Ensuring Your Software is Reliable

Alex Sheriff

***Abstract*—This document provides a detailed overview of what it means to have reliable software, and the best techniques to achieve a high mean time between failure. A software fault, or bug, causes software to fail. Faults can be classified as a Bohrbug or Mandelbug. Mandelbugs have sub-types called Heisenbugs and aging-related bugs. Using techniques like N-version programming, recovery blocks, and exception handling, engineers can enforce their system to be more fault-tolerant. Growth models can be incorporated to determine when a software system is ready to be shipped. Multi-stage models are used when systems are too complex for one growth model. Model-based reliability analysis utilizes models to attempt to quantify the reliability of a system before shipment.**

Index Terms--- Reliability, fault, fault-tolerant, failure, bohrbug, mandelbug, heisenbug, n-version programming, recovery blocks, exception handling, growth model, multi-stage models.

ACRONYMS

MTBF	Mean time between failure
MTTR	Mean time to repair
NVP	N-version programming
SDLC	Software development life cycle
QA	Quality assurance
OS	Operating system
IDE	Integrated development environment
PEH	Programmed exception handling
DEH	Default exception handling
RSS	Residual sum of squares
AIC	Akaike information criterion
MBRA	Model-based reliability analysis
DMA	Design margin analysis
SLA	Specification limit analysis

I. INTRODUCTION

TYPICALLY, software reliability is determined after shipment of the system. User reports, system outages, bug reports, customer feedback, etc. aid in software maintenance after shipment [1]. Faults, or bugs, are not only a product of bad programming, but can also be attributed to poor requirements, poor design documentation, unexpected environmental variables, outdated software, and more. In some cases, software engineers need to decide whether investing too much time in eliminating and tolerating bugs is worth the money. It can add unexpected time to the SDLC but can also affect process time because of added fault-tolerant code. For some real-time systems, hard deadlines are crucial for the system to function as desired. Adding lines of code to prevent bugs can cause a high system complexity which may indirectly cause more bugs.

The role of a software reliability engineer is to balance factors like scheduling and system performance while maintaining a high mean time before failure. A high MTBF means a high reliability. According to Armstrong, the goal of software reliability is to achieve "10 nines," e.g., 99.99999999% uptime [11]. Jim Gray also notes that a 99.6% reliability may be too low for some businesses. Hospitals depend on appliances and systems to treat patients, and a 99.6% availability would mean that every 10 days, a system breaks for about 1.5 hours [3].

To achieve "10 nines" of reliability, we must follow six laws: isolation, concurrency, failure detection, fault identification, live code upgrade, and stable storage. Isolation refers to the idea of having as little environmental variables affecting the system as possible. Environmental variables are unpredictable and can negatively affect the system if the system isn't prepared to handle unexpected variables. Concurrency refers having more than one system responsible for a process. Concurrency is synonymous with process-pairs, which will be discussed later, but generally means that multiple systems are responsible for the same process. Failure detection refers to logging failures and gathering data at the time of failure to determine the fault. Fault identification is recognizing what caused the fault, and then implementing the proper measures for it to not occur again. Live code upgrades are important because some systems should not be turned off, therefore any updates to a system need to be able to take place while the system is operating. Lastly, stable storage, means that the data being stored is safe, scalable and maintains a high integrity.

According to Wood, growth models are a good way to determine if a system is ready to be shipped [1]. Using appropriate models, the number of faults remaining in your software can be estimated to ensure that enough testing time has been completed. The general curve of faults, or defects, can be visualized using figure 1.

