Error handling and recovery strategies in software development

Alexy de Almeida
DEI
Universidade de Coimbra
Portugal, Coimbra
adenis@student.dei.uc.pt

Edgar Duarte
DEI
Universidade de Coimbra
Portugal, Coimbra
edgarduarte@student.dei.uc.pt

Rodrigo Ferreira
DEI
Universidade de Coimbra
Portugal, Coimbra
rferreira@student.dei.uc.pt

***Abstract*—This paper presents the findings of a survey on error handling and recovery strategies in software development, which are important components of software quality that are used in order to avoid, identify and recover from problems that could happen during a product's or service's lifespan. It discusses several techniques for error handling and recovery, including exception handling, assertion, and defensive programming, highlighting their significance and importance, whilst also showing their elasticity and compatibility with other methods. Our research also demonstrates how proficient error management and recovery may drastically lower system downtime, increase dependability, and improve user experience. We also describe how the future of error handling and recovery is expected to be shaped by emerging themes, such as tools for automated error detection and recovery, chaotic engineering and many more.**

***Index Terms*—Error Handling, Recovery Strategies, Software Development, Metrics, Case Studies, Emerging Trends, Quality, Testing, Fault Tolerance, Machine Learning, Automation, Detection, Reporting, Evaluation**

I. Introduction

Nowadays, when developing any type of software, the occurrence of errors is nearly inevitable. Due to this, it is absolutely crucial to have systems that can effectively and accurately handle and recover from errors. This survey aims to offer in-depth explanations of error handling and recovery strategies, highlighting their significance in improving the reliability and overall quality of the software. Understanding these concepts accurately should prove to be an effective weapon in the arsenal of anyone involved in developing software that needs quality assurance. Furthermore, this document should be of value to developers or researchers seeking to deepen their knowledge in this field.

The survey will focus on diverse topics, starting with a general overview of definitions, techniques, practices and strategies on error handling and recovery. Afterwards, the document dives into error detection and reporting, where a deeper look is taken into the methods used and the best practices currently employed. The evaluation of error handling and recovery will also be explored, where different criterion, metrics and case studies will be scrutinized. Finally, a glance will be put upon different emerging trends in the field, like machine learning algorithms and autonomous systems.

A. Overview of software quality and dependability concepts and terminology

As software becomes more and more complex, the need to ensure its quality increases. As consequence, when thinking about **Software Quality** we should keep in mind some key terms. One of these terms is **Reliability**, which, in short, means the system works as expected. A reliable system is one that has a high probability of performing its intended function, with low downtime and few failures. Several factors may impact how reliable a system is, with software testing being one of the most important because how helpful it can be for fixing bugs prior to the product's release.

Another key term that is closely related to reliability is the **Robustness** of the software, as it refers to the ability to handle and recover from a wide range of error types and failures scenarios, such as unexpected errors, edge cases, and exceptions. Since the software system will communicate with other systems and users it's important to be able to handle unexpected behaviour.

On the topic of handling unexpected behavior, a **Resilient** system should have the ability to detect any unexpected behavior such as the presence of errors or failures, and be able to recover from them, thus returning to normal operation within an acceptable period of time.

The terms above allow to have more robust and resilient software system, but at the same time they may impact negatively the softawre **Performance**, which is the measurement of how efficiently that software system can accomplish its functionalities.

Additionally, **Scalability** is also an important concept that refers to how well a system can handle growing amounts of data and tasks. It's an important concept and depending on the work scale a software system is expected to handle, some architectural changes may be necessary.

Furthermore, the **Maintainability** of the software refers to the ease with which the software code can be improved, understood and corrected. Seeing that software is in constant change, it is crucial that it is built in a modular and flexible way, in order to allow the implementation of new features without the need of major overhauls, reducing costs and downtime.

In connection to downtime, **Availability** refers to the amount of time that the system is operational and functioning as intended. It differs from reliability in the sense that availability measures the percentage of time that the system is operational, while reliability measures the probability that the system will meet its intended purpose.

For a software quality, the quality of the user experience is also very important, meaning that Software **Usability** should be kept in mind when developing software. If a user cannot easily understand and operate a software it might lead to potential errors that could be prevented with a clearer interface.

