# A Standard for Static Code Analysis of Critical Systems


By

**Rida Shaukat**

00000275226

Supervisor

Dr. Tauseef Ahmed Rana

A thesis submitted to the faculty of Software Engineering Department,
Military College of Signals, National University of Science and Technology,
Islamabad, Pakistan, in partial fulfilment of the requirements for the degree for the MS in Software Engineering

September 2022

# ABSTRACT

The worth and precision of a software system is determined by the quality of source code and the degree to which the source code under consideration, satisfies the software quality parameters. As a matter of fact, until recent years, this aspect of software quality was not given due significance and the core focus of the underlying software system was on the feature implementation and the extent to which the system fulfills the functionality for which it was developed. It is worth-noting that as a consequence of the legacy approach, the IT industry and the giants of this industry faced significant losses only because the software system was not tested fairly to figure out potential and hidden defect in the source code, which ultimately led the system towards complete failure. We have a number of instances where the companies faced unbearable losses due to the hidden flaws in the source code. If these hidden defects would have been pointed out during system testing phase, those systems wouldn't have collapsed during production phase. The approach we are suggesting here is Static Code Analysis. This approach aims to test the source code against a set of guidelines based upon software quality indicators, are pre-defined and developed. Analysis of source code is conducted against these rules. Now, it is worth noting that we have multiple static code analysis tools available in the market, our primary concern here is that none of the tools available provides a go-to solution. Our aim is the research and development of such a static code analysis tool which checks the source code against critical rules pertaining to code quality. We will accumulate all rules for some specified quality parameters related to software quality from multiple coding standards and widely used tools and devise a comprehensive ruleset which would be an all-in-one solution for the system testers who want to test the software system for critical violations. Our aim behind this research and specifically targeting critical systems is that these type of signals are developed with high development costs and efforts and can risk human lives, or cause heavy financial damages if led towards failure. Therefore we aim to devise a comprehensive rule-set based upon a few quality parameters to make sure that it provides a go-to solution for the underlying software quality aspects and critical systems can be tested for those quality parameters making sure that no aspect is missed out and the violations detected by the tools developed based upon the underlying standard are capable enough of pointing out all potential issues and shortcomings in the source code.

# ACKNOWLEDGEMENT

I am thankful to Allah Almighty for his countless blessings and bestowing me with the courage strength in my work through out the research phase. Words cannot express my gratitude towards my Supervisor Dr. Tauseef Ahmed Rana for his valuable support and feedback. I would like to acknowledge my respected GEC Members Dr. Nauman Ali Khan and Dr. Imran Qureshi for their help and assistance throughout my research phase. I express my deepest thanks to my parents and family for their support and prayers. Their guidance and encouragement gave me immense support throughout the project.

# DEDICATION

To the Allah Almighty

&

My parents and faculty

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

*Chapter 1*

# 1. INTRODUCTION

This section provides a brief introduction regarding the necessity, significance of code analysis. It also highlights the difference between two major techniques of code analysis. Critical events signifying the importance of code analysis are presented and the research objectives for the research study are identified. Finally, the research methodology adopted has been presented.

## 1.1 Code Analysis:

The determination of the quality of a software or the source code is a debatable issue. As a matter of fact, it has been observed that until recently, code quality and the necessity to determine it's eminence, was not given due importance, as long as the system delivered correct results, consequently we have witnessed a number of critical systems going into heavy losses due to system failure when it was put into production. One of the core reason of systems going into losses was due to the potential defects in the source code. Now, the discussion was that when source code is compiled, even after all of the compile time and run time errors were fixed; even then during execution some software systems crashes. So, after deep research and study, the software developers and system testers reached the conclusion that apart from syntax errors and logical errors there is some factor which needs to be looked into and then software system should be tested against it. Hence, the term code analysis was introduced. It aims to test the software system based upon certain software quality parameters. The foundation of this assessment is a rule-set; it contains a set of best practices which must be followed during development phase and source code should then be tested against these practices to find out the hidden potential defects and short-comings in the code. The hidden defects in the source code caused system failure in multiple cases since these defects were neither pointed out nor addressed in the system testing phase.

## 1.2 Difference between Code Analysis Techniques:

Static code analysis targets to detect the potential imperfections in the software / source code before it is sent into production. It tests the source code for defects without actually executing the source code. Static code analysis is basically an assessment of source code before sending the software system into production. Static analysis is executed upon the source code against a static

set of pre-defined guidelines / rules, before the system is executed. The source code under consideration is provided as an input the static code analysis system and then the tool compares the provided source code against a pre-defined set of rules, and after analysis provides the violations in the source code which have been found during the analysis.

On the other hand, dynamic code analysis implies to the examination of source code against a standard/ set of rules during the actual execution of source code. The technique which we will be taking into account during research phase is static code analysis.

## 1.3 Incidents signifying the importance of Static code analysis:

Let's take a look at a few cases where the absence of an appropriate testing phase for critical systems was missed out and as a result the entire world saw huge systems going into failure.

As evident, static code analysis tools are of critical significance; it is worth-noting that threats increase exponentially is the underlying system is a critical system, due to the associated losses which are unbearable. Critical cases from past, highlight its significance and eminence. ARIANE 5, was a heavy-lift rocket, and was projected in order to take the satellites into geostationary orbit or low Earth orbit. This system unfortunately exploded right after 40 seconds of it's execution / launch. These associated development costs mounted upto $7 billion. After rigorous analysis and testing, it was surprising to discover that failure occurred due to a minor software error in the system. During system execution, a 64-bit floating point number converted into a 16-bit signed integer. Therefore, this numerical value being greater than the highest value storable in this data-type, made the conversion to fail, resulting into a series of wrong executions, which led to system explosion.

We have another case, where appropriate application of static code analysis could have saved a massive destruction i.e. Mars Climate Orbiter. The associated costs were around $125 million. The system failure occurred due to difference in units being used by different teams. One engineering team used the English units while the metric was used by the other team. Hence, the information failed to transfer between the spacecraft team and mission navigation team and led to failure. Consequently, now developers have started applying static code analyzers in testing phase, to evaluate the value of source code in terms of quality and productivity, before the system is sent into production.

## 1.4 Research Questions:

Keeping in view, a few of many incidents that occurred and resulted in heavy losses financially as well as costing human lives, we will identify the gap between the ideal case of detecting potential defects and the current approaches. It is worth-noting that a number of standards are available and we have seen some tools as well which provide static code analysis feature, but it is worth taking into account that we don't have a single comprehensive coding standard for static analysis which can be used as go-to-solution. Each standard, or tool implementing rules are providing different rule set with rules of different complexity levels and criticalities.

RQ1: Do all static code analysis basis for existing tools are similar?

RQ2: Do all existing rule-sets provide insight regarding similar software quality dimensions?

RQ3: Is the taxonomy of current analysis techniques similar or same?

RQ4: Does any existing rule-set provide a comprehensive list of checks/rules for any software quality parameter/dimension?

## 1.5 Research Objectives:

Our aim is to devise a comprehensive set of rules which can be used for conducting analysis of source code. Our aim is to develop a Rule-set in which we will research and integrate the existing rules of critical nature to provide a comprehensive rule-set to be used for assessing code quality and ruling out the hidden discrepancies in source code, which may ultimately lead to failure during system execution. We have identified our research objectives as follows:

- Research the existing major standards of static code analysis.
- Research the foundation of widely known tools for static code analysis.
- Identify the quality parameters for developing standard / rule-set.
- Develop a comprehensive rule-set / standard for static code analysis.
- Develop a prototype for reflecting the usage of developed rule-set / standard.

## 1.6 Research Outline:

This research work / thesis consists of the following chapters:

**Chapter 1:** This chapter briefly introduces the significance of code analysis. Further the major techniques for conducting analysis are also presented. Major incidents highlighting the losses occurred due to absence of proposed technique are presented. Finally we have collected research questions and presented our research work objectives.

**Chapter 2:** This chapter presents the level of study and research already carried out in the concerned area, research work of scholars and researchers has also been presented as research work.

**Chapter 3:** This chapter presents the scope of our research. Software quality is a vast term, therefore we've selected and presented a few software quality parameters for carrying out our research.

**Chapter 4:** This chapter presents an analysis of the existing technique for static code analysis. This phase has majorly provided us with gaps between present researches, thereofore we have covered the gap as a research in next chapter.

**Chapter 5:** In this chapter, we have presented an analysis of existing researches, and finally concluded our research in the form of tabular presentation of rule-sets. Each rule-set has been presented separately based upon the underlying software quality parameter.

**Chapter 6:** In this chapter, we have provided results and technical discussion for the obtained results.

**Chapter 7:** In this chapter, we have presented a part of our research work as a prototype, presenting how it will be implemented as a tool.

**Chapter 8:** In this chapter, conclusion of findings and research work directions have been identified and presented.

**Chapter 9:** This chapter contains the references of the existing research work which provided as a basis for our research and study.

*Chapter 2*

# 2. PRELIMINARY STUDY

## 2.1 Level of research already carries out in the underlying domain:

Researchers have rigorously worked in the domain of static code analysis and have developed coding standard to meet the testing criteria to some extent. It is worth-noting that no single coding standard exists which claims to be an all-in-one solution for testing coding quality and coding discrepancies, before sending the software system into production. Some standards which already exist our JPL, Misra, Microsoft Guidelines. Different domains of coding standards are covered by each of these standards, but none of them presents a comprehensive go-to solution in the form of rule-set. It is worth-noting that all the tools available conduct static code analysis upon source code based upon all levels of rule criticality; but, this is to be taken into account that in this case, the cost of conducting analysis would be higher for generic systems, whereas in critical systems we need a comprehensively established rule-set only which ensures that the underlying standard / rule-set will test the source code in all aspects for particular software quality aspects. Besides, our research will be based upon the formation of a coding standard in which we will comprehensively establish a rule-set. The static code analysis after code compilation will then be conducted upon source code against the proposed standard (rule-set).

## 2.2 Literature Review:

The author [1] here mentions that code quality static code analysis tools are being employed for testing code quality and discrepancies in code, likewise a number of such tools exist in market with each tool providing basis for a different or programming language, but still there are quite few tools that provide support for domain specific languages. Another issue highlighted by the author is that no tool clearly states the rule set it is applying for static analysis, neither do they mention the type of defects which will be detected by the tool. Secondly, another major challenge is that almost all tools make use of different taxonomies.

The researchers [2] have signified the importance of static code analysis tools and re-enforced that appropriate utilization of static code analysis tools during early development phases could significantly reduce the work to be redone. The researchers have further conducted an

analysis upon static code analysis tools of Java, the aim behind this research was to conduct a comparative analysis of tools to against a set of parameters i.e. Input type, Availability, Rule-set, extensibility, report type, error type, violations. The analysis was mostly based upon the rule-set of 2 coding standards for Java i.e. JPL and Rule of Ten.

The paper [3] signifies the importance of the application of static code analysis tools during initial phases of software development unlike the dynamic code analysis technique which implies to detect code quality issue during source code execution. The paper also presents a comparative analysis of multiple code analysis (static) tools available for Java, C, C++. It is found that for C/C++, CPPCheck detected the maximum number of violations, and for Java, Find bugs analysis tool reported the maximum number of violations, which were not detected by other tools present in study.