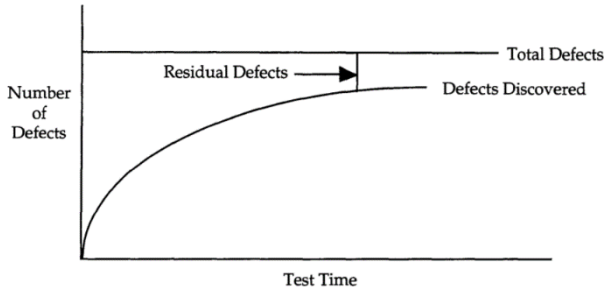

Fig. 1. Residual Defects

The residual defects represent the faults present after the testing phase of the software system. This point represents the nature of accepting potential failures for the sake of

staying within schedule. Software can continue to be updated and made to be more reliable after the shipment.

Using models can be incredibly beneficial to increasing the reliability of software. Models are non-invasive and cost-effective and are typically used before testing phases. These models can aid in testing efforts by estimating how many faults are likely to be undiscovered at any point in time.

II. FAULT CLASSIFICATION

A fault, or bug, is a defect in software code that can make it so the software system does not operate properly. Some faults can cause a system failure, in which the entire system becomes inoperable. Resolving faults that cause system failures before shipping the product are of the highest priority for developers. These faults have different classifications as described in [2].

A. Bohrbugs

As described by Grottke, Bohrbugs are a type of fault that is easily isolated and that manifests consistently under a well-defined set of conditions because its activation and error propagation lack "complexity" as set out in the definition of Mandelbug [2]. During the testing phase of a software development life cycle, QA Engineers will likely find an abundance of Bohrbugs. Furthermore, after the product has shipped and been in production for many months or years, it is likely that all bohrbugs will be eliminated.

Bohrbugs are considered permanent faults [5]. Permanent faults will exist until the fault is fixed.

B. Mandelbugs

As described by Grottke, A fault whose activation and/or error propagation are complex, where "complexity" can take two forms:

1) The activation and/or error propagation depend on interactions between conditions occurring inside the application and conditions that accrue within the system internal environment of the application.
2) There is a time lag between fault activation and failure occurrence, e.g., because several different error states have to be traversed in the error propagation. [2]

Mandelbugs, in more simplistic terms, are faults that are not easily detectable and reproduceable. This may be because of a change in the system environment, or other unforeseen causes. Mandelbugs have two sub-types: heisenbugs and aging-related bugs. Figure 1 illustrates the relationship between bohrbugs and mandelbugs.

Mandelbugs are considered transient faults [5]. Transient faults may occur in seemingly identical environments. Sometimes they occur, sometimes they do not.

C. Heisenbugs

Jim Gray writes of heisenbugs in his 1985 technical report on reliability [3]. Based on the Heisenberg Uncertainty Principle, heisenbugs manifest when we aren't looking for them. Heisenbugs are very difficult to resolve because they don't appear consistently. A heisenbug will manifest seemingly at random, and any attempt to isolate the bug is met with a working fault-free code module. It is certainly not impossible to catch a heisenbug, however it is significantly more difficult than catching a bohrbug.

One example, albeit outdated, of a heisenbug from an unknown commenter tells a college story of trying to analyze why printing from a buffer would not work [4]. On attempt to troubleshoot the issue, the developers added a dump to determine the status of the system before printing, and suddenly the program operated as intended. What they eventually learned was that the developers were using a read command when they should've used a "read and wait until done" command. The dump they were using to troubleshoot the problem was creating enough time for the read command to finish processing, and a proper print would occur.

All heisenbugs are mandelbugs. Not all mandelbugs are heisenbugs. This is visualized in figure 2.

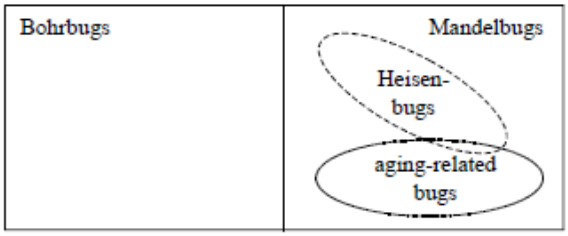

Fig. 2. Venn diagram of software fault types

D. Aging-related bugs

The last fault classification necessary to discuss is aging-related bugs. By the name, these are bugs that manifest because of some quantity of time. Age may refer to the length of time a system has been running, or the physical length of time the system has been in production.

