

Ministry of Education and Science of Ukraine  
National University of "Kyiv-Mohyla Academy"

Informatics Department Informatics Faculty



## **Statical and Dynamical Software Analysis**

**Course work  
in specialty "Software Engineering"**

Coursework supervisor  
Associate Professor, Doctor of Technical Science  
Hlybovets A.M.

\_\_\_\_\_  
(signature)

“       ” \_\_\_\_\_ 2020

Made by student  
Sosnytskyi S. O.

“       ” \_\_\_\_\_ 2020

Kyiv 2020

Ministry of Education and Science of Ukraine  
National University of "Kyiv-Mohyla Academy"

Department of Computer Science of the Faculty of Informatics

APPROVED

Coursework supervisor

Head of the Department of Computer Science,  
Associate Professor of Computer Science

\_\_\_\_\_  
(signature) Gorokhovsky S.S.

“ \_\_\_\_ ” \_\_\_\_\_ 2020

INDIVIDUAL TASK  
for course work

For the student of the Software Engineering Faculty,  
Masters Degree first year  
THEME Statical and Dynamical Software Analysis

Output data:

Text part content of coursework:

An individual task

Calendar plan

Annotation

An introduction

Part 1: Software Analysis Overview

Part 2: Projects Analysis

Summary

References

Issue date “ \_\_\_\_ ” \_\_\_\_\_ 2020 Supervisor \_\_\_\_\_  
(signature)

Task received \_\_\_\_\_  
(signature)

**Theme:** Statical and Dynamical Software Analysis

**Calendar plan of coursework execution:**

№	Stage name	Deadline	Note
1.	Gettting of coursework topic	01.09.2019	
2.	Searching of appropriate literature	10.09.2019	
3.	Researching in software analysis area	10.12.2019	
4.	Exploring software analysis techniques	25.12.2019	
5.	Researching statical software analysis	01.01.2020	
6.	Researching dynamical software analysis	30.01.2020	
7.	Specification definition	05.02.2020	
8.	Documentation forming	10.02.2020	
9.	Tests implementation	10.04.2020	
10.	Writing architectures overview part	23.04.2020	
11.	Coursework analysis with the supervisor	24.04.2020	
12.	Writing second part of the coursework	30.04.2020	
13.	Coursework analysis with the supervisor	01.05.2020	
14.	Writing coursework summary	01.05.2020	
15.	Coursework analysis with the supervisor	02.05.2020	
16.	Coursework changing according to the supervisor's remarks	03.05.2020	
17.	Creating of the presentation	05.05.2020	
18.	Defending of the coursework		

Student Sosnytskyi S. O.

Supervisor Hlybovets A.M.

“ ”

---

## Content

<b>Annotation</b>	<b>5</b>
Introduction	<b>6</b>
<b>Software Analysis Overview</b>	<b>7</b>
Software Analysis Challenges	7
Static Analysis Concepts	9
Areas to Analyze	9
Families of Program Analysis Techniques	10
Testing: Checking a Set of Finite Executions	10
Assisted Proof: Relying on User-Supplied Invariants	10
Model Checking: Exhaustive Exploration of Finite Systems	11
Conservative Static Analysis: Automatic, Robust, and Incomplete Approach	11
Bug Finding: Error Search, Automatic, Unsound, Incomplete, Based on Heuristics	12
Static Analysis Overview	13
Dynamic Software Analysis Overview	15
Software Metrics	16
The SQALE Model	17
SQALE Model Indexes	17
<b>Tools overview</b>	<b>18</b>
IBM Rational Software Architect	18
iPlasma	19
SonarQube as a platform for continuous analysis	20
<b>Projects Analysis</b>	<b>20</b>
Project overview	20
Structure visualization	21
Package diagram	21
Class diagram	21
Project metrics	24
<b>Unit tests</b>	<b>32</b>
<b>Continuous Code Analysis with the SonarQube</b>	<b>36</b>
<b>Three Lines of Analysis</b>	<b>36</b>
<b>Conclusions</b>	<b>38</b>
<b>Sources</b>	<b>39</b>

### **Annotation**

Development of software with build in quality has become an important trend and a natural choice in many organisations. Currently, methods of measurement and assessment of software quality, security, trustworthiness cannot guarantee safe and reliable operations of software systems completely and effectively.

In this coursework overviewed statistical and dynamical software analysis methods, main concepts and techniques families.

The second part of the coursework is dedicated software analysis including solutions for continuous analysis in agile development life cycle.

**Key words:** Statical Analysis, Dynamical Analysis, Testing, SQALE Mode, IBM Rational Software Architect, Continuous Code Analysis, SonarQube

## Introduction

In each engineering discipline, the main question is whether the design work will actually be done as planned. This question is relevant when we develop mechanical machines, electrical circuits or chemical processes. The answer comes from the analysis of structures, using knowledge of the nature that will perform the projects. For example, when we design a bridge, we analyze how nature manages the project, namely how different forces (such as gravity, wind, and vibration) are applied to the bridge and whether the bridge structure is strong enough to withstand them. The same question applies to computer software. We want to make sure our software works as intended. The intention to analyze varies widely. For overall reliability, we want to ensure that the software does not fail in the event of an unexpected termination. If the software interacts with the outside world, we want to make sure that it will not be tricked into compromising the security of the host computer. Regarding specific functions, we want to check whether the software realizes its functional purpose. If the software is designed to drive cars, we want to make sure that it does not lead to an accident.