The authors [4] have presented an analysis after conducting research upon multiple static code analysis tools. Further they've identified and presented some of the techniques and presented them as: Starting the code analysis once all defects are fixed, running the analysis after correcting any of the previously detected defect, executing analysis tools by external tool calling procedure during software project development, executing analysis tools by embedding tools in the development IDE where the software project is developed.

The authors [5] have signified the importance of static analysis tools. The major focus regaring the underlying research presented in the paper is to conduct an in-depth analysis of a trending code analysis tool i.e. Coverity Scan. It conducts dynamic code analysis as well as static code analysis. The focus during this research was on static code analysis conducted by the tool. The research presents the types of defects covered by the tool. The process includes majorly 2 phases: Analysis of data flow between procedures, Statistical investigation of data. The major defect checkers implemented in the tool are: Null returns, forward null, reverse null, unused value, reverse negative, return local, reverse null, unused value, stack use, resource leak, checked return, deadcode, UNINIT.

This paper [6] high-lightens the importance of static code analysis tools especially when the underlying software project is to be deployed or to be used in the domains of mission-critical or safety critical systems. The aim of the research is based upon the checking of software quality

constraint of ‘maintenance’ by the static code analysis tools. Maintenance of software project is a critical factor in updating, extending or defect handling in the source project.

This research [7] has been based upon the theme of software assurance. As we all know that software security is a critical step in the software systems. Recently, there has been a rapid growth in cyber attacks. It is worth noting that despite the use of IDS - Intrusion Detection Systems as well as IPS - Intrusion Prevention Systems, the firewalls and many other such mechanisms to prevent the software system against security breach, still the cyber attacks are not completely controlled, this due to the fact that mentioned systems only assist in minimizing the security threats but do not control the base from which the software vulnerabilities are exploited. The cyber attacks and the associated losses can only be controlled if appropriate software security checks are applied in the underlying software systems. A Security Code Analysis (SCA) is the solution and step-ahead towards the cyber security issues. A few tools have also been analyzed and compared in the study.

*Chapter 3*

# 3. SCOPE OF RESEARCH

## 3.1 Software Quality Parameters:

Software quality is a vast term and it would be impractical to achieve software quality fully. As our focus is upon software systems of critical nature, therefore we aim to research and pick out all of the software quality aspects and their respective checks in order to be assure that the underlying software system which is to be sent into production is bug-free and there lies no such code construct or potential defect in the source code which might lead the system towards complete failure. Rule-set of software quality is based upon multiple factors and categories of checks including:

- Design Rules
- Naming Rules
- Globalization Rules
- Performance Rules
- Security Rules
- Usage Rules
- Maintainability Rules
- Portability Rules
- Interoperability Rules
- Reliability Rules
- Architecture Rules
- API breaking changes
- Code Coverage
- Dead code

- Code Smells
- Code smells regression
- Visibility
- Source Files Organization
- Naming Conventions
- Object Oriented Design
- API usage
- .NET Framework usage
- Performance Rules
- Immutability
- Single-file Rules
- Reliability Rules
- Style Rules

## 3.2 Our Scope of Research:

For research purpose, we have narrowed the scope of rule-set used in the static code analysis. Since our aim is to identify rules for static code analysis of critical systems, therefore we have taken into consideration 3 parameters of quality. For being sure that we don't miss out any aspect or dimension within a quality parameter, we've narrowed down our research to 3 quality parameters. We will conduct research on the rules for the selected categories so that no rule is missed out for those categories and the proposed standard can be used as a go-to-solution for the underlying quality parameters in the code quality assessment for critical systems. The category of rules we've chosen for research purpose are as follows:

- Design Rules

Such rules that detect potential design flaws in the code of software system, these coding errors usually do not affect the execution of your code, but may become a cause of failure in an unprecedented scenario of events.

- Maintainability Rules

  Rules which detect issues pertinent to software maintenance or scalability.

- Usage Rules

  Rules that detect potential flaws in code assemblies which can affect code execution.

*Chapter 4*

# 4. ANALYSIS OF TECHNIQUES FOR STATIC CODE ANALYSIS USED IN EXISTING TOOLS:

We have chosen a few widely known static code analyzers present in market, we have researched the foundation of these based upon which these tools conduct analysis. We have been able to find out the rule-set implemented in these tools based on which these tools conduct static code analysis upon software system and as an output present the shortcomings in the source code with respect to the underlying rule-set. The violations presented by these tools are of different categories based upon the underlying rules, but since our research caters three quality parameters i.e. Design, Maintainability, Usage; therefore we've researched these tools majorly sorted out rules pertinent to the code quality parameters which are an inherent part of our research.

## 4.1 FxCop:

FxCop is a .net framework's development tool. It's core feature is to perform assessment of managed code. The core task is to keep check on any indiscretion or incongruity with the rules put out by the standard Design Guidelines.

The Microsoft guidelines assist in preparing manageable and flexible programming, by making use of the .net framework. The tool Fxcop is developed for analysis and to be used as a desktop application with a user-friendly and detailed GUI as well as a CLI in case the scope of execution lies beyond the range of visual studio.

Contrasting with the other tools that scan source codes, Fxcop analyses the compiled object code. Fxcop static code analyser analyses the source code for any of the 200 probable violations of the coding standards in the fields mentioned below:

- COM (Interoperability) Rules
- Design Rules
- Usage Rules
- Globalization Rules

- Performance Rules
- Maintainability Rules
- Portability Rules
- Reliability Rules
- Security Rules
- Naming Rules

| S. No. | Rules/Metrics | Category | Standard / Tool |
|---|---|---|---|
| 1. | Static members should not be declared on generic types | Design | FxCop |
| 2. | Do not expose generic lists | Design | FxCop |
| 3. | Use generic event handler instances | Design | FxCop |
| 4. | Generic methods should provide type parameter | Design | FxCop |
| 5. | Avoid excessive parameters on generic types | Design | FxCop |
| 6. | Do not nest generic types in member signatures | Design | FxCop |
| 7. | Use generics where appropriate | Design | FxCop |
| 8. | Enums should have zero value | Design | FxCop |
| 9. | Collections should implement generic interface | Design | FxCop |
| 10. | Passing of the base types as parameters should be considered | Design | FxCop |

| | | | |
|---|---|---|---|
| 11. | Constructors should not be used by abstract types | Design | FxCop |
| 12. | Overload operator equals on overloading add and subtract | Design | FxCop |
| 13. | Indexers should not be multidimensional | Design | FxCop |
| 14. | Params array is better than repetitive arguments | Design | FxCop |
| 15. | Default parameters should not be used | Design | FxCop |
| 16. | Use events where appropriate | Design | FxCop |
| 17. | Do not catch general exception types | Design | FxCop |
| 18. | Implement standard exception constructors | Design | FxCop |
| 19. | Nested types should not be visible | Design | FxCop |
| 20. | ICollection implementations have strongly typed members | Design | FxCop |
| 21. | Override methods on comparable types | Design | FxCop |
| 22. | Lists are strongly typed | Design | FxCop |
| 23. | Use integral or string argument for indexers | Design | FxCop |
| 24. | Properties should not be write only | Design | FxCop |
| 25. | Equals operator should not be overloaded on reference types | Design | FxCop |

| | | | |
|---|---|---|---|
| 26. | Protected members should not be declared in sealed types | Design | FxCop |
| 27. | Virtual members should not be declared in sealed types | Design | FxCop |
| 28. | Static holder types are preferable to be sealed | Design | FxCop |
| 29. | Constructors should not be used by static holder types | Design | FxCop |
| 30. | URI return values are not preferable in the form of strings | Design | FxCop |
| 31. | Certain base types should not be extended by Types | Design | FxCop |
| 32. | Members should not expose certain concrete types | Design | FxCop |
| 33. | Exceptions should be public | Design | FxCop |
| 34. | Avoid excessive complexity | Design | FxCop |
| 35. | Differentiation between identifiers should be by more than one case | Maintainability | FxCop |
| 36. | Types that own disposable fields should be disposable | Design | FxCop |
| 38. | Mark assemblies with AssemblyVersionAttribute | Design | FxCop |
| 39. | Child types should be able to call Interface methods | Design | FxCop |
| 40. | Types that own native resources should be disposable | Design | FxCop |

| | | | |
|---|---|---|---|
| 41. | Base class methods should not be hidden | Design | FxCop |
| 42. | Exceptions should not be raised in unexpected locations | Design | FxCop |
| 43. | P/Invoke entry points should exist | Design | FxCop |
| 44. | Dispose objects before losing scope | Design | FxCop |
| 45. | Do not indirectly expose methods with link demands | Design | FxCop |
| 46. | Override link demands should be identical to base | Design | FxCop |
| 47. | Types must be at least as critical as their base types and interfaces | Design | FxCop |
| 48. | Do not dispose objects multiple times | Design | FxCop |
| 49. | Disposable fields should be disposed | Design | FxCop |
| 50. | Mark all non-serializable fields | Design | FxCop |
| 51. | Implement IDisposable correctly | Usage | FxCop |
| 52. | Avoid duplicate accelerators | Usage | FxCop |
| 53. | Wrap vulnerable finally clauses in outer try | Usage | FxCop |
| 54. | Default constructors must be at least as critical as base type default constructors | Usage | FxCop |
| 55. | Objects with weak identity should not be locked | Usage | FxCop |

| | | | |
|---|---|---|---|
| 56. | Pointers should not be visible | Usage | FxCop |
| 57. | Methods must keep consistent transparency when overriding base methods | Usage | FxCop |
| 58. | Rethrow to preserve stack details | Usage | FxCop |
| 59. | Value type static fields should be initialized inline | Usage | FxCop |
| 60. | Overridable methods should not be called in constructors | Usage | FxCop |
| 61. | Finalizers should call base class finalizer | Usage | FxCop |
| 62. | Declare event handlers correctly | Usage | FxCop |
| 63. | Avoid namespaces with few types | Usage | FxCop |
| 64. | Avoid out parameters | Maintainability | FxCop |
| 65. | Avoid empty interfaces | Maintainability | FxCop |
| 66. | Do not pass types by reference | Maintainability | FxCop |
| 67. | Avoid excessive inheritance | Maintainability | FxCop |
| 68. | Review misleading field names | Maintainability | FxCop |
| 69. | Avoid unmaintainable code | Maintainability | FxCop |
| 70. | Avoid excessive class coupling | Maintainability | FxCop |
| 71. | Resource string compound words should be cased correctly | Maintainability | FxCop |

| | | | |
|---|---|---|---|
| 72. | Casing is very important especially in compound words | Maintainability | FxCop |
| 73. | Resource strings should be spelled correctly | Maintainability | FxCop |
| 74. | Identifiers should be spelled correctly | Maintainability | FxCop |
| 75. | Identifiers should not contain underscores | Maintainability | FxCop |
| 76. | Identifiers should be cased correctly | Maintainability | FxCop |
| 77. | Identifiers should have correct suffix | Maintainability | FxCop |
| 78. | Identifiers should not have improper suffix | Maintainability | FxCop |
| 79. | Enum values with type name should not be prefixed | Maintainability | FxCop |
| 80. | Parameter names should not match member names | Maintainability | FxCop |
| 81. | Get methods should not match property names | Maintainability | FxCop |
| 82. | Type Names Should Not Match Namespaces | Maintainability | FxCop |
| 83 | Base declaration should not match parameter names | Maintainability | FxCop |

Table 1. Rule-set of FxCop with quality parameters: Design, Usage, Maintainability

## 4.2 NDepend:

NDepend static code analysis tool can be incorporated as an add-on to the Visual Studio. Uptil now NDepend is the only tool to keep check on the accumulating debt even within the last hour of

its implementation. With the detailed amount of accumulated debt the user is given the chance to lay it off before integrating it within the whole system by making it addition to the source code. It is of critical importance that NDepend assesses the concrete quality of the source code in comparison with the parameters of well-defined standards.