For example, a system requires input from different variables, and then casts each variable to one singular type. The system then uses these variables to make computations when the customer recognizes that sometimes the result is off by a few decimals- just enough to cause suspicion. The act of casting each variable is causing rounding errors in the program [2]. This scenario is considered an aging-related bug because the system does not detect any faults during the processing of the program. It is only discovered after the result is displayed, and the root cause of the problem is hidden within the body of the program code.

Another example of aging-related bugs is simple deprecation of program methods and tools. For example, some IDEs do not recognize when a built-in function is deprecated. When compiling the code, the IDE throws an error, and the developer begins researching that specific error. What the developer may not realize is not a problem with the syntax or logic itself, rather a problem with the literal function used. This is considered an aging-related bug because the program has been in production for a very long time, and the code supporting that application is no longer applicable to the purpose of the program.

A more formal definition of aging-related bugs is as follows: A fault that leads to the accumulation
of errors either inside the running application or in
its system-internal environment, resulting in an increased failure rate and/or degraded performance. [2]

III. TECHNIQUES TO BECOME FAULT-TOLERANT

It is unreasonable to expect a 100% fault-free system before deployment. Therefore, it is much more time-efficient to make your software fault-tolerant. Fault tolerance refers to the idea that even in the presence of bugs, your software will continue to run and produce accepted results. Jim Gray lists his keys to fault-tolerant software in [3].

- Software modularity through processes and messages
- Fault containment through fail-fast software modules.
- Process-pairs to tolerate hardware and transient software faults.
- Transaction mechanism to provide data and message integrity.
- Transaction mechanism combined with process-pairs to ease exception handling and tolerate software faults.

Fault-tolerant approaches can be classified as fault removal and fault-masking. Within fault-removal approaches, there is forward error recovery and backward error recovery. Forward error recovery attempts to fix the fault and restore the system state without system failure. Backward error recovery restores the system to a state before the fault occurred.

A. Process-pairs

Process-pairs is a technique described in [6] where a system has a primary process and a back-up process. When the primary process fails, the back-up process continues execution. Gray lists five approaches as to how to design process-pairs [3].

The first approach, lockstep, the primary and backup processes execute the same code at the same time. If one process fails, the other continues. Lockstep is the simplest in design because any bug present in the primary process will be present in the back up process. Gray notes that this approach is only good to prevent failures related to hardware and is not tolerant to heisenbugs.

The next design approach is called state checkpointing. State checkpointing involves assigning a requestor to a process-pair. The primary process executes code as per usual; however, it sends updates on its current state to the back-up process. When a primary process fails, the back-up process begins. The design, while effective, is difficult to implement and develop.

Delta checkpointing is like state checkpointing, but it sends logical updates rather than physical. This decreases the number of overall messages being sent to the back-up process, increasing performance time as well as reducing the chances of a bug in the primary process also causing the back-up to fail.

In state checkpointing, the programmer is tasked with explicitly distinguishing what constitutes a checkpoint. Obviously, this can be very time consuming. With automatic checkpointing, the operating system kernel can automatically determine these checkpoints. As with state checkpointing, the primary process runs until it fails, at which point it sends all its logs to the back-up process to continue. This design suffers from a relatively longer execution time, but otherwise is effective.

According to Gray, the simplest design, and easiest to program, is persistent process-pairs. Unfortunately, it only checkpoints start states and fail states, so that when the primary process does inevitably fail, the back-up process begins without knowledge of what the primary process was executing. This objectively is the worst process-pair design as it is. Gray however suggests incorporating persistent pairs with transactions in order to resynchronize the pairs.