However, there is one difference between software analysis and analysis of other types of engineering structures: for computer software, it is not nature that runs, but the computer itself. The computer executes the software according to the language values of the source software. The behavior of the software is determined solely by the values of the language of the software source. A computer is simply an illegible tool that blindly executes software exactly as it is written. Any execution behavior that deviates from our intent is that the software is misspelled in this way. So, for computer software, to answer the question of whether our design will work as we intended, we need knowledge with which we can somehow analyze the meaning of the language of the software source. Such knowledge corresponds to the knowledge accumulated by the natural sciences of nature. We need the knowledge that computer science has accumulated about processing the values of software source languages. There is a formal definition of software behavior, which is determined by the meanings, semantics of the language of its source.

## 1. Software Analysis Overview

The semantics of a program is a description of its execution behavior. A semantic property is any property in relation to the behavior (semantics) of a program during program execution. Therefore, checking whether the software works as we intended is equivalent to checking whether the software satisfies the semantic property of interest. Next, we call the method of verifying that the program satisfies the semantic analysis of the program property, and we call the implementation of program analysis as a tool for analyzing the program. Figure 1. illustrates the correspondence between software analysis and design analysis of other engineering disciplines [17].

Name	Computing area	Other engineering areas
Object	Software	Machine/building/circuit/chemical process design
Execution	Computer	Nature
Question	Work as intended	Work as intended
Knowledge	Program analysis	Thermodynamic equations, and other principles

Figure 1. Program analysis versus other engineering areas

### 1.1 Software Analysis Challenges

Program analysis can be applied wherever understanding program semantics is important or beneficial. Software developers can use program analysis for quality assurance, to locate errors of any kind in their software. Software maintainers can use program analysis to understand legacy software that they maintain. System security gatekeepers can use program analysis to proactively screen out programs whose semantics can be malicious. Software that handles programs as data can use program analysis for the programs' performance improvement too. Language processors such as translators or compilers need program analysis to translate the input programs into optimized ones. Interpreters, virtual machines, and query processors need program analysis for optimized execution of the input programs. Automatic program synthesizers can use program analysis to check and tune what they synthesize [18].

The use of program analysis is not limited to professional software or its developers. As programming becomes a way of living in a highly connected digitized environment, citizen programmers can benefit from program analysis, too, to sanity-check their daily program snippets. Once the object's source language has semantics, program analysis can circumscribe its semantics to provide useful information. For example, program analysis of high-level system configuration scripts can provide information about any existing conflicting requests.

Though the benefits of program analysis are obvious, building a cost-effective program analysis is not trivial, since computer programs are complex and often very large. For example, the number of lines of smartphone applications frequently reaches over half a million, not to mention larger software such as web browsers or operating systems, whose source sizes are over ten million lines. With semantics, the situation is much worse because a program execution is highly dynamic. Programs usually need to react to inputs from external, uncontrolled environments. The number of inputs, not to mention the number of program states, that can arise in all possible use cases is so huge that software developers are likely to fail to handle some corner cases. Given that software is in charge of almost all infrastructures in our personal, social, and global life, the need for cost-effective program analysis technology is greater than ever before. We have already experienced a sequence of appalling accidents whose causes are identified as mistakes in software. Such accidents have occurred in almost all sectors, including space, medical, military, electric power transmission, telecommunication, security, transportation, business, and administration. The long list includes accidents, the large-scale Twitter outage (2016), the fMRI software error (2016) that invalidated fifteen years of brain research, the Heartbleed bug (2014) in the popular OpenSSL cryptographic library that allows attackers to read the memory of any server that uses certain instances of OpenSSL, the stack overflow issues that can explain the Toyota sudden unintended acceleration (2004–2014), the Northeast blackout (2003), the explosion of the Ariane 5 Flight 501 (1996), which took ten years and \$7 billion to build and it is just a few of the more prominent software accidents [17].

Though building error-free software may be unlikely and unconvincing, at least within reasonable costs for large-scale software, cost-effective ways to reduce as many errors as possible are always in high demand. Static analysis, which is the focus of this book, is one kind of program analysis.



## 1.2 Static Analysis Concepts

### 1.2.1 Areas to Analyze

The first question to answer to characterize program analysis techniques is what programs they analyze in order to determine what properties.

Target Program - An obvious characterization of the target programs to analyze is the programming languages in which the programs are written [19].

- **Domain-specific analyses** - certain analyses are aimed at specific families of programs. This specialization is a pragmatic way to achieve a cost-effective program analysis, because each family has a particular set of characteristics on which a program analysis can focus. For example, consider the C programming language. Though the language is widely used to write software, including operating systems, embedded controllers, and all sorts of utilities, each family of programs has a special character.
- **Non-domain-specific analyses** - some analyses are designed without focus on a particular family of programs of the target language. Such analyses are usually those incorporated inside compilers, interpreters, or general-purpose programming environments. Such analyses collect information about the input program to help compilers, interpreters, or programmers for an optimized or safe execution of the program. Non-domain-specific analyses risk being less precise and cost-effective than domain-specific ones in order to have an overall acceptable performance for a wide range of programs.

Besides the language and family of programs to consider, the way input programs are handled may also vary and affects how the analysis works. An obvious option is to handle source programs directly just like a compiler would, but some analyses may input different descriptions of programs instead. There are available two classes of techniques:

- **Program-level** analyses are run on the source code of programs (e.g., written in C or Java) or on executable program binaries and typically involve a front end similar to a compiler's that constructs the syntax trees of programs from the program source or compiled files.
- **Model-level** analyses consider a different input language that aims at modeling the semantics of programs; then the analyses input not a program in a language such as C or Java but a description that models the program to analyze. Such models either need to be constructed manually or are computed by a separate

tool. In both cases, the construction of the model may hide either difficulties or sources of inaccuracy that need to be taken precisely into account.