NDepend offers usage in 2 ways: either it can be integrated intto Visual Studio or it can also be used as a standalone tool. The feasibility of the system lies also on the Csharp LINQ queries that the code makes use of. These can also be custom-developed in a short time. The C# formulae provide the client with the chance to calculate the gathering procedural debt with high accuracy. The default rule-set which is the basis if analysis in this tool, offers the client with a range of rules accumulating to over a hundred.

These rules / guidelines assist to detect potential code structures in source code which contradicts with any of the precise standard. Code deads are also detected with modifications that affect the API (Application Program Interface) or the OOP (Object Oriented Programming) usage. NDepend successfully has approval of around 6000 companies in total, who have confirmed the critical involvement of this tool for improved .NET code. However it is slightly financially burdensome as compared to some other static code analysis tools in practice.

The technique based upon which NDepend conducts analysis on source code is mentioned below. The tool run analysis based upon following rule categories:

- Object Oriented Design
- Design
- Architecture
- Security
- Immutability
- Visibility
- .NET Framework usage
- API breaking changes
- API usage
- Code Coverage
- Code Smells

- Code smells regression
- Dead code
- Source Files Organization
- Naming Conventions

| S. No. | Rules/Metrics | Category | Standard / Tool |
|---|---|---|---|
| 1. | Avoid custom delegates | Design | NDepend |
| 2. | Disposable input field types with should be disposable | Design | NDepend |
| 3. | Finalizer should not be declare by disposable types with unmanaged resources | Design | NDepend |
| 4. | Methods creating disposable objects should not be used if they don't call Dispose() | Design | NDepend |
| 5. | Focus classes that are eligible to be converted into structures | Design | NDepend |
| 6. | Namespaces with few types should be avoided | Design | NDepend |
| 7. | Visibility should be hidden for nested types | Design | NDepend |
| 8. | Types should be declared in namespaces | Design | NDepend |
| 9. | Empty static constructor can be discarded | Design | NDepend |
| 10. | Size shouldn't be too big for instances | Design | NDepend |
| 11. | It is better for attribute classes to be marked sealed | Design | NDepend |
| 12. | Obsolete types, fields and methods should not be used | Design | NDepend |

| | | | |
|---|---|---|---|
| 13. | Methods throwing NotImplementedException should not be implemented | Design | NDepend |
| 14. | Override equals and operator equals on value types | Design | NDepend |
| 15. | Must avoid boxing and unboxing | Usage | NDepend |
| 16. | ISerializable types should be marked with SerializableAttribute | Usage | NDepend |
| 17. | CLSCompliant assemblies should be marked | Usage | NDepend |
| 18. | Attributes with AttributeUsageAttribute should be marked | Usage | NDepend |
| 19. | Calls to GC.Collect() should be removed | Usage | NDepend |
| 20. | GC.WaitForPendingFinalizers() should be called before calling GC.Collect() | Usage | NDepend |
| 21. | Int32 should be used Enum storage | Usage | NDepend |
| 22. | Too general exception types should not be raised | Usage | NDepend |
| 23. | Reserved exception types should not be raised | Usage | NDepend |
| 24. | System.Uri should be the type of Uri fields | Usage | NDepend |
| 25. | ICloneable shouldn't be implemented | Usage | NDepend |
| 26. | Collection properties should not be read only | Usage | NDepend |
| 27. | List.Contains() should be cautioned | Usage | NDepend |

| | | | |
|---|---|---|---|
| 28. | Return collection abstraction should be preferred instead of implementation | Usage | NDepend |
| 29. | Native methods class should be static and internal | Usage | NDepend |
| 30. | Threads shouldn't be created explicitly | Usage | NDepend |
| 31. | Dangerous threading methods should be avoided | Usage | NDepend |
| 32. | TryEnter/Exit both must be called within same method | Usage | NDepend |
| 33. | Both ReaderWriterLock and AcquireLock/ReleaseLock must be called within the same method | Usage | NDepend |
| 34. | Instance fields shouldn't be tagged with ThreadStaticAttribute | Usage | NDepend |
| 35. | Method non-synchronized that read mutable states | Usage | NDepend |
| 36. | Concrete XmlNode shouldn't be returned by methods | Usage | NDepend |
| 37. | System.Xml.XmlDocument shouldn't be extended by types | Usage | NDepend |
| 38. | Float/date parsing be culture aware | Usage | NDepend |
| 39. | Mark Assembles with their assembly version | Usage | NDepend |
| 40. | Assemblies should have the same version | Usage | NDepend |

Table 2. Rule-set of NDepend with quality parameters: Design, Usage, Maintainability

## 4.3 .Net Analyzer:

.Net framework based applications can be smoothly tested using the .Net Analyzers. The potential issues are detected by .Net Analyzer and potential fixes are also displayed. This analyzer covers the following aspects:

- Design Rules
- Usage Rules
- Portability Rules
- Interoperability Rules
- Maintainability Rules
- Naming Rules
- Performance Rules
- Security Rules
- Single-file Rules
- Reliability Rules
- Style Rules
- Documentation Rules
- Globalization Rules

| S. No. | Rules/Metrics | Category | Standard / Tool |
|---|---|---|---|
| 1. | Static members should not be declared on generic types | Design | .Net Analyzer |
| 2. | Types that own disposable fields should be disposable | Design | .Net Analyzer |
| 3. | Do not expose generic lists | Design | .Net Analyzer |
| 4. | Use generic event handler instances | Design | .Net Analyzer |
| 5. | Avoid excessive parameters on generic types | Design | .Net Analyzer |

| | | | |
|---|---|---|---|
| 6. | Enums should have zero value | Design | .Net Analyzer |
| 7. | Collections should implement generic interface | Design | .Net Analyzer |
| 8. | Abstract types should not have constructors | Design | .Net Analyzer |
| 9. | Mark assemblies with CLSCompliantAttribute | Design | .Net Analyzer |
| 10. | Mark assemblies with AssemblyVersionAttribute | Design | .Net Analyzer |
| 11. | Mark assemblies with ComVisibleAttribute | Design | .Net Analyzer |
| 12. | Mark attributes with AttributeUsageAttribute | Design | .Net Analyzer |
| 13. | Define accessors for attribute arguments | Design | .Net Analyzer |
| 14. | Avoid out parameters | Design | .Net Analyzer |
| 15. | Use properties where appropriate | Design | .Net Analyzer |
| 16. | Mark enums with FlagsAttribute | Design | .Net Analyzer |
| 17. | Enum storage should be Int32 | Design | .Net Analyzer |
| 18. | Use events where appropriate | Design | .Net Analyzer |
| 19. | General exception types should not be catched | Design | .Net Analyzer |
| 20. | Implement standard exception constructors | Design | .Net Analyzer |
| 21. | Child types should be able to call Interface methods | Design | .Net Analyzer |
| 22. | Nested types should not be visible | Design | .Net Analyzer |

| | | | |
|---|---|---|---|
| 23. | Override methods on comparable types | Design | .Net Analyzer |
| 24. | Avoid empty interfaces | Design | .Net Analyzer |
| 25. | Provide ObsoleteAttribute message | Design | .Net Analyzer |
| 26. | Indexers should use integral or string arguments | Design | .Net Analyzer |
| 27. | Properties should not be write only | Design | .Net Analyzer |
| 28. | Types should not be passed by reference | Design | .Net Analyzer |
| 29. | Equal operator should not be overloaded on reference types | Design | .Net Analyzer |
| 30. | Do not declare protected members in sealed types | Design | .Net Analyzer |
| 31. | Declare types in namespaces | Design | .Net Analyzer |
| 32. | Visible instance fields shouldn't be declared | Design | .Net Analyzer |
| 33. | Static holder types should be sealed | Design | .Net Analyzer |
| 34. | Constructors shouldn't be used by static holder types | Design | .Net Analyzer |
| 35. | URI parameters should not be strings | Design | .Net Analyzer |
| 36. | URI return values should not be strings | Design | .Net Analyzer |
| 37. | URI properties should not be strings | Design | .Net Analyzer |
| 38. | Certain base types shouldn't be extended by Types | Design | .Net Analyzer |
| 39. | P/Invokes should be moved to NativeMethods class | Design | .Net Analyzer |

| | | | |
|---|---|---|---|
| 40. | Base class methods shouldn't be hidden | Design | .Net Analyzer |
| 41. | Validate arguments of public methods | Design | .Net Analyzer |
| 42. | Implement IDisposable correctly | Design | .Net Analyzer |
| 43. | Exceptions should be public | Design | .Net Analyzer |
| 44. | Do not raise exceptions in unexpected locations | Design | .Net Analyzer |
| 45. | Implement IEquatable when overriding Equals | Design | .Net Analyzer |
| 46. | Override Equals when implementing IEquatable | Design | .Net Analyzer |
| 47 | CancellationToken parameters must come last | Design | .Net Analyzer |
| 48. | Duplicate values shouldn't be used in Enums | Design | .Net Analyzer |
| 49. | Event fields shouldn't be declared virtual | Design | .Net Analyzer |
| 50. | Avoid excessive inheritance | Maintainability | .Net Analyzer |
| 51. | Avoid excessive complexity | Maintainability | .Net Analyzer |
| 52. | Avoid unmaintainable code | Maintainability | .Net Analyzer |
| 53. | Avoid excessive class coupling | Maintainability | .Net Analyzer |
| 54. | Use nameof in place of string | Maintainability | .Net Analyzer |
| 55. | Avoid dead conditional code | Maintainability | .Net Analyzer |
| 56. | Invalid entry in code metrics configuration file | Maintainability | .Net Analyzer |

| | | | |
|---|---|---|---|
| 57. | Review unused parameters | Usage | .Net Analyzer |
| 58. | Call GC.SuppressFinalize correctly | Usage | .Net Analyzer |
| 59. | Rethrow to preserve stack details | Usage | .Net Analyzer |
| 60. | Reserved exception types should not be raised | Usage | .Net Analyzer |
| 61. | Value type static fields should be initialized inline | Usage | .Net Analyzer |
| 62. | Instantiate argument exceptions correctly | Usage | .Net Analyzer |
| 63. | Non-constant fields should not be visible | Usage | .Net Analyzer |
| 64. | Disposable fields should be disposed | Usage | .Net Analyzer |
| 65. | Overridable methods should not be called in constructor | Usage | .Net Analyzer |
| 66. | Base class dispose should be called by dispose methods | Usage | .Net Analyzer |
| 67. | Disposable types should declare finalizer | Usage | .Net Analyzer |
| 68. | Enums should not be marked with with FlagsAttribute | Usage | .Net Analyzer |
| 69. | Override GetHashCode on overriding Equals | Usage | .Net Analyzer |
| 70. | Exceptions should not be raised in exception clauses | Usage | .Net Analyzer |
| 71. | Override equals on overloading operator equals | Usage | .Net Analyzer |
| 72. | Operator overloads have named alternates | Usage | .Net Analyzer |
| 73. | Operators should have symmetrical overloads | Usage | .Net Analyzer |