Incorporating transactions with persistent process pairs allows the back-up pair to restart interrupted transactions from the failed primary process. When the back-up process "wakes up," all incomplete transactions will be undone, and then the back-up process attempts to execute them. This allows persistent process pairs to be fault-tolerant, which is easy to develop, and tolerant to hardware and software faults.

It should be noted that the above process-pair design principles are outdated. They were included in this paper to aid in giving a fundamental understanding of how different design principles can be applied to a system to achieve a high fault-tolerance. Going forward we will describe more modern solutions to creating fault-tolerant software.

B. N-Version Programming

N-version programming is the concept of having N number of teams or individuals, independently developing the same functional system under the same requirements. The teams do not consult each other regarding the programming itself. Moreover, it is recommended that the way each team goes about programming their system is as different as possible. This includes programming languages, algorithms, techniques, tools, and even IDEs when possible.

After each programmer or team has developed their system, testing each system to determine the highest reliability occurs. This is done by using a mechanism called a voter. Each system is executed simultaneously with results sent to the voter. The voter then decides on the best result. This naturally entails that some systems will fail when others don't, however with a voter in place it is likely and expected that we will usually obtain an acceptable result. The probability of failure is described in figure 3. The first term being the probability that all versions fail, the second that only one version is correct, and, d, the probability that there are two correct results however the voter delivers the incorrect result.

$$p_{nv} = \prod_{i=1}^{n} e_i + \prod_{i=1}^{n}(1-e_i)e_i^{-1}\prod_{j=1}^{n} e_j + d$$

Fig. 3. Probability of NVP failing

C. Recovery Blocks

Recovery blocks are divided into three main parts. The primary module, acceptance tests, and alternate module. The general idea is that if the primary module fails, an alternate module will attempt to execute the code. Alternate modules will continue attempting to executing code until either one succeeds or no more modules are left, and the system fails. Recovery blocks are a similar concept to NVP, the main difference between them is that the latter is executed simultaneously. Figure 4 represents the probability of failure of a recovery block scheme.

$$P_{rb} = \prod_{i=1}^{n}(e_i + t_{2i}) + \sum_{i=1}^{n} t_{1i}e_i \left(\prod_{j=1}^{i-1}(e_j + t_{2j})\right)$$

Fig. 4. Probability of recovery block scheme failure

N represents the number of alternative modules, while *t* represents the acceptance test. Next, e_i is the probability of failure for version P_i , t_{1i} is the probability that *t* judges an incorrect result as correct, and t_{2i} is the probability that *t* judges a correct result as incorrect. The second term of the equation determines the probability that the i^{th} acceptance test judges an incorrect result as correct at the i^{th} trial of the n versions [7].

D. Exception Handling

When a system experiences something unexpected and is unable to resolve it properly, it typically will fail. These occurrences are classified as "exceptional conditions". Exceptional conditions are a term synonymous with faults or bugs. According to Shelton, failures caused from these exceptional conditions make up two-thirds of system crashes [8]. Exception handling is the attempt of recovering from such faults and continue system run time. Shelton argues that exception handling is less-so fault-tolerance, more-so fault avoidance. This is because exception handling usually involves tackling unpredictable and unforeseen faults, however exception handling known faults is common. Building exception handlers to tackle mandelbugs can be incredibly difficult as the programmer is unaware of what or why certain system faults may occur. The following will discuss exception handling for expected and unexpected faults.

Programmed exception handling is a technique used for common faults that are likely to occur. PEH is also a derivative of recovery blocks, as when an exceptional condition occurs, a secondary code module takes control to avoid system failure. PEH is capable of forward error recovery, allowing the system to recover from faults during processing, and backwards recovery. If the system fails with forward error recovery, it tries backward error recovery. In other words, if the system can't continue processing than it will return to a previous state and try to run the code again. The most important aspect of PEH to understand is that it is used when programmers are aware of common and likely to occur bugs, and specifically add such exception handlers to recover from these specific bugs.