### **1.3 Families of Program Analysis Techniques**

#### **1.3.1 Testing: Checking a Set of Finite Executions**

When trying to understand how a system behaves, often the first idea that comes to mind is to observe the executions of this system. In the case of a program that may not terminate and may have infinitely many executions, it is of course not feasible to fully observe all executions. Therefore, the testing approach observes only a finite set of finite program executions. This technique is used by all developers, from beginners to large teams designing complex computer systems. In industry, many levels of testing are performed at all stages of development, such as unit testing and integration testing. Basic testing approaches, such as random testing [15], typically provide a low coverage of the tested code. However, more advanced techniques improve coverage. As an example, concolic testing [16] combines testing with symbolic execution (computation of exact relations between input and output variables on a single control flow path) so as to improve coverage and accuracy.

Testing has the following characteristics:

- Easy to automate
- Not robust
- Complete since a failed testing run will produce an execution that is incorrect

#### **1.3.2 Assisted Proof: Relying on User-Supplied Invariants**

This is essentially the approach followed by machine-assisted techniques. This means that users may be required to supply additional information together with the program to analyze. In most cases, the information that needs to be supplied consists of loop invariants and possibly some other intermediate invariants. This often requires some level of expertise. On the other hand, a large part of the verification can generally still be carried out in a fully automatic way. We can cite several kinds of program analyses based on machine-assisted techniques. A first approach is based on theorem-proving tools like Coq [18]. This approach is adapted to the proof of sophisticated program properties. It was applied to the verified CompCert compiler from C to Power-PC assembly.

Machine-assisted techniques have the following characteristics:

- They are not fully automatic and often require the most tedious logical arguments to come from the human user.
- In practice, they are sound with respect to the model of the program semantics used for the proof, and they are also complete up to the abilities of the proof assistant to verify proofs.

In practice, the main limitation of machine-assisted techniques is the significant resources they require, in terms of time and expertise.

### 1.3.3 Model Checking: Exhaustive Exploration of Finite Systems

Another approach focuses on finite systems, that is, systems whose behaviors can be exhaustively enumerated, so as to determine whether all executions satisfy the property of interest. This approach is called finite-state model checking [19] since it will check a model of a program using some kind of exhaustive enumeration. In practice, model checking tools use efficient data structures to represent program behaviors and avoid enumerating all executions thanks to strategies that reduce the search space. This solution is very different from the testing approach discussed in section 1.3.1. Indeed, testing samples a finite set of behaviors among a generally infinite set, whereas model checking attempts to check all executions of a finite system. The finite model-checking approach has been used both in hardware verification and in software verification.

Model checking has the following characteristics:

- It is automatic
- It is robust and complete with respect to the model

An important caveat is that the verification is performed at the model level and not at the program level. As a first consequence, this means that a model of the program needs to be constructed, either manually or by some automatic means. In practice, most model checking tools provide a front end for that purpose. A second consequence is that the relation between this model and the input program should be taken into account when assessing the results. If the model cannot capture exactly the behaviors of the program, the checking of the synthesized model may be either incomplete or not robust, with respect to the input program.

In practice, model-checking tools are often conservative and are thus robust and incomplete with respect to the input program. A large number of model-checking tools have been developed for verifying different kinds of logical

assertions on various models or programming languages.

### **1.3.4 Conservative Static Analysis: Automatic, Robust, and Incomplete Approach**

Instead of constructing a finite model of programs, static analysis relies on other techniques to compute conservative descriptions of program behaviors using finite resources. The core idea is to finitely over-approximate the set of all program behaviors using a specific set of properties, the computation of which can be automated [20]. An example is the type inference present in many modern programming languages such as variants of ML. Types provide a coarse view of what a function does but do so in a very effective manner, since the correctness of type systems guarantees that a function of type  $\text{int} \rightarrow \text{bool}$  will always input an integer and return a Boolean.

Static analysis approaches have the following characteristics:

- It is automatic;
- It produces robust results, as they compute a conservative description of program behaviors, using a limited set of logical properties.
- It is generally incomplete because they cannot represent all program properties and rely on algorithms that enforce termination of the analysis even when the input program may have infinite executions.

While a static analysis is incomplete in general, it is often possible to design a robust static analysis that gives the best possible answer on classes of interesting input programs. However, it is then always possible to craft a correct input program for which the analysis will fail to return a conclusive result.

### **1.3.5 Bug Finding: Error Search, Automatic, Unsound, Incomplete, Based on Heuristics**

Some automatic program analysis tools sacrifice not only completeness but also robustness. The main motivation to do so is to simplify the design and implementation of analysis tools and to provide lighter-weight verification algorithms. The techniques used in such tools are often similar to those used in model checking or static analysis, but they relax the robustness objective. For instance, they may construct not robust finite models of programs so as to quickly enumerate a subset of the executions of the analyzed program, such as by considering

only what happens in the first iteration of each loop, whereas a robust tool would have to consider possibly unbounded iteration numbers. As an example, the tool CODESONAR [21] relies on such approaches so as to search for defects in C/C++ or Assembly programs. It is thus a case that a model checker gives up on robustness in order to produce fewer alarms. Since the main motivation of this approach is to discover bugs (and not to prove their absence), it is often referred to as bug finding. Such tools are usually applied to improve the quality of noncritical programs at a low cost.

Bug-finding tools have the following characteristics:

- They are automatic
- They are neither robust nor complete, instead, they aim at discovering bugs rather quickly, so as to help developers.