| | | | |
|---|---|---|---|
| 74. | Collection properties should always be declared read only | Usage | .Net Analyzer |
| 75. | Implement serialization constructors | Usage | .Net Analyzer |
| 76. | Overload operator equals on overriding ValueType.Equals | Usage | .Net Analyzer |
| 77. | Instead of strings, System.Uri objects should be passed | Usage | .Net Analyzer |
| 78. | Mark all non-serializable fields | Usage | .Net Analyzer |
| 79. | Mark ISerializable types with SerializableAttribute | Usage | .Net Analyzer |
| 80. | Correct arguments should be provided to formatting methods | Usage | .Net Analyzer |
| 81. | Test for NaN correctly | Usage | .Net Analyzer |
| 82. | Attribute string literals should parse correctly | Usage | .Net Analyzer |
| 83. | Indexed element initializations shouldn’t be duplicated | Usage | .Net Analyzer |
| 84. | Do not assign a property to itself | Usage | .Net Analyzer |
| 85. | Symbol and its member shouldn’t be assigned in the same statement | Usage | .Net Analyzer |
| 86. | Argument passed to TaskCompletionSource constructor should be TaskCreationOptions enum instead of TaskContinuationOptions enum | Usage | .Net Analyzer |
| 87. | Correct 'enum' argument should be provided to 'Enum.HasFlag' | Usage | .Net Analyzer |
| 88. | String.Contains should be used instead of String.IndexOf | Usage | .Net Analyzer |
| 89. | Use ThrowIfCancellationRequested | Usage | .Net Analyzer |

| | | | |
|---|---|---|---|
| 90. | Use String.Equals over String.Compare | Usage | .Net Analyzer |
| 91. | Opt in to preview features | Usage | .Net Analyzer |
| 92. | Named placeholders should not be numeric values | Usage | .Net Analyzer |
| 93. | Template should be a static expression | Usage | .Net Analyzer |
| 94. | The ModuleInitializer attribute should not be used in libraries | Usage | .Net Analyzer |
| 95. | All members declared in parent interfaces must have an implementation in a DynamicInterfaceCastableImplementation-attributed interface | Usage | .Net Analyzer |
| 96. | Members defined on an interface with 'DynamicInterfaceCastableImplementationAttribute' should be 'static' | Usage | .Net Analyzer |
| 97. | Providing a 'DynamicInterfaceCastableImplementation' interface in Visual Basic is unsupported | Usage | .Net Analyzer |

Table 3. Rule-set of .Net Analyzer with quality parameters: Design, Usage, Maintainability

*Chapter 5*

# 5. RESEARCH RESULTS

## 5.1 Analysis Findings:

After conducting a detailed analysis on the existing standards and the techniques applies for static code analysis by widely known and renowned tools, we've collected a comprehensive rule-set based upon our research criteria i.e. the software quality parameters we've chosen for analysis i.e. Design, Usage, Maintainability. We've finally come-up with a comprehensive list of rules as a standard which would be immensely useful especially for the critical systems which require an extensive analysis of source code before it is sent into production.

## 5.2 Proposed list of Rules:

## Maintainability Rules:

| S. No. | Rules/Metrics | Category | Standard |
|---|---|---|---|
| 1. | Avoid excessive inheritance | Maintainability | .Net Analyzer / C# coding standard by Lance Hunt |
| 2. | Avoid excessive complexity | Maintainability | .Net Analyzer |
| 3. | Avoid unmaintainable code | Maintainability | .Net Analyzer / FxCop |
| 4. | Avoid excessive class coupling | Maintainability | .Net Analyzer |
| 5. | Nameof should be used instead of string | Maintainability | .Net Analyzer |
| 6. | Avoid dead conditional code | Maintainability | .Net Analyzer |
| 7. | Invalid entry in code metrics configuration file | Maintainability | .Net Analyzer |
| 8. | Differentiation between identifiers | Maintainability | FxCop |

| | | | |
|---|---|---|---|
| | should be by more than one case | | |
| 9. | Avoid out parameters | Maintainability | FxCop |
| 10. | Avoid empty interfaces | Maintainability | FxCop |
| 11. | Types should not be passed by reference | Maintainability | FxCop |
| 12. | Avoid excessive inheritance | Maintainability | FxCop |
| 13. | Review misleading field names | Maintainability | FxCop |
| 14. | Avoid excessive class coupling | Maintainability | FxCop |
| 15. | Resource string compound words' casing should be done correctly | Maintainability | FxCop |
| 16. | Compound words should be cased correctly | Maintainability | FxCop |
| 17. | Resource strings should be spelled correctly | Maintainability | FxCop |
| 18. | Identifiers should be spelled correctly | Maintainability | FxCop |
| 19. | Identifiers should not contain underscores | Maintainability | FxCop |
| 20. | Identifiers should be cased correctly | Maintainability | FxCop |
| 21. | Identifiers should have correct suffix | Maintainability | FxCop |
| 22. | Enum values with type name should not be prefixed | Maintainability | FxCop |
| 23. | Parameter names should not match member names | Maintainability | FxCop |

| | | | |
|---|---|---|---|
| 24. | Property names should not match get methods | Maintainability | FxCop |
| 25. | Namespaces should not match type names | Maintainability | FxCop |
| 26. | Base declaration should not match parameter names | Maintainability | FxCop |
| 27. | "abstract" classes should not have "public" constructors | Maintainability | C# coding standard by Lance Hunt |
| 28. | "out" and "ref" parameters should not be used | Maintainability | C# coding standard by Lance Hunt |
| 29. | Attribute constructor must not use unnecessary parenthesis | Maintainability | C# coding standard by Lance Hunt |
| 30. | Files should contain an empty newline at the end | Maintainability | JPL |
| 31. | Collapsible "if" statements should be merged | Maintainability | C# coding standard by Lance Hunt |
| 32. | In source files, per file, one namespace and one class | Maintainability | CERT |
| 33. | Variables: One variable per declaration. | Maintainability | C# coding standard by Lance Hunt |
| 34. | Mark members as static. | Maintainability | C# coding standard by Lance Hunt |
| 35. | Multiple classes should not be added in one class | Maintainability | C# coding standard by Lance Hunt |
| 36. | 'TODO' or 'FIXME' should be resolved | Maintainability | JPL |
| 37. | Methods should not have excessive | Maintainability | JPL |

| | | | |
|---|---|---|---|
| | lines of code | | |
| 38. | More than one Class, Enum (global), or Delegate (global) per file should be avoided. Descriptive file names should be used when having multiple Class, Enum, or Delegates. | Maintainability | C# coding standard by Lance Hunt |
| 39. | No unused variable should be present in any file | Maintainability | C# coding standard by Lance Hunt |
| 40. | Dispose From Dispose | Maintainability | C# coding standard by Lance Hunt |
| 41. | Don't test modulus for equality | Maintainability | CERT |
| 42. | No exceptions used be used in finally block. | Maintainability | CERT |
| 43. | Index Of Check Against Zero. | Maintainability | C# coding standard by Lance Hunt |
| 44. | Parameters Correct Order | Maintainability | C# coding standard by Lance Hunt |
| 45. | Optimize the number of fields in classes | Maintainability | JPL |
| 46. | Avoid creating files that contain many lines of code | Maintainability | JPL |
| 47 | Don't leave nested blocks f code empty | Maintainability | JPL |
| 48. | Remove empty finalizer | Maintainability | JPL |

Table 4. Proposed set of Rules - Maintainability

## Usage Rules:

| S. No. | Rules/Metrics | Category | Standard |
|---|---|---|---|
| 1. | Result of integer multiplication shouldn't be casted to type 'long' | Usage | JPL |
| 2. | Overridable methods should not be called in constructors | Usage | C# coding standard by Lance Hunt, CERT, .Net Analyzer, FxCop |
| 3. | Review unused parameters | Usage | .Net Analyzer |
| 4. | Call GC.SuppressFinalize correctly | Usage | .Net Analyzer |
| 5. | Rethrow to preserve stack details | Usage | .Net Analyzer |
| 6. | Reserved exception types should not be raised | Usage | .Net Analyzer / NDepend |
| 7. | Value type static fields should be initialized inline | Usage | .Net Analyzer, FxCop, .Net Analyzer |
| 8. | Instantiate argument exceptions correctly | Usage | .Net Analyzer |
| 9. | Non-constant fields should not be visible | Usage | .Net Analyzer |
| 10. | Disposable fields should be disposed | Usage | .Net Analyzer |
| 12. | Base class dispose should be called by dispose methods | Usage | .Net Analyzer |
| 13. | Disposable types should declare finalizer | Usage | .Net Analyzer |
| 14. | Enums should not be marked with FlagsAttribute | Usage | .Net Analyzer |

| | | | |
|---|---|---|---|
| 15. | Override GetHashCode on overriding Equals | Usage | .Net Analyzer |
| 16. | Exceptions shouldn't be raised in exception clauses | Usage | .Net Analyzer |
| 17. | Override equals on overloading operator equals | Usage | .Net Analyzer |
| 18. | Operator overloads have named alternates | Usage | .Net Analyzer |
| 19. | Operators should have symmetrical overloads | Usage | .Net Analyzer |
| 20. | Collection properties should be read only | Usage | .Net Analyzer |
| 21. | Implement serialization constructors | Usage | .Net Analyzer |
| 22. | Overload operator equals on overriding ValueType.Equals | Usage | .Net Analyzer |
| 23. | System.Uri objects should be passed instead of strings | Usage | .Net Analyzer |
| 24. | Mark all non-serializable fields | Usage | .Net Analyzer |
| 25. | Mark ISerializable types with SerializableAttribute | Usage | .Net Analyzer |
| 26. | Correct arguments should be provided to formatting methods | Usage | .Net Analyzer |
| 27. | Test for NaN correctly | Usage | .Net Analyzer |
| 28. | Parsing of attribute string literals should be correct | Usage | .Net Analyzer |
| 29. | Indexed element initializations should not be duplicated | Usage | .Net Analyzer |
| 30. | Do not assign a property to itself | Usage | .Net Analyzer |
| 31. | Symbol and its member shouldn't be assigned in the same statement | Usage | .Net Analyzer |