Default exception handling is designed to handle any faults that weren't predicted by programmers. This entails creating a "catch-all" exception handler for all unexpected faults that occur. Logically, a universal exception handler won't be very capable of forward error recovery, so backward error recovery is its best chance. DEH attempts to recover from errors with use of checkpointing and reverting to a previous state to try code execution again. Often, the expectation of a good default exception handler is that it will make failing "graceful" [8].

IV. RELIABILITY MODELS

It is impossible to quantify the reliability of a system. The closest thing that could be measured to quantify reliability is Mean time before failure. MTBF is an average of how long a system will remain operable before failure. The higher the MTBF the higher the reliability of the system. Reliability models are an attempt at quantifying the reliability of a system. Over 200 models have been made since the 1970s [9] and no model can be used universally. Models are based on observing and collecting failure data and cross referencing with statistical expectations. Table I outlines the two sub-categories of software reliability modeling techniques.

TABLE I
Difference between prediction and estimation models [9]

Basics	Prediction Models	Estimation Models
Data Reference	Uses historical information	Uses data from the current software development effort.
When used in development cycle	Usually made before development or test phases; can be used as early as concept phase.	Usually made later in the life cycle (after some data have been collected); not typically used in concept or development phases.
Time Frame	Predict reliability at some future time.	Estimate reliability at either present or some next time.

A. Growth Model

A reliability growth model is a numerical interpretation of software reliability designed to predict how reliability should improve over time [9]. As mentioned earlier, it can be

difficult for a project manager to balance time invested in each stage of a project. Using a growth model can assist the project manager with determining how much time should be spent testing the system. Growth models are a form of estimation models.

One type of growth model is the Wall and Ferguson Model. The model simply tries to predict the failure rate of a system during testing phases [12]. Figure 5 represents the total number of failures at time t. The parameter a_0 is unidentified, and b(t) can be calculated from the number of test cases or total test time.

$$\mathbf{m_{(t)} = a_0[b(t)]^3}$$

Fig. 5. The total number of failures at time t

Using figure 6, one can determine the failure rate of a system.

$$\lambda_{(t)} = \mathbf{m}'^{(t)} = \mathbf{a_0}\beta\mathbf{b}'^{(t)}[\mathrm{b(t)}]^{\beta-1}$$

Fig. 6. Failure rate at time t

The variable β is a model parameter, which can be estimated using the least squares approach. While not much information is available specifically on the Wall and Ferguson Model, according to [12] the two individuals found a high correlation between failure data and their predictability of faults remaining. This confirmed to them that while the model isn't completely accurate, it can still be used to estimate roughly how likely their system would fail.

As mentioned earlier, there are over 200 different models. It may be worth a team's time to design an independent model catered to their specific design principles. This model could then be used for future products based on the reliability of past products.

B. Multi-stage Model

In some cases, a system is expected input of many different types of data. One model is not sufficient to predict the reliability of the system, so multiple must be used. Multi-stage models are the solution to such problems. Dividing the data into stages, the multi-stage model uses many different models and combines them into one [13]. Each unique model is responsible for one data stage.

A multi-stage model can have many growth models implemented. A significant amount of time should be spent determining if models are compatible with each other. If they are not, the multi-stage model may not be functional. One common problem is the number of parameters for independent models. If one model has only two parameters necessary, while another model has four parameters, one model may skew results from other models.

This can be determined using an evaluation tool called the residual sum of squares [13]. The RSS can be used to determine the effectiveness of a model in recognizing how many faults have been found in a system. This is important because we want RSS ratios to be of similar value when included in a multi-stage model. When models have a different number of parameters from each other, their RSS values will naturally differ more-so than if they had the same number of parameters.

[13] describes an effective way to determining how to create a multi-stage model, shown in figure 8. The flowchart is based on a two model multi-stage model. The two models used are a logistic model and the Gompertz model. AIC refers to the Akaike information criterion which is used to evaluate stage models. AIC estimates the quality of each model relative to each of the other models.