Techniques	Automatic	Robust	Complete	Object	Execution
Testing	Yes	No	Yes	Program	Dynamic
Assisted proving	No	Yes	Yes/No	Model	Static
Model checking of finite-state model	Yes	Yes	Yes	Finite Model	Static
Model checking at program level	Yes	Yes	No	Program	Static
Conservative static analysis	Yes	Yes	No	Program	Static
Bug finding	Yes	No	No	Program	Static

Figure 2. An overview of program analysis techniques

#### 1.4 Static Analysis Overview

Static analysis is an automatic technique for program-level analysis that approximates in a conservative manner semantic properties of programs before their execution [17].

This term is applied to the analysis performed by an automated tool, with accompanied human analysis being called program understanding or code review.

The code review of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include

the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification).

It can be argued that software metrics and reverse engineering are forms of static analysis. In fact, deriving software metrics and static analysis are increasingly deployed together, especially in creation of embedded systems, by defining so called software quality objectives.

A growing commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code. Some of the implementation techniques of static analysis include:

- *Model checking* considers systems that have finite state or may be reduced to finite state by abstraction;
- *Data-flow analysis* is a lattice-based technique for gathering information about the possible set of values;
- *Abstract interpretation* models the effect that every statement has on the state of an abstract machine. This abstract machine over-approximates the behaviors of the system: the abstract system is thus made simpler to analyze, at the expense of incompleteness. If properly done, though, abstract interpretation is robust.
- *Use of assertions* in program code. There is tool support for most programming languages.

This term is applied to the analysis carried out by the automated tool, therefore the accompanying analysis of the person is called understanding the program or displaying the code.

The verification of the analysis code carried out by the tools ranges from those that only take into account the behavior of individual instructions and declarations to those that include the complete source code of the program in their analysis. The use of the information obtained from the analysis varies from the selection of possible coding errors to formal methods that mathematically substantiate the properties of a particular program (e.g. its behavior corresponds to that specified in its specification).

It can be argued that software metrics and reverse engineering are forms of static analysis. In fact, software output and static analysis are increasingly being developed together, especially when creating embedded systems by defining so-called software quality goals.

The increasing commercial use of static analysis is to test the properties of software used in security-critical computer systems and to look for potentially vulnerable code. Some of the methods for implementing static analysis include:

- Model checking - takes into account systems that have a finite state or that can be reduced to a finite state by abstraction.
- Data-flow analysis - analysis is a grid method for collecting information about a possible set of values.
- Abstract interpretation - simulates the effect of each statement on the state of the abstract machine. This abstract machine approaches the behavior of the system: In this way, the abstract system becomes easier to analyze due to incompleteness. An abstract interpretation, however, is reliable when executed correctly.
- Use of assertions

## **1.5 Dynamic Software Analysis Overview**

Dynamic analysis is the testing and evaluation of a program by executing data in real-time. The aim is to find bugs in the program during startup and not to study the code repeatedly offline.

By debugging the program in all scenarios for which it was developed, dynamic analysis eliminates the need to artificially create situations that can lead to errors. Other advantages include lower testing and maintenance costs, the identification and elimination of unnecessary program components, and ensuring that the program to be tested is compatible with other programs.

For the analysis of dynamic programs to be effective, the target program must be executed with sufficient test inputs to generate interesting behavior. The use of software testing methods such as B. Code coverage helps to ensure that an appropriate part of the possible program behavior is adhered to. Care should also be taken to minimize the impact of the toolkit on the execution (including the time characteristics) of the target program. Inadequate testing can lead to catastrophic failures.

The dynamic analysis is carried out through program execution and performance monitoring. Testing and profiling is dynamic standard analysis. The dynamic analysis is done precisely because no approximation or abstraction is required: the analysis can examine the actual, precise behavior of the program in the execution process. It is not clear which control flow paths were adopted, which values were calculated, how much memory was used, how long it took for the program to

run, or other interesting values.

Dynamic analysis can be as fast as program execution.

Advantages of dynamic analysis:

- Is able to detect dependencies that are not possible in static analysis.  
Example: dynamic dependencies using reflection, dependency injection, polymorphism.
- Can collect temporal information
- Deals with real runtime values

The disadvantage of dynamic analysis is that its results may not generalize to future executions. There is no guarantee that the test suite over which the program was run (that is, the set of inputs for which execution of the program was observed) is characteristic of all possible program executions. Applications that require correct inputs (such as semantics-preserving code transformations) are unable to use the results of a typical dynamic analysis, just as applications that require precise inputs are unable to use the results of a typical static analysis. Whereas the chief challenge of building a static analysis is choosing a good abstraction function, the chief challenge of performing a good dynamic analysis is selecting a representative set of test cases (inputs to the program being analysed). (Efficiency concerns affect both types of analysis.) A well-selected test suite can reveal properties of the program or of its execution context; failing that, a dynamic analysis indicates properties of the test suite itself, but it can be difficult to know whether a particular property is a test suite artefact or a true program property.

## 1.6 Software Metrics

A software metric is a measure of a property of software or its specifications. Because quantitative measurements are important in all sciences, computer scientists and theorists are constantly trying to introduce similar approaches to software development. The goal is to obtain objective, reproducible and quantifiable indicators that can contain numerous valuable programs for budget planning and planning, cost estimation, quality assurance tests, software debugging, software performance optimization and optimal tasks for employees.