| | | | |
|---|---|---|---|
| 32. | Argument passed to TaskCompletionSource constructor should be TaskCreationOptions enum instead of TaskContinuationOptions enum | Usage | .Net Analyzer |
| 33. | Correct 'enum' argument should be provided to 'Enum.HasFlag' | Usage | .Net Analyzer |
| 34. | String.Contains should be used instead of String.IndexOf | Usage | .Net Analyzer |
| 35. | Use ThrowIfCancellationRequested | Usage | .Net Analyzer |
| 36. | Use String.Equals over String.Com pare | Usage | .Net Analyzer |
| 37. | Opt in to preview features | Usage | .Net Analyzer |
| 38. | Implement IDisposable correctly | Usage | FxCop |
| 39. | Avoid duplicate accelerators | Usage | FxCop |
| 40. | Wrap vulnerable finally clauses in outer try | Usage | FxCop |
| 41. | Default constructors must be at least as critical as base type default constructors | Usage | FxCop |
| 42. | Objects with weak identity should not be locked | Usage | FxCop |
| 43. | Pointers should not be visible | Usage | FxCop |
| 44. | Methods must keep consistent transparency when overriding base methods | Usage | FxCop |
| 45. | Rethrow to preserve stack details | Usage | FxCop |

| | | | |
|---|---|---|---|
| 46. | Finalizers should call base class finalizer | Usage | FxCop |
| 47. | Declare event handlers correctly | Usage | FxCop |
| 48. | Avoid namespaces with few types | Usage | FxCop |
| 49. | Boxing/unboxing should be avoided | Usage | NDepend |
| 50. | ISerializable types should be marked with SerializableAttribute | Usage | NDepend |
| 51. | CLSCompliant assemblies should be marked | Usage | NDepend |
| 52. | Attributes with AttributeUsageAttribute should be marked | Usage | NDepend |
| 53. | Calls to GC.Collect() should be removed | Usage | NDepend |
| 54. | GC.WaitForPendingFinalizers() should be called before calling GC.Collect() | Usage | NDepend |
| 55. | Int32 should be used Enum storage | Usage | NDepend |
| 56. | Too general exception types should not be raised | Usage | NDepend |
| 57. | Reserved exception types should not be raised | Usage | NDepend |
| 58. | System.Uri should be the type of Uri fields | Usage | NDepend |
| 59. | ICloneable shouldn't be implemented | Usage | NDepend |
| 60. | Collection properties should not be read only | Usage | NDepend |
| 61. | List.Contains() should be cautioned | Usage | NDepend |

| | | | |
|---|---|---|---|
| 62. | Return collection abstraction should be preferred instead of implementation | Usage | NDepend |
| 63. | Native methods class should be static and internal | Usage | NDepend |
| 64. | Threads shouldn't be created explicitly | Usage | NDepend |
| 65. | Dangerous threading methods should be avoided | Usage | NDepend |
| 66. | TryEnter/Exit both must be called within same method | Usage | NDepend |
| 67. | Both ReaderWriterLock and AcquireLock/ReleaseLock must be called within the same method | Usage | NDepend |
| 68. | Instance fields shouldn't be tagged with ThreadStaticAttribute | Usage | NDepend |
| 69. | Method non-synchronized that read mutable states | Usage | NDepend |
| 70. | Concrete XmlNode shouldn't be returned by methods | Usage | NDepend |
| 71. | System.Xml.XmlDocument shouldn't be extended by types | Usage | NDepend |
| 72. | Float/date parsing be culture aware | Usage | NDepend |
| 73. | Mark Assembles with their assembly version | Usage | NDepend |
| 74. | Assemblies should have the same version | Usage | NDepend |
| 75. | Property assignments for "readonly" fields shouldn't be done which are not constrained to reference types | Usage | JPL |
| 76. | Hardcoded IP addresses should be avoided | Usage | JPL |

| | | | |
|---|---|---|---|
| 77. | Literals should not be passed as localized parameters | Usage | C# coding standard by Lance Hunt |
| 78. | In-source issue suppression should be avoided. | Usage | JPL |
| 79. | Method's return value should not be ignored | Usage | JPL |
| 80. | Maximum 7 parameters should be used in a method. | Usage | C# coding standard by Lance Hunt |
| 81. | base and this should only be used in constructors or within an override. | Usage | C# coding standard by Lance Hunt |
| 82. | Avoid embedded assignment. | Usage | Misra |
| 83. | Embedded method invocation should be avoided. | Usage | Misra |
| 84. | Always invoke Dispose() & Close()if offered, declare where needed. | Usage | JPL |
| 85. | Include braces for control structures | Usage | JPL |
| 86. | Redundant types should be avoided | Usage | JPL |
| 87. | Access modifiers should always be explicitly declared | Usage | C# coding standard by Lance Hunt |
| 88. | Member variables should be declared private. Properties should be used to provide them access with | Usage | C# coding standard by Lance Hunt |

| | | | |
|---|---|---|---|
| | public, protected, or internal access modifiers. | | |
| 89. | Avoid specifying type for enum - unless you have an explicit need for long instead of default int. | Usage | C# coding standard by Lance Hunt |
| 90. | Hidden string allocations should be avoided within a loop. Use String.Compare() for case-sensitive | Usage | C# coding standard by Lance Hunt, JPL |
| 91. | Use preferably loops or nested loops instead pf recursive methods. | Usage | JPL |
| 92. | Enumerated items within should not be modified within a foreach statement. | Usage | Misra |
| 93. | Avoid assignment within conditional statements. | Usage | C# coding standard by Lance Hunt |
| 94. | Nested if/else is preferred over switch/case for short and complex conditions. | Usage | C# coding standard by Lance Hunt |
| 95. | Never declare an empty catch block. | Usage | JPL |
| 96. | Nesting a try/catch within a catch block should be avoided. | Usage | JPL |
| 97. | Avoid re-throwing an exception.. | Usage | C# coding standard by Lance Hunt |
| 98. | If re-throwing an exception, preserve the original call stack by | Usage | Misra |

| | | | |
|---|---|---|---|
| | omitting the exception argument from the throw statement. | | |
| 99. | Locking a Type should be avoided. Example: lock(typeof(MyClass)); | Usage | C# coding standard by Lance Hunt |
| 100. | Don’t lock the current object instance. Example: lock(this); | Usage | C# coding standard by Lance Hunt |
| 101. | Don’t invoke methods within a conditional expression. | Usage | C# coding standard by Lance Hunt |
| 102. | Initialize variables where declared. | Usage | C# coding standard by Lance Hunt |
| 103. | Avoid calling 'toString' on a string | Usage | JPL |
| 104. | Test Event & Delegate instances for null | Usage | C# coding standard by Lance Hunt |
| 105. | Disposable member in non-disposable class. | Usage | CWE |
| 106. | Loss Of Fraction In Division. | Usage | C# coding standard by Lance Hunt |
| 107. | Empty statements should be avoided | Usage | JPL |
| 108. | 'break' must be included in a 'case' statement | Usage | JPL |
| 109. | Do not compare identical expressions | Usage | JPL |
| 110. | Do not test floating point equality | Usage | JPL |

| | | | |
|---|---|---|---|
| 111. | Methods calling many other methods should be avoided | Usage | JPL |
| 112. | Do not perform self-assignment | Usage | JPL |
| 113. | Non-static nested classes are not preferable unless necessary | Usage | JPL |
| 114. | Loop counter should not be updated within the loop body | Usage | C# coding standard by Lance Hunt |
| 115. | Static constructors should be removed | Usage | C# coding standard by Lance Hunt |
| 116. | Useless object instantiation should either be removed or utlilized | Usage | JPL |
| 117. | Pass the missing user - supplied value to the base | Usage | C# coding standard by Lance Hunt |
| 118. | Avoid overriding only one of 'equals' and 'hashCode | Usage | C# coding standard by Lance Hunt, JPL |
| 119. | Avoid unused fields | Usage | JPL |
| 120. | Ensure that the fields are explicitly initialized | Usage | JPL |

Table 5. Proposed set of Rules - Usage

## Design Rules:

| S. No. | Rules/Metrics | Category | Standard |
|---|---|---|---|
| 1. | Cast concrete type to interface. | Design | C# coding standard by Lance Hunt |
| 2. | class with equality should implement IEquatable | Design | C# coding standard by Lance Hunt |
| 3. | Interfaces should not be empty | Design | C# coding standard by Lance Hunt |
| 4. | Conditional expressions should avoid type mismatch | Design | JPL |
| 5. | Object identity of strings should not be compared | Design | JPL |
| 6. | Don't check if a string is equal to an empty string | Design | C# coding standard by Lance Hunt, JPL |
| 7. | Avoid assignments in Boolean expressions | Design | JPL |
| 8. | Curly braces ({ and }) should be placed on a new line. | Design | C# coding standard by Lance Hunt |
| 9. | Related attribute declarations should be on a single line, else make each attribute be a separate declaration. | Design | C# coding standard by Lance Hunt |
| 10. | Assembly scope attribute declarations should be on a separate line. | Design | C# coding standard by Lance Hunt |
| 11. | Type scope attribute declarations should be on a separate line. | Design | C# coding standard by Lance Hunt |

| | | | |
|---|---|---|---|
| 12. | Method scope attribute declarations should be on a separate line. | Design | C# coding standard by Lance Hunt |
| 13. | Member scope attribute declarations should be on a separate line. | Design | C# coding standard by Lance Hunt |
| 14. | Direct casting should be avoided. Instead use “as”, check null. | Design | C# coding standard by Lance Hunt |
| 15. | String.Format() or StringBuilder should be preferred over string concatenation. | Design | C# coding standard by Lance Hunt |
| 16. | Never concatenate strings inside a loop. | Design | C# coding standard by Lance Hunt |
| 17. | Ternary conditional operator should be used only for trivial conditions. | Design | C# coding standard by Lance Hunt |
| 18. | Compound conditional expressions should be avoided instead boolean variables should be used to split parts into multiple manageable expressions. | Design | C# coding standard by Lance Hunt |
| 19. | Multiple levels of nesting should be avoided in methods | Design | JPL |
| 20. | Declaring of array constants should be avoided | Design | JPL |
| 21. | Comparing arrays by using 'Object.equals' should be avoided | Design | JPL |
| 22. | Avoid assigning to a local variable in a 'return' statement | Design | JPL |
| 23. | 'switch' must include the cases for all 'enum' constants | Design | JPL |

| | | | |
|---|---|---|---|
| 24. | Avoid assigning to parameters in a method or constructor | Design | JPL |
| 25. | Only to hold constants, defining abstract class or interface should be avoided | Design | JPL |
| 26. | Avoid overriding only one of 'equals' and 'hashCode | Design | JPL, C# coding standard by Lance Hunt |
| 27. | Use string or intergal type or refactor index into method | Design | C# coding standard by Lance Hunt |
| 28. | Static members should not be declared on generic types | Design | FxCop, .Net Analyzer |
| 29. | Generic lists should not be exposed | Design | FxCop |
| 30. | Generic event handler instances should be used | Design | FxCop |
| 31. | Generic methods should provide type parameter | Design | FxCop |
| 32. | Avoid excessive parameters on generic types | Design | FxCop |
| 33. | Do not nest generic types in member signatures | Design | FxCop |
| 34. | Use generics where appropriate | Design | FxCop |
| 35. | Enums should have zero value | Design | FxCop |
| 36. | Collections should implement generic interface | Design | FxCop |
| 37. | Base types should be passed as parameters | Design | FxCop |