Multi-stage models are best used when one singular growth model is not compatible with all the different data types being input to a system.

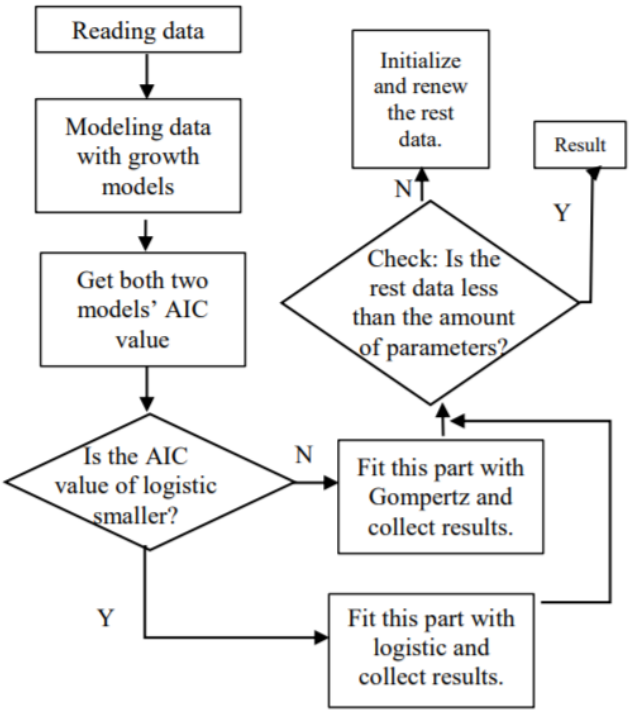

Fig. 8. Flowchart of the process to create a multi-stage model

V. MODEL-BASED RELIABILITY ANALYSIS

The object of model-based reliability analysis is to identify ways to capitalize on the insights gained from physical-response modeling both to supplement the information obtained from testing and to better-understand test results [14]. Bierbaum, et al, describes that there are five general processes which can obtain quantitative and qualitative remarks on reliability:

- Design margin analysis
- Specification limit analysis
- Lifetime prediction
- Anomaly Investigation
- Probability Quantification.

A. *Design Margin Analysis*

DMA is the foundation to the other MBRA processes. It isn't limited to only one phase of the SDLC. DMA serves to identify components of the system that are most influential to system performance. It also analyzes the margin of performance required for systems. In other words, if a system requires response time within a few milliseconds, what can we do to improve or change that margin for the better.

A generalized process for this can be found in [14] and main steps are as follows:

1) Identify scope of analysis
2) Identify key response variables
3) Plan variability study
4) Gather needed device-model data
5) Develop model and validate
6) Run variability study
7) Analyze results

The results generated from a DMA can be used to target imperfections in the system to better improve performance, as well as determining the environmental specifications needed to run such a system. Improving the performance of a system will not inherently make your system more reliable; it will, however, shed light on areas of improvement to increase the reliability of the system. More information on the specific steps of DMA can be found at [14].

B. *Specification Limit Analysis*

The intention of a specification limit analysis is to reduce the overall time and cost that normally would not be optional with DMA. When the specifications of an environment are not high enough, it could cause the system to fail. A common modern example is computer video games. The computer specifications required to run a game are mandatory, or else significant performance problems may arise. When running a game underneath the minimum specifications, it is more than likely that the game will fail or crash often, and for all intents and purposes be unplayable.

One problem with DMA that does not affect SLA is that during tests to determine specifications, the subsystem might have failed the specification limit but would not have prevented the system from working [14]. In other words, under testing specifications, the tests may fail, but they result in passing. SLA can determine the specifications of a system, and in turn allow programmers to optimize the system further if the specifications aren't as they'd like.

