
Task Segmentation	Estimating effort for the parts of a task leads to a larger estimate than estimating the task as a whole, mitigating the planning fallacy.
Choose Good Estimators	Experts should have both domain knowledge and a record of good estimates. Do not have them estimate their own projects. Mitigates the planning fallacy.
Planning Poker	Avoid anchoring on estimates when collaborating.
Box Structure Thinking	Decreased confirmation bias through training in and application of logic.

Conclusion

Developing software is difficult. There are ample statistics that make it clear that a lot of software development projects experience trouble. With IT being such a large industry, these troubles end up costing a lot money and producing problematic systems. While domains cannot be made easier to model,

and systems cannot be made easier to code, the process that produces that code and models those system can be examined and some fundamental areas of improvement can be identified. This paper sought to examine the people involved in production of software, the types of judgments they make, and some of the ways that cognitive psychology suggests that those judgments can be biased. This examination does not provide a silver bullet to fix all of the problems with the software development process. It does not provide a way to avoid making biased judgments. It does, however, provide a starting point for understanding how those judgments are formed, in what situations they are likely to occur, and what the results might be. These ideas are applied specifically to activities common to the domain of software development, and some best practices are recommended to mitigate some of the effects of these biases.

While simply knowing that these biases exist and understanding them has little to no effect in reducing their effects, a team, or an organization, or an industry that is more aware of the impact that these biases have on their projects can start discussing them. This discussion can lead to strategies to avoid them, and these strategies can lead to better business processes and practices. While this will not exactly make software easier to develop, it can keep it from being any more complex than it really needs to be. Hopefully this will lead to software systems being delivered closer to deadlines and budgets, with more of their expected functionality, and fewer bugs.

References

1. Kahneman, D. (2011). *Thinking, fast and slow*. Farrar, Straus and Giroux.
2. Tversky, A., & Kahneman, D. (1974). Judgment under uncertainty: Heuristics and biases. *Science; Science*.
3. Kahneman, D., & Tversky, A. (1984). Choices, values, and frames. *American psychologist*, *39*(4), 341.
4. Stacy, W., & MacMillan, J. (1995). Cognitive bias in software engineering. *Communications of the ACM*, *38*(6), 57-63.
5. Calikli, G., & Bener, A. (2010, September). Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* (p. 10). ACM.
6. Teasley, B., Leventhal, L. M., & Rohlman, D. S. (1993). Positive test bias in software testing by professionals: what's right and what's wrong. In *Empirical Studies of Programmers: Fifth Workshop* (p. 206).
7. Forsyth, D. K., & Burt, C. D. (2008). Allocating time to future tasks: The effect of task segmentation on planning fallacy bias. *Memory & cognition*, *36*(4), 791-798.
8. Jørgensen, M. (2004). A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, *70*(1), 37-60.
9. Browne, G. J., & Ramesh, V. (2002). Improving information requirements determination: a cognitive perspective. *Information & Management*, *39*(8), 625-645.
10. Browne, G. J., & Rogich, M. B. (2001). An empirical investigation of user requirements elicitation: Comparing the effectiveness of prompting techniques.*Journal of Management Information Systems*, *17*(4), 223-250.

11. Parsons, J., & Saunders, C. (2004). Cognitive heuristics in software engineering applying and extending anchoring and adjustment to artifact reuse. *Software Engineering, IEEE Transactions on*, *30*(12), 873-888.
12. Kirs, P. J., Pflughoeft, K., & Kroeck, G. (2001). A process model cognitive biasing effects in information systems development and usage. *Information & Management*, *38*(3), 153-165.
13. Mohan, K., & Jain, R. (2008). Using traceability to mitigate cognitive biases in software development. *Communications of the ACM*, *51*(9), 110-114.
14. Ralph, P. (2011). Toward a theory of debiasing software development. *Research in Systems Analysis and Design: Models and Methods*, 92-105.
15. Snow, A. P., Keil, M., & Wallace, L. (2007). The effects of optimistic and pessimistic biasing on software project status reporting. *Information & management*, *44*(2), 130-141.