Besides software quality, it's also important to consider Software **Dependability**, a metric that illustrates how much we can rely on the software to function as intended. As such, it is an essential attribute to have in critical systems like airplanes and medical equipment, as their failure could have disastrous effects. Some key concepts when talking about software dependability are:

- **Fault tolerance**: the capacity of the software to continue to function when encountering faults;
- **Fault prevention**: the attempt to predict and prevent faults or defects from occurring;
- **Fault removal**: the process of identifying and removing faults from the software;
- **Fault forecasting**: the process of detecting deviations from expected values, in order to predict the likelihood of the occurrence of a fault. This is achieved by using statistical models based on complex mathematics that analyse data based on past defects in order to detect common patterns.

As discussed, software quality and software dependability, while very different, are closely related in the sense that a dependable software is more likely to have high quality. Throughout the survey we shall discuss how software should aim to ensure both attributes to create a reliable, resilient and robust system.

Lastly, it is important to outline the different types of errors and mistakes that could affect the quality of a software. Starting by the most important concept, an **Error** is when a system or program behaves unexpectedly or produces an unpleasant result. Several things, including wrong inputs, improper code syntax, broken hardware, and network problems, might result in errors. **Code related errors** are errors that occur mainly in the development phase and are associated with the quality of the code being developed. They are:

- **Syntax errors**: derive from creating words or phrases that the code compiler or interpreter cannot understand, leading to an early execution abortion;
- **Runtime errors**: occur when a running program crashes unexpectedly. The root of the error varies, but the most common are incorrect input data and memory leaks;
- **Semantic errors**: when a code that is syntactically correct does not make sense.
- **Logical errors**: happen when the programmer's logic is flawed, leading to incorrect and and undesired behaviours.

A good software developer can positively impact the quantity and severity of the above mentioned errors. There also exist errors that are not directly related to code, but related to the overall design of the software. These errors are most common in the stages prior to the development phase of the software, some examples being:

- **Design errors**: occur when the software was poorly designed. A bad design might make it impossible or very hard to implement new functionalities and might lead to an inefficient and unreliable system;
- **Requirements errors**: derive from a poor requirement elicitation process that makes the software system incomplete and inconsistent, which may lead to unsatisfied stakeholders;
- **Testing errors**: occur when the test cases are incomplete, ineffective or defective, allowing defects and errors to go undetected and appear in the live software;
- **Poor documentation**: happen when the software system has poor documentation, which can lead to difficulties in understanding the functionalities, requirements and architecture of the system.

Most of the concepts above described will be further explored in the following sections.

II. Error detection and reporting

A. Methods and best practices for detecting and reporting

One of the most important aspects of software development is finding and fixing mistakes. Early error detection can prevent expensive and time-consuming fixes later on. As a result, developers employ a range of methods to find and fix errors in their code. Those are very diverse, but we will focus on the 4 most common used to identify and diagnose issues in software:

- **Static Code analysis**: In this method, the code is examined without being run. Static analysis tools examine the code for problems like grammar mistakes, bad code habits, and possible security flaws early in the production cycle, before the code is compiled or executed. Even though it can detect errors early in the development cycle, prior to the code execution, it cannot detect errors generated in runtime or caused by wrong inputs. This makes it ideal for large codebases and complex software projects, as well as for finding writing style errors, security flaws and other related problems.
- **Dynamic code analysis**: This method includes looking at the code as it runs. Dynamic analysis tools watch the code while it is being executed. Issues that are challenging to discover using static analysis can sometimes be found using this method. The main advantage of using this method is the fact that it can detect runtime errors, performance bottlenecks, memory leaks and others types, but, unfortunately, it does not guarantee detection of all errors and is computationally expensive. This method is ideal for smaller codebases and less complex software projects.