There are three groups of methods of data collection: direct, indirect and independent. The main criterion for grouping techniques is how the researcher should contact members. Direct methods require the researcher direct contact with research participants. Indirect methods allow indirect access to research participants through direct access to the workspace members. Independent methods require that the researcher seek only to objects of study participants, such as source code and



documentation.

- Direct methods include brainstorming and focus groups, interviews and questionnaires, conceptual modelling, doing business diaries, sound work, and general observations shadow observation, active surveillance.
- Indirect methods are also exploring the work of software engineers, but, unlike direct, do not require direct contact with participants in the experiment. The following indirect methods: instrumentation systems and remote monitoring.
- Independent method is a technique that allows to obtain information about software products over which the work of software engineers. The following independent methods - analysis of databases of the work, the analysis of log files tools, documentation analysis, statistical analysis and dynamic systems.

Some software developers point out that simplified measurements can do more harm than good. Others found that metrics have become an integral part of the software development process. The impact of measurements on the psychology of programmers has raised concerns about the detrimental effects on productivity from stress, anxiety, and attempts at performance, while others believe that they have a positive impact on the value of developers for their own work and prevent underestimation. Some argue that the definition of many measurement methods is inaccurate, and therefore it is often unclear how the tools for their calculation will produce a certain result, while others argue that incomplete quantification is better than none. There is evidence that software metrics from government agencies, like the U.S. military, NASA, are widely available.

## 1.7 The SQALE Model

SQALE (Lifecycle Quality Assessment) is a method to support the evaluation of software sources. This is a general method that is independent of language and source analysis tools.

The SQALE quality model is used to formulate and organize non-functional code quality requirements. It is organized in three hierarchical levels. The first level consists of characteristics, the second of sub-characteristics. The third level consists of the requirements for the internal attributes of the source code. These requirements usually depend on the context and language of the software.

The SQALE analysis model contains, on the one hand, the rules for normalizing the key figures and controls in relation to the code and on the other hand the aggregation rules for the normalized values. The SQALE method normalizes the reports obtained using source code analysis tools and converts them into correction costs. To do this, use either the recovery factor or the recovery function. The SQALE

method defines rules for the aggregation of restoration costs either in the structure of the quality model tree or in the hierarchy of the source code artifacts.

### 1.7.1 SQALE Model Indexes

The indicators of the model represent the costs. These costs can be calculated in a unit of work, in a unit of time, or in a unit of money. In all cases, the values of the indices are on a scale of the type of relationship. You can use them to perform all permissible operations for this type of scale. For each element of the source code artifact hierarchy, the cost of restoration associated with that characteristic can be estimated by adding all of the restoration costs associated with the characteristics of the characteristic. The indices of SQALE characteristics are the following [23]:

- Testability Index : STI
- Reliability Index : SRI
- Changeability Index : SCI
- Efficiency Index : SEI
- Security Index : SSI
- Maintainability Index : SMI
- Portability Index : SPI
- Reusability Index : SRuI

The method also defines a global index: for each element of the hierarchy of source code artifacts, the restoration costs that relate to all characteristics of the quality model can be estimated by adding all the restoration costs that are associated with all requirements of the quality model.

This derived measurement is called: SQALE Quality Index [23].

## 2. Tools overview

### 2.1 IBM Rational Software Architect

IBM Rational Software Architect (RSA) is a comprehensive modeling and development environment that uses the Unified Modeling Language (UML) to design the architecture of C ++ and Java 2 Enterprise Edition (J2EE) application and web services. Rational Software Architect is based on Eclipse open-source software and includes features that focus on architecture code analysis, C ++, and UML modeling (MDD) to build robust applications and web services.

Rational Software Architect includes the following capabilities:

1. Built on Eclipse
2. Supports UML
3. Supports model-to-code and code-to-model transformations.

4. Includes all of the capabilities of IBM Rational Application Developer
5. Enables model management for parallel development and architectural re-factoring.
6. Provides visual construction tools

Benefits of using Rational Software Architect include:

- Build a software architecture that supports change with a common platform that facilitates direct rotation, model and code synchronization.
- Accelerate implementation and facilitate support for a successful service-oriented architecture solution (SOA), such as web service with powerful tools and process management.
- Use UML to ensure continuous communication between multiple stakeholders in your software development projects and to use certain specifications to start development quickly.
- Familiarize yourself with distributed projects and control shared information more closely.

## 2.2 iPlasma

iPlasma is an integrated environment for analyzing the quality of object-oriented software systems that includes support for all required analysis phases: from model extraction (including scalable analysis for C ++ and Java) to monitoring based on high-level metrics or code duplication. iPlasma has three main advantages:

- Extensibility of the supported analysis
- Integration with other analysis tools
- Scalability as used in the past to analyze large projects the size of millions of lines of code.

An important task in software analysis is to create a suitable model of the system. The purpose of creating a model is to extract information from the source code that is relevant to a particular purpose. For analyzes that focus on object-oriented design, it is therefore important to know the analyzed system types, functions and variables as well as information about their use, inheritance relationships between classes, call plan, etc.

Based on the extracted information, we can identify different types of design analysis (e.g. metrics, rules based on metrics to identify design problems, quality models, etc.). Next we will discuss how these tests are supported in iPlasma.

iPlasma contains a library of more than 80 state-of-the-art and new design metrics that can be applied at various levels of abstraction, from system-level metrics that

create a system map to primitive metrics that describe details in a single method.

Indicators can be divided into the following categories:

- Size indicators - Measure the size of the analyzed object (e.g. lines of code).
- Complexity Indicators - Measure the complexity of the entity being analyzed (e.g. cyclomatic complexity).
- Connection metrics - Measure data connections between objects (e.g. connections between objects).
- Cohesion Coefficients - Measure the cohesion of classes (e.g. class cohesion).