C. *Lifetime Prediction*

Lifetime prediction is the prediction of how long a system will be operable. It can also predict the impact of age-related changes. Bierbaum, et al, describes the importance of such lifetime prediction, as the MBRA approach allows incorporating models of material or piece-part degradation into a subassembly- or system-level context. This also provides a framework to evaluate pervasive aging throughout the system, e.g., if multiple piece parts are changing with time [14]. The steps for lifetime prediction are as follows:

1) Analyze the subsystem or system for key contributors using the MBRA DMA process.
2) Develop a model of behavior as a function of age.
3) Incorporate the models developed in step 2 for all critical age-related variables identified in step 1, into a subsystem or system model.
4) Exercise the model as a function of time and examine the outputs to see if specifications are not met at some time point.

After step four, if you find an output that doesn't meet the specifications at a time point, that time point is the lifetime of the system. For real-time systems, this lifetime prediction gives builders a chance to change materials if the lifetime of the product is not favorable.

D. *Anomaly Investigation*

Anomalies in this context are synonymous with faults. Incorporating modeling in anomaly investigation is a cost-effective way to locate and fix faults as well as determine when other tests still need to be run before shipment of a product. There are three steps to providing a MBRA Anomaly investigation [14]:

1) Develop a model with adequate detail (as for the DMA) to emulate the anomalous results.
2) Postulate root-causes and inject them into the model; compare with the observed anomaly and add to the list of possible sets of anomalous behavior if they are similar.
3) After generating a set of anomalous behaviors, it is necessary to determine their impact. Each type of anomalous behavior should be explored vis-à-vis the sources of variability, the different environments that can be anticipated, and the interface conditions that might be experienced.

The general purpose of anomaly investigation is to understand how certain types of faults can affect a system in a generalized capacity. One key benefit is the ability to inject faults into the model without affecting the actual system performance. This is very useful for this tool. It gives teams a better understanding of how faults interact and affect systems, and therefore can allow testers and designers to know when to expect faults at code modules. Like with the other processes so far, more information on anomaly investigation can be found at [14].

E. *Probability Quantification*

Probability quantification is the culmination of previous MBRA methods, especially DMA. Probability quantification is used to make reliability statements. It also requires very precise and confident data, more-so than previous methods. The process for probability quantification is similar to DMA, with one added step to compare the output distributions of the key response variables to specified success/failure criteria to determine reliability [14]. Though it is imperative that confidence in the model, as well as key factors and other aspects, is as high as can be, if executed properly the reliability statement will be highly accurate.

VI. RECOMMENDATIONS

The most difficult aspect of achieving high software reliability is ensuring that the cost of a project will remain within its intended budget. Likewise, staying within the schedule of shipment dates is also important. As mentioned previously, software will never be 100% fault-free, so spending endless amounts of time trying to eliminate every contingent fault is not cost-effective.

During development, engineers may have to decide between reliability and functionality. Regarding exception handling, too many exception handles can add to runtime. For real-time systems, it can be a fine line between having a fault-tolerant system and a system that functions between time constraints appropriately. This means that teams will be responsible for deciding between getting correct results or getting results on time.

It is also recommended that after deployment, teams continue to monitor and aggregate failure data. Working with system designers to better design systems to alleviate potential faults in the future will ensure future generations of products are reliable.

Overall, the best way to ensure reliability for products is to use the data collected from past products. Designing your new products based on products that have achieved high reliability seems to be an effective means. In doing so, engineers can utilize past models to estimate current product fault count. Similarly, data on faults from past projects can assist in programmers ability to utilize programmed exception handling, as PEH is based on the anticipation of faults.

Likewise, utilizing NVP or recovery blocks is an effective way to ensure reliability for products that do not have previous systems to be designed from. Recovery blocks are not effective when computation deadlines are more important than processing time, and therefore should be avoided when working with real-time systems.

Lastly, the IEEE Recommended Practice on Software Reliability 2016 is available to read online at [15]. It is exhaustive and covers many of the topics discussed in this paper.