- **Testing**: This technique involves testing the software with the aim of finding faulty behaviour. *Unit testing*, *integration testing*, and *system testing* are a few examples of the various testing stages. While integration testing examines how various software modules interact, unit testing focuses on evaluating specific software modules or components. System testing entails putting the complete system to the test to make sure it complies with the criteria. Testing can identify mistakes at various software development stages, from a single module to the complete system, but it is time-consuming, computationally expensive, and cannot guarantee that the software has no errors. Those aspects make it useful for finding practical errors in code and for making sure the software complies with the requirements. System testing is the best option for testing the complete system, while integration testing is helpful for testing how various modules interact with each other.
- **Runtime monitoring**: This method includes watching the program in real-time for mistakes, exceptions, and failures. It can be applied while a system is operating to find problems that might not have been found during development or testing. A positive aspect of this method is that it can identify problems in production processes and give real-time feedback. Unfortunately, it can be complex to implement and might hinder the performance. In short, it's a useful method for monitoring production systems and detecting errors that may have gone undetected during development or testing, which makes it ideal for complicated software systems and large-scale software systems.

Error reporting refers to the process of identifying and communicating errors or issues that occur within a software system. When an error or issue is detected, it is important to collect as much information as possible to help diagnose and resolve the problem. The process of error reporting typically involves several steps, including:

1) **Identifying the error**: The first step in error reporting is to detect abnormal behaviour and identify the error or issue that has caused it. This can be achieved resorting to various error detection techniques such as the ones presented above.
2) **Collecting information**: After the error has been identified, it's critical to gather as much information as possible. This information differs based on the nature of the software, but the most common types of information are the type of error, its origin and any relevant information about the state of the system at the time of the error.
3) **Organizing the information**:After gathering pertinent data, we need to arrange it in a way that will aid in the error's detection and repair, which means classifying the data, making a timeline of the incidents that led to the mistake, or applying other organizing strategies.
4) **Communicating the information**: The information regarding the error must be shared with interested parties, such as development teams, quality testing teams, or end users. This may involve creating reports or dashboards that summarize the error-related information in a clear and concise manner.

Effective error reporting is important to ensure a smooth software operations. An error reporting mechanism facilitates the process of debugging and troubleshooting, which leads to continuous improvement by giving developers insights into the system's behavior. There are several mechanisms that can be used for error reporting, which includes log files that contain the historic of events and actions, error messages - a type of error or issue notification -, alerts that contain the Warning of critical errors or issues, and telemetry data, a type of relevant data collected during the system execution.

III. Error handling and recovery

A. *Definitions*

Error handling and recovery possess a variety of definitions, each one acting and presenting in a different way than the others. The process of identifying, reporting, and resolving errors is called **Error Handling**, which tries to identify the root cause of the error and tries to take action to recover from it, whilst also preventing its recurrence.

This leads us to the definition of **Recovery**, the process of fixing an issue and getting a system or program back to functioning. Recovery might involve a number of tactics, including going back in time, restarting the system, fixing damaged data, or taking precautions to prevent problems in the future.

Those terms are very different from an **Exception**, which is a type of error that occurs during the execution of a program or system and typically disrupts its normal flow, often caused by runtime errors or conditions that are difficult. Some examples may be division by zero, timeouts and files not found.

B. *Error handling techniques*

There are several techniques to handle errors. A common technique is **Exception Handling**, which involves raising an exception whenever an error occurs and is integrated into the syntax of several programming languages. The program has the ability to handle exceptions in a systematic way, i.e, it follows the structured approach of identifying, catching, handling and recover the exception.

Another technique is **Logging**. It involves keeping track of events that take place when a program is running. Some examples of logging can be error messages, cautionary notes, and debugging data. In order to help the developer in analysing and debugging, this technique helps them keep track of what happened within the program.

When a software makes assumptions, **Assertions** are used to check them. They are often used to check that the software is functioning properly and to identify issues early in the development phase. In testing and debugging, assertions are frequently used to assist find faults before they have an impact on the live system. For example, this can be done in Promela with the *assert()* command.

To make sure a program can manage unexpected input or mistakes, we can use the **Defensive Programming** technique. In this approach, the application must be designed to handle exceptional situations and incorrect inputs. This technique can enhance the program's dependability by assisting in error prevention and some methods can be used in this technique, such as **Validation** and **Input Sanitization**, which ensure that input data is safe and valid for the program. One specific example is a smart home system, where those methods ensure the security and reliability of the system. Device settings and sensitive user data are often handled by smart home applications and devices, making them easy targets for hackers. Validation ensures that user-provided data is accurate and adheres to predetermined standards or limitations. This involves making sure the input data has the right type, falls inside a suitable range of numbers, or complies with any formatting specifications [1].