| | | | |
|---|---|---|---|
| 38. | Abstract types should not have constructors | Design | FxCop |
| 39. | Overload operator equals on overloading add and subtract | Design | FxCop |
| 40. | Indexers should not be multidimensional | Design | FxCop |
| 41. | Params array should be used instead of repetitive arguments | Design | FxCop |
| 42. | Default parameters should not be used | Design | FxCop |
| 43. | Use events where appropriate | Design | FxCop |
| 44. | Do not catch general exception types | Design | FxCop |
| 45. | Implement standard exception constructors | Design | FxCop |
| 46. | Nested types should not be visible | Design | FxCop |
| 47. | Strongly typed members should be used in ICollection implementations | Design | FxCop |
| 48. | Override methods on comparable types | Design | FxCop |
| 49. | Lists are strongly typed | Design | FxCop |
| 50. | Integral / string argument should be used for indexers | Design | FxCop, .Net Analyzer |
| 51. | Properties should not be write only | Design | FxCop |
| 52. | Do not overload operator equals on reference types | Design | FxCop |

| | | | |
|---|---|---|---|
| 53. | Protected members should not be declared in sealed types | Design | FxCop |
| 54. | Virtual members should not be declared in sealed types | Design | FxCop |
| 55. | Assure that Static holder types are sealed | Design | FxCop |
| 56. | Constructors should not be there in static holder types | Design | FxCop |
| 57. | Avoid URI return values being strings | Design | FxCop |
| 58. | Certain base types should not be extended by Types | Design | FxCop, .Net Analyzer |
| 59. | Members should not expose certain concrete types | Design | FxCop |
| 60. | Exceptions should be public | Design | FxCop |
| 61. | Avoid excessive complexity | Design | FxCop |
| 62. | Differentiation between identifiers should be by more than one case | Design | FxCop |
| 63. | Types that own disposable fields should be disposable | Design | FxCop |
| 64. | Mark assemblies with AssemblyVersionAttribute | Design | FxCop |
| 65. | Child types should be able to call Interface methods | Design | FxCop, .Net Analyzer |
| 66. | Types that own native resources should be disposable | Design | FxCop |

| | | | |
|---|---|---|---|
| 67. | Base class methods should not be hidden | Design | FxCop |
| 68. | Exceptions should not be raised in unexpected locations | Design | FxCop |
| 69. | P/Invoke entry points should exist | Design | FxCop |
| 70. | Dispose objects before losing scope | Design | FxCop |
| 71. | Methods with link demands should not be indirectly exposed | Design | FxCop |
| 72. | Override link demands should be identical to base | Design | FxCop |
| 73. | Types must be at least as critical as their base types and interfaces | Design | FxCop |
| 74. | Do not dispose objects multiple times | Design | FxCop |
| 75. | Disposable fields should be disposed | Design | FxCop |
| 76. | Mark all non-serializable fields | Design | FxCop |
| 77. | Disposable fields should be disposed | Design | FxCop |
| 78. | Avoid custom delegates | Design | NDepend |
| 79. | Types with disposable input fields must be disposable | Design | NDepend |
| 80. | Finalizer should be declared by Disposable types with unmanaged resources | Design | NDepend |
| 81. | Dispose() cannot be called by methods creating disposable objects | Design | NDepend |

| | | | |
|---|---|---|---|
| 82. | Classes that are candidate to be turned into structures | Design | NDepend |
| 83. | Namespaces with few types should be avoided | Design | NDepend |
| 84. | Do not make nested types visible | Design | NDepend, .Net Analyzer |
| 85. | Types should be declared in namespaces | Design | NDepend |
| 86. | Discard empty static constructor | Design | NDepend |
| 87. | Keep a check on instance's size | Design | NDepend |
| 88. | Mark the attribute classes as sealed | Design | NDepend |
| 89. | Obsolete types, fields and methods should not be used | Design | NDepend |
| 90. | Methods throwing throw NotImplementedException should be implemented | Design | NDepend |
| 91. | Override equals and operator equals on value types | Design | NDepend |
| 92. | Types that own disposable fields should be disposable | Design | .Net Analyzer |
| 93. | Do not expose generic lists | Design | .Net Analyzer |
| 94. | Use generic event handler instances | Design | .Net Analyzer |
| 95. | Excessive parameters on generic types should be avoided | Design | .Net Analyzer |
| 96. | Enums should have zero value | Design | .Net Analyzer |
| 97. | Collections should implement generic interface | Design | .Net Analyzer |
| 98. | Abstract types should not have constructors | Design | .Net Analyzer |

| | | | |
|---|---|---|---|
| 99. | Mark assemblies with CLSCompliantAttribute | Design | .Net Analyzer |
| 100. | Mark assemblies with AssemblyVersionAttribute | Design | .Net Analyzer |
| 101. | Mark assemblies with ComVisibleAttribute | Design | .Net Analyzer |
| 102. | Mark attributes with AttributeUsageAttribute | Design | .Net Analyzer |
| 103. | Define accessors for attribute arguments | Design | .Net Analyzer |
| 104. | Avoid out parameters | Design | .Net Analyzer |
| 105. | Use properties where appropriate | Design | .Net Analyzer |
| 106. | Mark enums with FlagsAttribute | Design | .Net Analyzer |
| 107. | Enum storage should be Int32 | Design | .Net Analyzer |
| 108. | Use events where appropriate | Design | .Net Analyzer |
| 109. | General exception types should not be catched | Design | .Net Analyzer |
| 110. | Implement standard exception constructors | Design | .Net Analyzer |
| 111. | Override methods on comparable types | Design | .Net Analyzer |
| 112. | Avoid empty interfaces | Design | .Net Analyzer |
| 113. | Provide ObsoleteAttribute message | Design | .Net Analyzer |
| 114. | Properties should not be write only | Design | .Net Analyzer |
| 115. | Types shouldn’t be passed by reference | Design | .Net Analyzer |

| | | | |
|---|---|---|---|
| 116. | Do not overload operator equals on reference types | Design | .Net Analyzer |
| 117. | Protected members should not be declared in sealed types | Design | .Net Analyzer |
| 118. | Declare types in namespaces | Design | .Net Analyzer |
| 119. | Do not declare visible instance fields | Design | .Net Analyzer |
| 120. | Mark static holder types as sealed | Design | .Net Analyzer |
| 121. | There shouldn't be constructors in static holder types | Design | .Net Analyzer |
| 122. | URI params shouldn't be in the form of strings | Design | .Net Analyzer |
| 123. | URI return values shouldn't be in the form of strings | Design | .Net Analyzer |
| 124. | URI properties shouldn't be of type strings | Design | .Net Analyzer |
| 125. | P/Invokes should be moved to NativeMethods class | Design | .Net Analyzer |
| 126. | Base class methods should not be hidden | Design | .Net Analyzer |
| 127. | Validate arguments of public methods | Design | .Net Analyzer |
| 128. | Implement IDisposable correctly | Design | .Net Analyzer |
| 129. | Exceptions should be public | Design | .Net Analyzer |
| 130. | Do not raise exceptions in unexpected locations | Design | .Net Analyzer |
| 131. | Implement IEquatable when overriding Equals | Design | .Net Analyzer |
| 132. | Override Equals when implementing IEquatable | Design | .Net Analyzer |

| | | | |
|---|---|---|---|
| 133. | CancellationToken parameters must come last | Design | .Net Analyzer |
| 134. | Duplicate values shouldn’t be present in Enums | Design | .Net Analyzer |
| 135. | Event fields should not be declared as virtual | Design | .Net Analyzer |

Table 6. Proposed set of Rules - Design

*Chapter 6*

# 6. RESULTS AND TECHNICAL DISCUSSION

We have finally proposed a rule-set which ensures that all quality factors pertaining to the 3 factors we've chosen for research purpose i.e. Design, Usage, Maintainability are completely researched. We've come up with a comprehensive rule-set of 303 rules. We suggest to keep the proposed rule-set as a basis for conducting research on software systems especially critical systems in order to make sure that no rule is missed out.

We have proposed a comprehensive Rule-set for static code analysis of critical systems based upon 3 quality parameters i.e. Design, Usage, Maintenance. The proposed rule-set is of significant importance for the software industry, especially for the domain experts of critical systems. The idea behind proposing the researched study for critical systems is that critical systems are of significant importance and in the worst case, the failure of such systems cause huge sum of loss either it be a financial loss or loss of lives. Therefore, while conducting testing of critical systems, we need to take into account very minute details as well, because we never know, even an ignorable aspect can also become a loophole in the system, leading to such scenarios which may involve a chain of executions pushing the entire system towards failure, as also discussed in section 1.3.

As software quality is a vast term, it is an ideal case for a software system to fulfill and cover all aspects of software quality which is nearly impossible, therefore in order to get closer to the ideal case of software quality, we selected 3 quality parameters of: Maintainability, Usage, Design. Then, we studied popular and widely known standards for assessing code quality, and researched the rule-set / foundation based upon which top-notch static code analysis tools conduct testing. And we found that the rule-sets varied alot. Therefore after conducting an in-depth analysis and research we proposed a cumulative rule-set of 303 rules which we've split into 3 parts i.e. underlying software quality parameters for research.