A detection strategy is a quantified expression of a rule that allows design fragments that match this rule to be recognized in the source code. With this quantification, we can automatically detect design errors in the software system, design units that represent some deviations from good object-oriented design criteria, which prevents the simple development of the system and degrades its design quality. The main problem with any metric-based approach, including detection strategies, is that a pair of thresholds must be used to identify those design objects that have some "abnormal" properties. DSTM (Detection Strategy Tuning Strategy) implements a method that is metaphorically called a tuning machine and can be used to set suitable threshold values for a detection strategy. An important advantage of this method is that the detection strategy can be calibrated for a specific development environment. The downside is that the method requires human feedback over a period of time to get good results.

### **2.3 SonarQube as a platform for continuous analysis**

SonarQube is an open-source platform for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities on 20+ programming languages. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.

SonarQube can record metrics history and provides evolution graphs. SonarQube provides fully automated analysis and integration with Maven, Ant, Gradle, MSBuild and continuous integration tools (Atlassian Bamboo, Jenkins, Hudson, etc.).

### 3. Projects Analysis

#### 3.1 Project overview

Art of Illusion is a free, open source 3D modelling and rendering studio. Many of its capabilities rival those found in commercial programs. Highlights include subdivision surface based modelling tools, skeleton based animation, and a graphical language for designing procedural textures and materials [22].

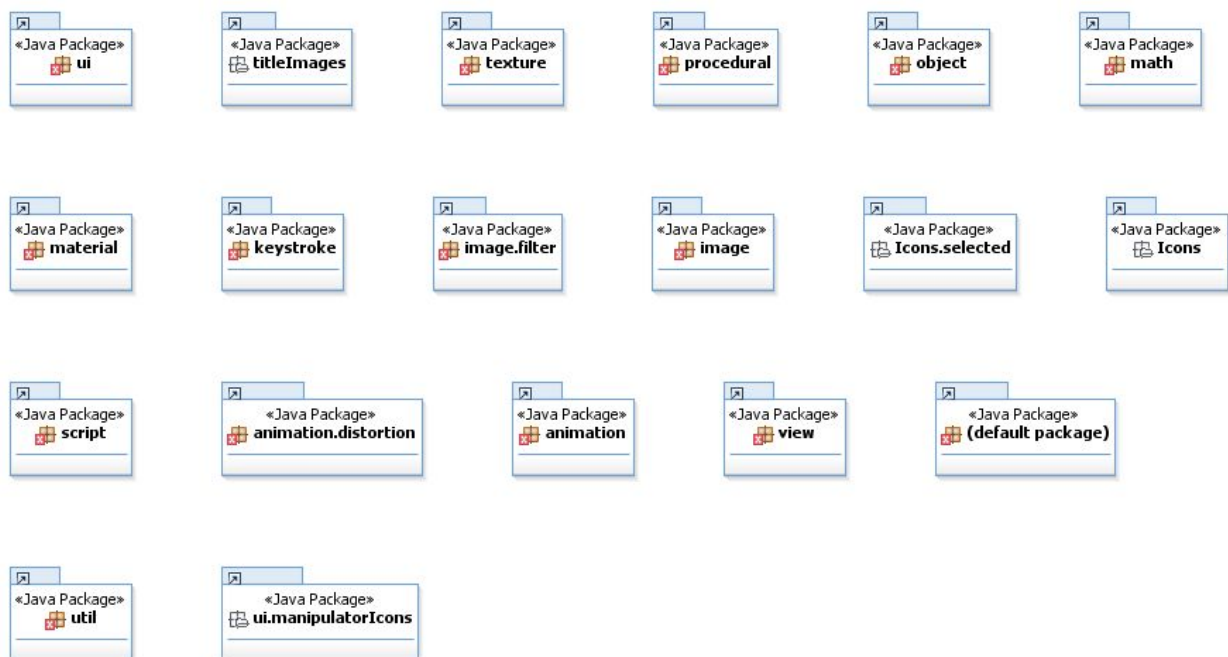
#### 3.2 Structure visualization

##### 3.2.1 Package diagram

The package diagram in a common modeling language shows the relationships between the packages that make up the model.

Package diagrams can use packages that contain usage options to demonstrate the functionality of the software system. Package diagrams can use packages that represent different levels of the software system to illustrate the tiered architecture of the software system. The relationships between these packages can be embellished with labels/stereotypes to show the communication mechanism between the levels.

The diagram of the ArtOfIllusion package is shown in Figure 3. Since it is too large, I have only included part of it in the report.



*Figure 3. ArtOfIllusion package diagram*

##### 3.2.2 Class diagram

The class diagram is the main building block of object-oriented modeling. It is

used both for general conceptual modeling of the systematics of the application and for the detailed modeling of the translation of the models into program code. Class diagrams can also be used for data modeling. The classes in a class diagram represent both the main objects and the interactions in the application and the objects to be programmed. In the class diagram, these classes are shown with fields that contain three parts:

- The upper part contains the name of the class
- The middle part contains the attributes of the class
- The lower part specifies the methods or operations that the class can perform

When designing a system, several classes are identified and summarized in a class diagram to determine the static relationships between these objects. In the case of detailed modeling, the classes of the conceptual design are often divided into several sub-classes.

To further describe the behavior of systems, these class diagrams can be supplemented by a state diagram or a UML state machine. Instead of object class diagrams, role modeling can also be used if you only want to model the classes and their relationships.