C. Best Practices and Recovery Strategies

A few best practices and recovery procedures are needed to improve the resilience and stability of software applications. Each method focuses on a distinct aspect of failure management and aims to lessen the impact of errors on the program's overall functionality and user experience. Enhancing user experience, system stability, maintenance, fault tolerance, resilience, error recovery, and error handling are some of the benefits of employing these procedures. All of this results in a more robust system, but there may be drawbacks, such as increased system complexity, challenging testing, performance overhead, potential growth in code complexity, and many more.

There are some aspects to take into account when selecting the best mechanisms. For instance, we need to understand where the failure comes from and how it occurred, understand the architecture of the system so that better choices can be made to prevent specific errors, understand the complexity and performance trade-offs by selecting certain mechanisms and understand how every decision made in the selection of the mechanisms can impact the user experience. It's important to balance every one of those aspects in order to make use of the best practices and recovery strategies.

An important practice in software development is called **Graceful Degradation**, which focuses on allowing the system to continue to operate even when certain components break down or have problems, but with a decrease in the performance. The main objective is to deliver a positive user experience in the face of errors thanks to redundancy, fault tolerance, and modular architecture that allow the system to operate at a reduced capacity, while still allowing users to access essential functionality even when some system components aren't working properly. One concrete example of this is Netflix's Chaos Monkey, where it intentionally introduces faults into their systems to ensure that their service degrades gracefully when encountering issues [2].

Retry is another technique, which consists of attempts to carry out a failed operation more than once expecting that the operation will finally succeed. Retries can aid systems in recovering from mistakes like network problems or service interruptions. However, retrying an operation too soon might put too much stress on the system, escalating the issue or leading to further failures. To address this, we can use **Exponential backoff**, which expands exponentially the interval between retries. It begins with a brief initial delay and increases by a factor before the subsequent try after each unsuccessful attempt. This strategy provides the system time to recover and avoids overtaxing the target service or resource. For instance, Amazon Web Services [3] or Microsoft Azure [4] provide SDKs that implement this method, which handles faults and improve the resilience and fault tolerance of the client applications.

Another famous and commonly used technique is the **Rollback**, which is the operation of returning the system to a condition that existed before an unsuccessful operation. When one of the phases in a distributed system or multi-step process fails, the system may end up in an inconsistent state. In such cases, the system can either undo the modifications made during the operation or execute several actions that reserve the effects of the failed operation, which is the opposite of doing a rollback. When using PostgreSQL, for instance, a transaction may be used to aggregate a number of SQL actions into a single atomic unit, guaranteeing that either every operation is correctly carried out or that none of them take effect. A rollback won't take the operation into account and will put the system back to the last checkpoint, thanks to a method known as **Checkpointing** that periodically saves the state of a system.

So many more methods exist and all of them have their uses for every type of situation, but it is up to the system architect, developers and quality assurance engineers to select the appropriate mechanisms and understand their benefits and trade-offs based on the system architecture and business model.

D. Metrics

To measure the effectiveness of error handling and recovery strategies according to the different evaluation criteria above defined, various metrics may be used.

- **Error detection rate**: Counts the number of errors that were successfully detected. An high error detection rate indicates that the system is effective in detecting errors.
- **Recovery rate**: Counts the numbers of errors that the system successfully recovered from. If the recovery mechanism has a high recovery rate, then the system is effective in correcting errors.
- **Mean time to recover (MTTR)** and **Mean time between failures (MTBF)**: These metrics measure the average time taken to recover and the average time between the occurrence of failures in the system respectively. A low MTTR is associated to a resilient system, while a high MTBF is related to a more reliable and robust system.
- **Error response time** and **Error resolution time**: These metrics monitor the time it takes the system to respond to an error and the time it takes to correct that error,

respectively. A small error response time and a small resolution time indicate that the handling system has a high performance.