*Chapter 7*

# 7. PROTOTYPE OF STATIC CODE ANALYZER REFLECTING IMPLEMENTATION OF A SMALL SET OF RULES FROM OUR PROPOSED RULE SET

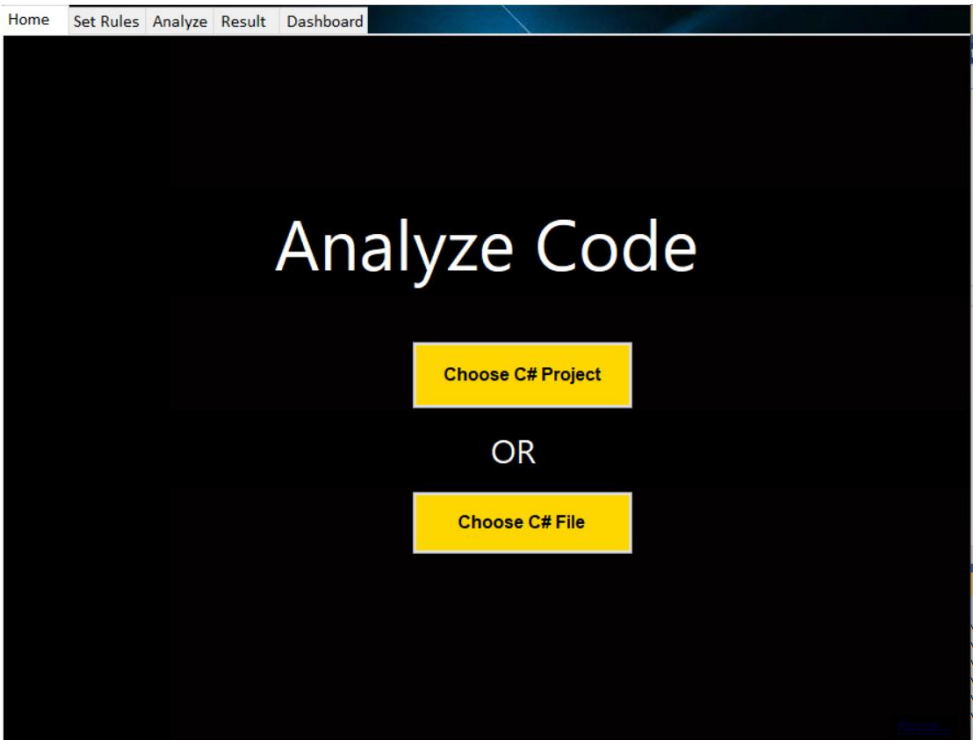



Fig 1. Main Layout

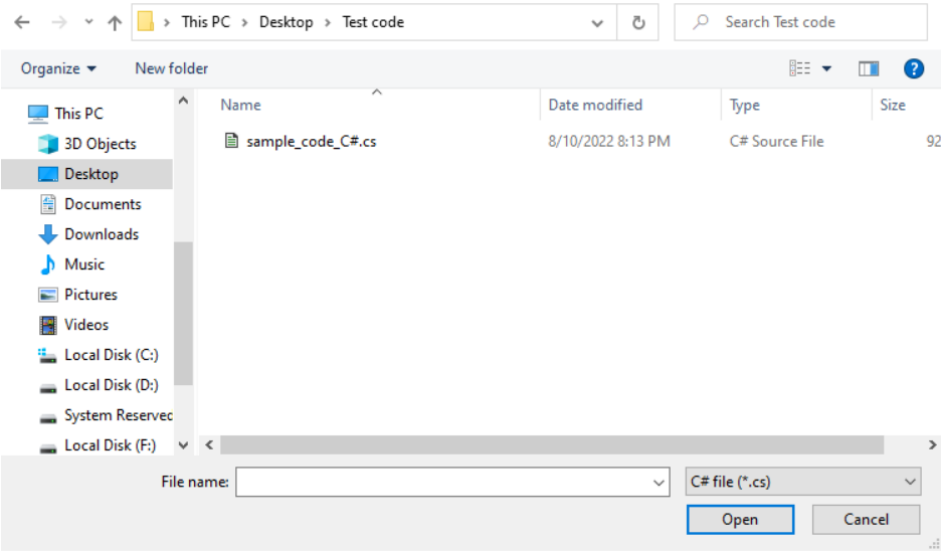



Fig 2. File Selection

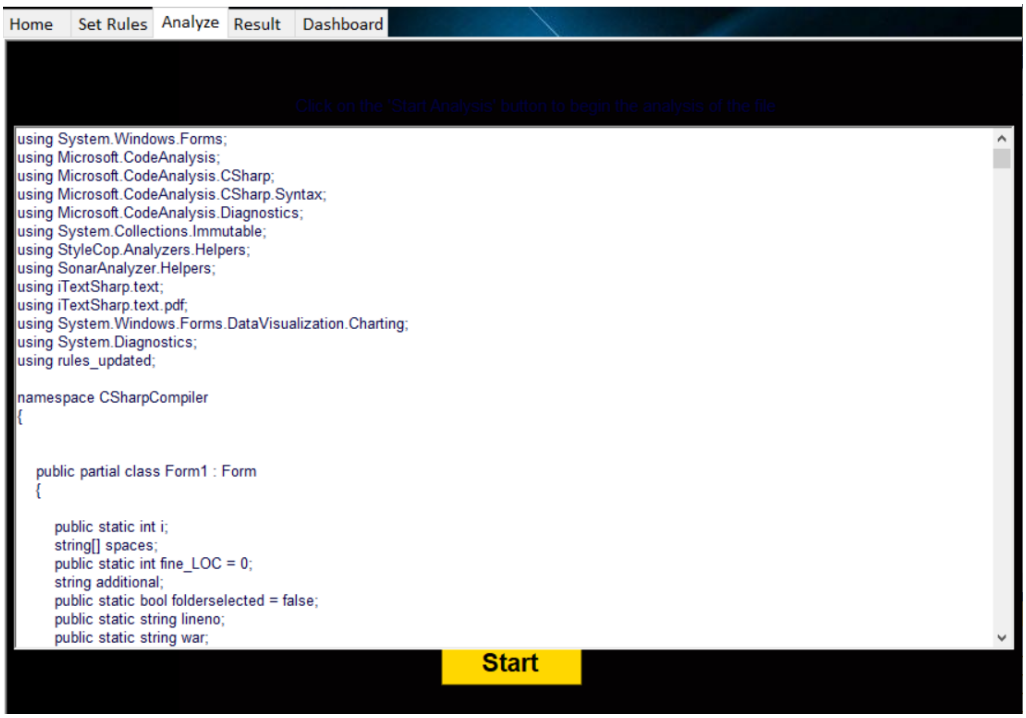



Fig 3. File loaded for analysis

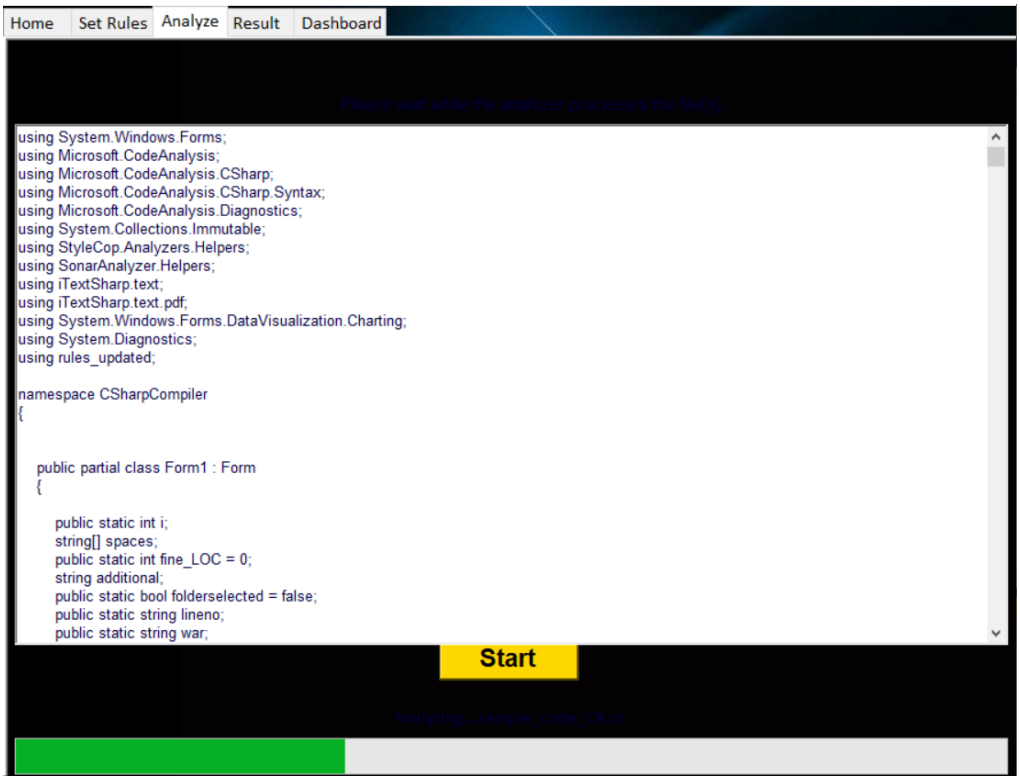
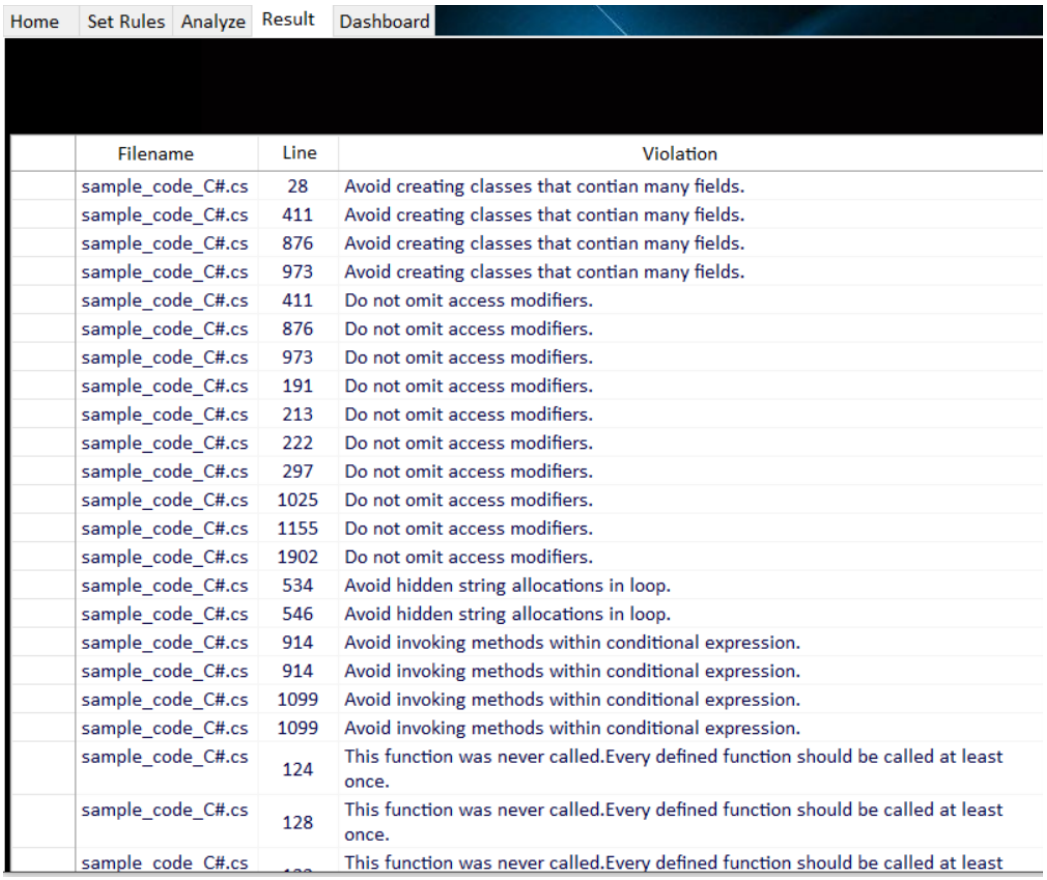