VI. CONCLUSION

Reliable software is essential for happy clients. Achieving reliable software is difficult, but not impossible. Teams that spend time and energy into data collection on the reliability of shipped products will be able to incorporate better design principles for future products. Incorporating this data into models will similarly add to the engineer's ability to create reliable software. Nonetheless, over their lifetime, software systems will increase in reliability with adequate maintenance performed.

VII. REFERENCES

[1] A. Wood, "Predicting Software Reliability", Tandem Computers, Cupertino, CA, Tech. Report. TR-96.1 PN130056, November, 1996. [Accessed 12 December 2021].

[2] M. Grottke and K.S. Trivedi, "A classification of software faults." *Supplemental Proc. Sixteenth International IEEE Symposium on Software Reliability Engineering*, pages 4.19–4.20, 2005. [Online]. [Accessed 10 December 2021].

[3] J. Gray, "Why do computers stop and what can be done about it?" ,*Technical Report* 85.7, PN87614, Tandem Computers, Cupertino, CA, Tech. Report. TR- 85.7 PN87614, June 1985. [Accessed 10 December 2021].

[4] "Heisen bug examples", 21 January, 2004, [Online]. Available: http://c2.com/cgi/wiki?HeisenBugExamples. [Accessed 13 December, 2021].

[5] G. Kumar Saha, "Transient software fault tolerance using single-version algorithm", *Ubiquity*, August 2005. [Online], available: *https://doi.org/10.1145/1088431.1086449*. [Accessed 12 December 2021].

[6] E. Roberts and J. Hennessy, "How is fault tolerant computing done?", 2003-4. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/fault-tolerant-computing/how-tandem.html. [Accessed 8 December 2021].

[7] "Basic fault tolerant software techniques," *geeksforgeeks.org*, 8 October 2018. [Online]. Available: https://www.geeksforgeeks.org/basic-fault-tolerant-software-techniques/?ref=lbp, [Accessed 16 December 2021].

[8] C.P. Shelton, "Exception Handling," 1999. [Online]. Available: https://users.ece.cmu.edu/~koopman/des_s99/exceptions/. [Accessed 14 December 2021].

[9] JavaTpoint, "Software reliability models." [Online]. Available: https://www.javatpoint.com/software-engineering-software-reliability-models. [Accessed 15 December 2021].

[10] IEEE Reliability Society, "IEEE recommended practice on software reliability," IEEE Std 1633™-2008, 27 June 2008. [Online]. Available https://sci-hub.se/10.1109/ieeestd.2008.4554206. [Accessed 11 December 2021].

[11] J. Armstrong, "Systems that never stop (and Erlang)," *Qconlondon*, 2009. [Online]. Available: https://qconlondon.com/london-2009/qconlondon.com/dl/qcon-london-2009/slides/JoeArmstrong_ErlangALanguageForProgrammingReliableSystems.pdf. [Accessed 18 December 2021].

[12] V. Nanda, “Reliability growth models,” tutorialspoint.com, 30 October 2021. [Online]. Available: https://www.tutorialspoint.com/reliability-growth-models. [Accessed 10 December 2021].

[13] J. Chi *et al.*, "Defect analysis and prediction by applying the multistage software reliability growth model," *2017 8th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2017, pp. 7-11. [Online]. Available: http://www.washi.cs.waseda.ac.jp/wp-content/uploads/2017/01/IWESEP2017_paper_12.pdf. [Accessed 13 December 2021].

[14] R.L. Bierbaum, T. D. Brown, and T. J. Kerschen, “Model-based reliability analysis,” *IEEE Transactions on Reliability*, Vol. 51, no. 2, June 2002. [Online]. Available: https://sci-hub.se/10.1109/tr.2002.1011517. [Accessed 18 December 2021].

[15] IEEE Reliability Society, “IEEE recommended practice on software reliability,” IEEE Std 1633™-2017, 18 January 2017. [Online]. Available https://standards.ieee.org/standard/1633-2016.html. [Accessed 11 December 2021].