- **Failure Rate**: Measures the amount of failures that occur within a time window. A low failure rate means that the system is reliable.
- **False Negative and False Positive rates**: Metrics used to measure the amount of times the system warns about an error that is not in fact an error (false negative) and the amount of times the system fails to warn about an error (false positive). In software quality, false positives are far more dangerous than false negatives.
- **Cost of recovery**: Measures the costs required to recover from an error.

IV. Emerging Trends

Error handling and recovery strategies have come a long way in the software development history, and they shall continue that way. Some trends are starting to emerge and their main goal is to automate and facilitate even more the quality review process of any software, as well as to better manage errors and recover from them, which in turn improves the quality of software products.

One of those emerging trends is **Machine learning-based error handling**, which is a technique that can be used to predict errors before they occur and suggest recovery strategies. The thesis by Justin Fennis on ”Machine Learning solutions for exception handling” [5] talks about key aspects related to this field, like the importance of agile methodologies and practices such as Scrum and Kanban, which encourage iterative development, collaboration, and continuous improvement. Other important aspects of this thesis are related to the importance of Continuous Integration and Continuous Deployment, Automated Testing, Test-driven development, Error monitoring and Logging tools.

Sometimes the software team might not get any progress or the expected results when errors occur, that’s why it may be important to introduce **Chaos engineering** [6], which is another technique where the goal is to deliberately introduce errors into the system in order to find vulnerabilities and boost the resilience of the system, and it does so by assessing the system’s capacity to resist unforeseen events and recover quickly. There’s even a Chaos Engineering Slack [7] where engineers from across the world uncover critical weaknesses in their systems and level up their teams. People can ask and answer questions, as well as make connections with software engineers from big companies such as Google and Facebook.

But what if the software team wants to make informed decisions about how to allocate resources and prioritize improvements? That’s where **Error Budgeting** [8] comes to aid, as it allows to establish the maximum level of unreliability that is acceptable for the service during a certain time and acts as a warning for when you need to take remedial action. One famous example of a company that uses Error Budgeting is Google. The book *”Site Reliability Engineering: How Google Runs Production Systems”* [9] highlights the importance of tracking error budgets over time to identify trends and patterns that can inform future improvements. The book places a strong emphasis on the value of cooperation and communication throughout the Error Budgeting process, especially between the development and operations teams. The authors mentions that in order to guarantee that the Error Budget is not exceeded, development teams must prioritize reliability enhancements and collaborate to define realistic Error Budget objectives.

Following a similar principle to the Machine learning-based error handling, we can use **Automated Error Detection and Recovery** tools to detect and recover from errors without any human intervention. The key difference between those two methods is that one involves using machine learning algorithms to predict errors before they occur, which can be very costly, while the other uses tools and processes to detect and initiate recovery processes. Let’s talk about several tools:

- The famously known AWS owns a service called **AWS Lambda** [10], a ”serverless compute service that runs your code in response to events and automatically manages the underlying compute resources for you”. It works in a very easy way: we only need to write the code and upload it as a .zip file or container image.
- To automate solutions for infrastructure problems, **Icinga** [11] and **Nagios** [12] are excellent choices, the latter being described as ”a powerful monitoring system that enables organizations to identify and resolve IT infrastructure problems before they affect critical business processes”, incorporating automated processes to do so.
- **Splunk Enterprise Security** [13] is able to ”combat threats, protect your business and mitigate risk at scale with analytics you can act on” and it can be integrated with other services such as AWS, Mongodb, Kafka and so on.
- Another useful tool for automated security monitoring and debug latency is **New Relic** [14], as it allows to do monitoring for Kubernetes, Mobiles, Synthetics, Serveless, Infrastructures and so on.
- **PagerDuty** [15] allows to ”Automate, orchestrate, and accelerate responses across your digital infrastructure” by using Process Automation and AIOps with automated self-service tasks.

There are many more emerging trends, but we deemed those most important, as they will have an extreme impact in the future of software quality and everything that it surrounds, as there is a growing number of automated tools, services and products at our disposal, and the methods are becoming more intelligent and the machine learning algorithms are becoming more vast and complex.

V. Conclusion

This survey covered several aspects and components related to error handling and recovery strategies in software development. We made an overview of the most important software quality and dependability concepts and terminology to explain the background and critical importance of errors in the context of software development, as well as defining the different types of errors that can happen in a software system.