Fig 4. Analyzing code




| | Filename | Line | Violation |
|---|---|---|---|
| | sample_code_C#.cs | 28 | Avoid creating classes that contian many fields. |
| | sample_code_C#.cs | 411 | Avoid creating classes that contian many fields. |
| | sample_code_C#.cs | 876 | Avoid creating classes that contian many fields. |
| | sample_code_C#.cs | 973 | Avoid creating classes that contian many fields. |
| | sample_code_C#.cs | 411 | Do not omit access modifiers. |
| | sample_code_C#.cs | 876 | Do not omit access modifiers. |
| | sample_code_C#.cs | 973 | Do not omit access modifiers. |
| | sample_code_C#.cs | 191 | Do not omit access modifiers. |
| | sample_code_C#.cs | 213 | Do not omit access modifiers. |
| | sample_code_C#.cs | 222 | Do not omit access modifiers. |
| | sample_code_C#.cs | 297 | Do not omit access modifiers. |
| | sample_code_C#.cs | 1025 | Do not omit access modifiers. |
| | sample_code_C#.cs | 1155 | Do not omit access modifiers. |
| | sample_code_C#.cs | 1902 | Do not omit access modifiers. |
| | sample_code_C#.cs | 534 | Avoid hidden string allocations in loop. |
| | sample_code_C#.cs | 546 | Avoid hidden string allocations in loop. |
| | sample_code_C#.cs | 914 | Avoid invoking methods within conditional expression. |
| | sample_code_C#.cs | 914 | Avoid invoking methods within conditional expression. |
| | sample_code_C#.cs | 1099 | Avoid invoking methods within conditional expression. |
| | sample_code_C#.cs | 1099 | Avoid invoking methods within conditional expression. |
| | sample_code_C#.cs | 124 | This function was never called.Every defined function should be called at least once. |
| | sample_code_C#.cs | 128 | This function was never called.Every defined function should be called at least once. |
| | sample_code_C#.cs | | This function was never called.Every defined function should be called at least |

Fig 5. Analysis Result

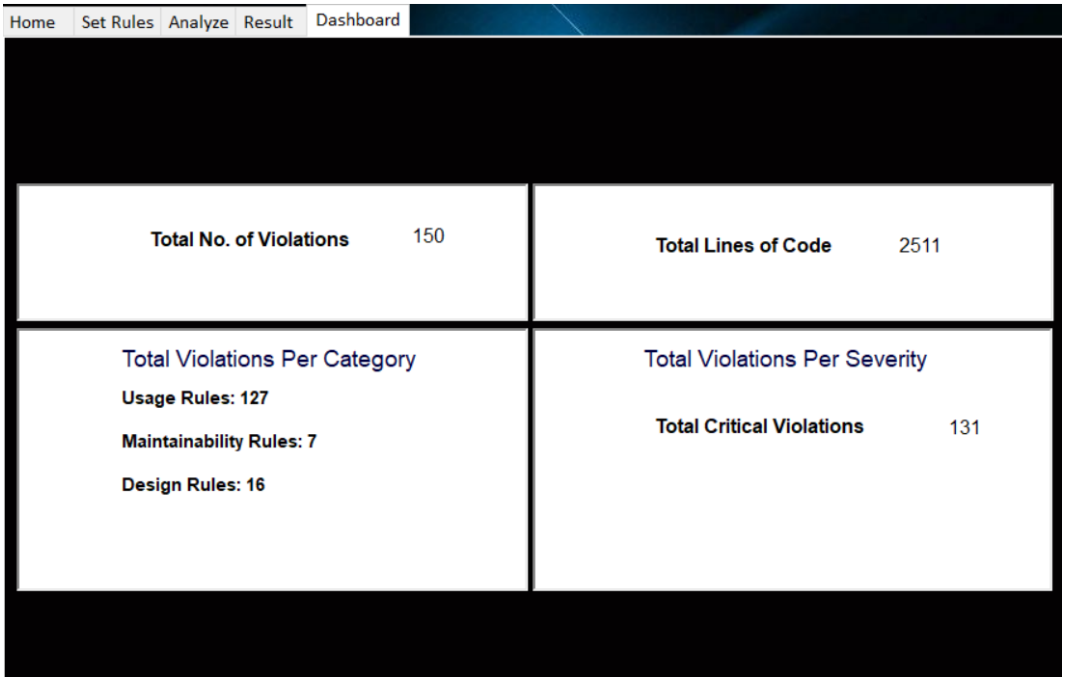



Fig 6. Analysis Summary

*Chapter 8*

# 8. Conclusion and Future work

We have finally devised a rule-set / standard for static code analysis of critical systems. We've researched multiple coding standards, studied and sorted out the basis upon based upon which, world-renowned tools conduct static code analysis. Finally we devised a rule of almost 303 rules which contains strategies to figure out potential defects and shortcomings in the source code pertaining to all aspects of the chosen software quality parameters i.e. Design, Usage, Maintainability. The project can be extended in multiple dimensions. The domain of static code analysis is wide, the proposed research and can be extended in multiple manners as follows:

- Rule-set can be extended by designing and adding more rules for analysis
- Severity of violations could be specified as: Minor, Major, Critical
- The rule-set can be extended to accommodate more dimensions of software design and development, likewise integrate best practices for each.
- Rule-set can be extended and can be categorized into multiple categories such as:
    - Code Smells
    - Code smells regression
    - Object Oriented Design
    - Design
    - Architecture
    - API breaking changes
    - Code Coverage
    - Dead code
    - Security
    - Visibility
    - Immutability
    - Naming Conventions
    - Source Files Organization
    - .NET Framework usage
    - API usage

*Chapter 9*

# 9. REFERENCES


1. Stefanović, Darko & Nikolić, Danilo & Dakic, Dusanka & Spasojević, Ivana & Ristic, Sonja. (2020). Static Code Analysis Tools: A Systematic Literature Review. 10.2507/31st.daaam.proceedings.078.
2. Ashfaq, Qirat & Khan, Rimsha & Farooq, Sehrish. (2019). A Comparative Analysis of Static Code Analysis Tools that check Java Code Adherence to Java Coding Standards. 98-103. 10.1109/C-CODE.2019.8681007.
3. Kaur, A. and Nayyar, R., 2022. *A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code*.
4. Stefanović, Darko & Nikolić, Danilo & Havzi, Sara & Lolić, Teodora & Dakic, Dusanka. (2021). Identification of strategies over tools for static code analysis. IOP Conference Series: Materials Science and Engineering. 1163. 012012. 10.1088/1757-899X/1163/1/012012.
5. Saha, A. and Prasad, R., 2022. [online] Ijarcce.com. Available at: https://ijarcce.com/wp-content/uploads/2022/07/IJARCCE.2022.11764.pdf
6. Lenarduzzi, Valentina & Sillitti, Alberto & Taibi, Davide. (2018). A Survey on Code Analysis Tools for Software Maintenance Prediction. 10.1007/978-3-030-14687-0_15.
7. Rawat, Sanjay & Saxena, Ashutosh. (2009). Application Security Code Analysis: A Step towards Software Assurance. International Journal of Information and Computer Security (IJICS). 3. 86-110. 10.1504/IJICS.2009.026622.
8. Docs.microsoft.com. 2022. *C# Coding Conventions*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>
9. Hunt, L., n.d. *What Is Static Analysis? Static Code Analysis Overview | Perforce Software*. [online] Perforce Software. Available at: <https://www.perforce.com/blog/sca/what-static-analysis>
10. Docs.microsoft.com. 2022. *The .NET Compiler Platform SDK (Roslyn APIs)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/> [Accessed 25 August 2022].

11. NDepend. n.d. *Improve your .NET code quality with NDepend*. [online] Available at: <https://www.ndepend.com/>
12. Ndepend.com. n.d. *NDepend Rules Explorer*. [online] Available at: <https://www.ndepend.com/default-rules/NDepend-Rules-Explorer.html?ruleid=ND1600#!>
13. Docs.microsoft.com. 2022. *Code analysis rule categories - .NET*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/categories?source=recommendations>
14. Docs.microsoft.com. 2022. *Design rules (code analysis) - .NET*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/design-warnings>
15. Docs.microsoft.com. 2022. *Maintainability rules (code analysis) - .NET*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/maintainability-warnings>
16. Docs.microsoft.com. 2022. *Usage rules (code analysis) - .NET*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/usage-warnings>
17. A. Mandal, D. Mohan, R. Jetley, S. Nair and . M. D'Souza, "A Generic Static Analysis Framework for Domainspecific Languages," in IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2018.
18. G. Chatzieleftheriou, A. Chatzopoulos and P. Katsaros, "Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities," in LECTURE NOTES IN COMPUTER SCIENCE. (8803), Heidelberg, 2011.
19. A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut and D. Lucanu, "A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code," in 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2017.
20. "Domain specific infrastructure for code smell detection in large-scale software systems," in International Research Symposium on Engineering Advancements 2016 (IRSEA 2016), 2016.

21. Z. Fiorella, S. Simone, O. Rocco, C. Gerardo and D. P. Massimiliano, "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines," in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017.
22. K. Tuma, G. Calikli and R. Scandariato, "Threat analysis of software systems: A systematic literature review," Journal of Systems and Software, vol. 144, pp. 275-294, 2018. [16] A. Kaur and R. Nayyar, "A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code," Procedia Computer Science, vol. 171, no. 2019, pp. 2023-2029, 2020.
23. D. Marcilio, C. Furia, R. Bonifacio and P. Gustavo, "SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings," The Journal of Systems & Software, vol. 168, 2020.
24. P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia and M. Vieira, "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios," Computing, vol. 101, no. 2, pp. 161-185, 2019.
25. I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota and J. M. Sánchez-Jara, "Applying static code analysis for domain-specific languages," Software & Systems Modeling, vol. 19, no. 1, pp. 95-110, 2020.
26. M. Beller, R. Bholanath and M. S., "Analyzing the state of static analysis: A large-scale evaluation in open source software," 2016.
27. Flawfinder [online] available:" https://www.dwheeler.com/flawfinder/", 2019
28. RATS [online] available: "https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml,", 2019
29. CPPCHECK [online] available: ": http://cppcheck.sourceforge.net/, 2019 [14] SPOTBUGS [online] available: "https://spotbugs.readthedocs.io/en/stable/", 2019
30. PMD [online] available: "https://pmd.github.io/latest/index.html", 2019
31. R.Mahmood, and Q.H. Mahmoud, "Evaluation of static analysis tools for finding vulnerabilities in JAVA nad C/C++ source code", arXiv e-prints, abs/1805.09040,August 2018
32. R.Amankwah, and P.K.Kudjo, "Evaluation of software vulnerability detection methods and tools: A Review", vol 169, issue 8, pp. 22-27, July 2017
33. A. Moller and I. Schwartzbach, "Static program analysis," 2019.

34. I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota and J. M. Sánchez-Jara, "Applying static code analysis for domain-specific languages," Software & Systems Modeling, vol. 19, no. 1, pp. 95-110, 2020.
35. D. Marcilio, C. Furia, R. Bonifacio and P. Gustavo, "SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings," The Journal of Systems & Software, vol. 168, 2020.
36. P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia and M. Vieira, "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios," Computing, vol. 101, no. 2, pp. 161-185, 2019.
37. N. Imtiaz, B. Murphy and L. Williams, "How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage," 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 2019, pp. 323-333, doi: 10.1109/ISSRE.2019.00040.
38. A. Janes, V. Lenarduzzi, and A. C. Stan. “A continuous software quality monitoring approach for small & medium enterprises” 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion) 2017
39. V. Lenarduzzi, A. C. Stan, D. Taibi, D. Tosi and G. Venters “A dynamical quality model to continuously monitor software maintenance.” 11th European Conference on Information Systems Management (ECISM2017), 2017
40. Owasp.org. n.d. *Source Code Analysis Tools | OWASP Foundation*. [online] Available at: <https://owasp.org/www-community/Source_Code_Analysis_Tools>