To be able to utilize the methods and strategies at our disposal, it was necessary to first refer how error detection and reporting works, as it is a crucial aspect of software quality that helps the errors before they occur. We talked about several methods, such as static code analysis and runtime monitoring, while also providing their trade-offs and usefulness.

Following those errors detections and reports, we also mention the way we can handle and recover from them, which we explain by defining several key concepts and techniques that are essential to the development of software. These methods let software systems identify faults and recover from them quickly and effectively thanks to some best practices and strategies that are used nowadays, such as graceful degradation and rollback. One lesson that can be learned from successful error handling and recovery strategies is the importance of continuous testing and monitoring. Additionally, effective error handling and recovery strategies often require cross-functional collaboration between developers, operations teams, and quality assurance teams. By working together, it is made possible to incorporate error handling and recovery techniques early on in the software development process and to make sure they are consistent with the organization's overarching objectives.

We finished the survey by talking about new trends and technologies such as automated error detection and recovery tools. By automating and streamlining the software quality review process and enhancing the resilience and dependability of software products, these technologies will revolutionize the way we handle error management and recovery for the next few years.

One final observation must be made about the number issues and future research options that remain. The approaches that want to include machine learning algorithms into the development process still require further study, even though machine learning-based error management has been making huge progress, proving that it has a lot of promise. In addition, additional study is required to find the best strategy for putting chaos engineering and error budgeting into practice. Even though they are promising strategies to increase system resilience and dependability, they will have to compete against the other non-human related methods and will need to continue to prove their usefulness.

References

[1] Maria Teresa Rossi et al. "Defensive Programming for Smart Home Cybersecurity". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). Sept. 2020, pp. 600–605. DOI: 10.1109/EuroSPW51379.2020.00087.

[2] *Home - Chaos Monkey*. URL: https://netflix.github.io/chaosmonkey/ (visited on 04/04/2023).

[3] *Error Retries and Exponential Backoff in AWS - AWS General Reference*. URL: https://docs.aws.amazon.com/general/latest/gr/api-retries.html (visited on 04/04/2023).

[4] martinekuan. *Azure Service Retry Guidance - Best Practices for Cloud Applications*. Mar. 13, 2023. URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/retry-service-specific (visited on 04/04/2023).

[5] J. Fennis. *Machine Learning Solutions for Exception Handling*. Mar. 20, 2019. URL: https://essay.utwente.nl/77555/ (visited on 04/06/2023).

[6] *What Is Chaos Engineering? Chaos Engineering and Its Principles Explained*. IT Operations. URL: https://www.techtarget.com/searchitoperations/definition/chaos-engineering (visited on 04/06/2023).

[7] *Join the Chaos Engineering Slack*. URL: https://www.gremlin.com/slack (visited on 04/06/2023).

[8] *A Complete Guide to Error Budgets: Setting up SLOs, SLIs, and SLAs to Maintain Reliability*. Sept. 22, 2022. URL: https://www.nobl9.com/resources/a-complete-guide-to-error-budgets-setting-up-slos-slis-and-slas-to-maintain-reliability (visited on 04/06/2023).

[9] Jennifer Petoff et al. *Site Reliability Engineering: How Google Runs Production Systems*. 1st edition. Beijing ; Boston: O'Reilly Media, May 10, 2016. 550 pp. ISBN: 978-1-4919-2912-4.

[10] *Serverless Computing - AWS Lambda - Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/lambda/ (visited on 04/06/2023).

[11] *Icinga Monitor Your Entire Infrastructure with Icinga*. URL: https://icinga.com/ (visited on 04/06/2023).

[12] *Nagios - The Industry Standard In IT Infrastructure Monitoring*. Nagios. URL: https://www.nagios.org/ (visited on 04/06/2023).

[13] *Splunk — The Key to Enterprise Resilience*. URL: https://www.splunk.com/ (visited on 04/06/2023).

[14] *New Relic — Monitor, Debug and Improve Your Entire Stack*. URL: https://newrelic.com/ (visited on 04/06/2023).

[15] *PagerDuty — Real-Time Operations — Incident Response — On-Call*. PagerDuty. URL: https://www.pagerduty.com (visited on 04/06/2023).