The ArtOfIllusion class diagram is shown on Figure 5 and 6. As it has too many classes, I placed only a part of it (classes for one package) into the report.

The depth of inheritance for classes from 'Distortion' class is shown on the Figure 5.

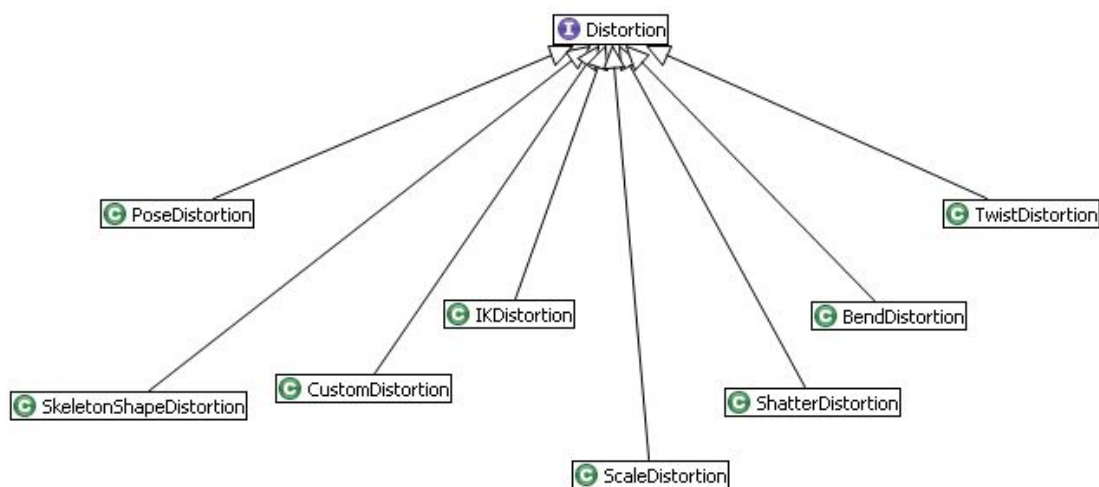


Figure 4. Inheritance tree of 'Distortion' class

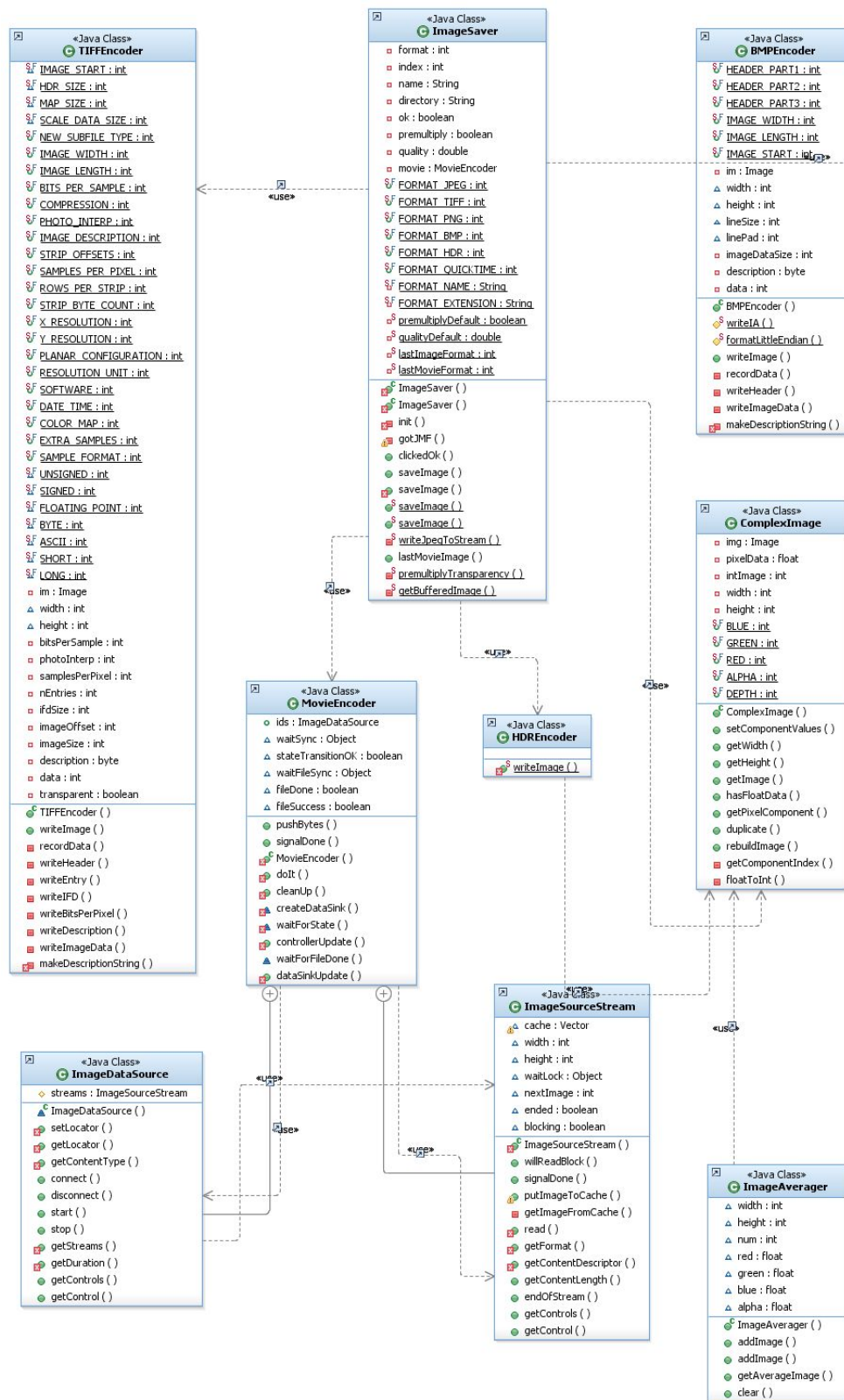


Figure 5. ArtOfIllusion class diagram for “Image” package. Part I



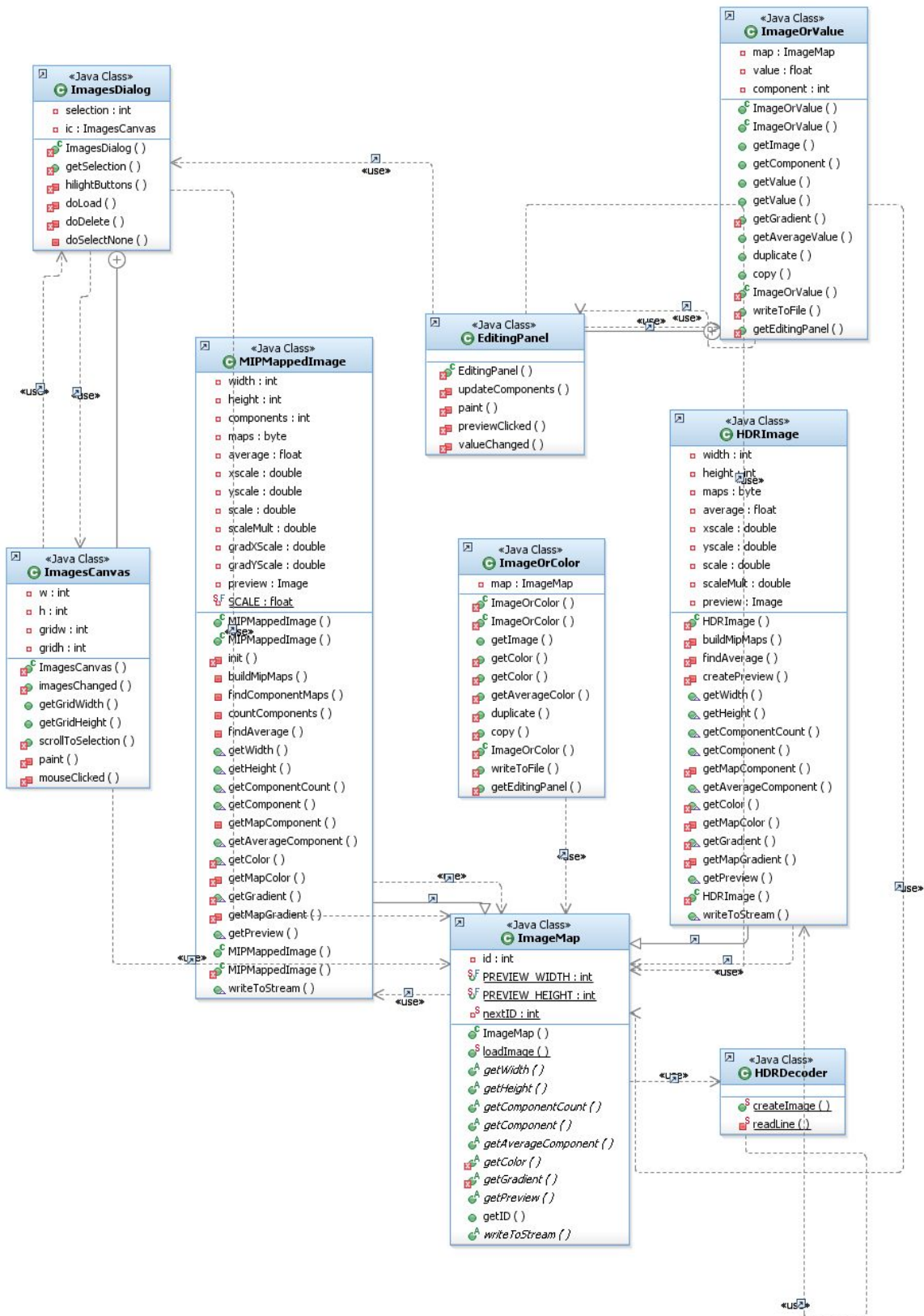


Figure 6. ArtOfIllusion class diagram for “Image” package. Part II

### 3.3 Project metrics

Main directed metrics measured in iPlasma are shown in Table 1.

Each line has a colored percentage. The percentage is derived from the ratio of the number in this line to the number below.



Table 1. Metrics

Code	Description
<b>NDD</b>	Number of direct descendants
<b>HIT</b>	Height of inheritance tree
<b>NOP</b>	Number of packages
<b>NOC</b>	Number of classes
<b>NOM</b>	Number of methods
<b>LOC</b>	Lines of code
<b>CYCLO</b>	Cyclomatic complexity
<b>CALL</b>	Calls per method
<b>FOUT</b>	Fan out (number of other methods called by a given method)

The numbers indicate the ratio; Colors indicate where the conditions fit into industry-standard areas (derived from numerous open-source projects). Each ratio is either green (inside the area), blue (below the area), or red (outside the area). For the Struts code base, NDD and CYCLO are outside of the industry standards for these values, and LOC and NOM are listed below.

Table 2. Industry ranging for metrics

	Low	Medium	High
<b>CYCLO / Line</b>	0.16	0.20	0.24
<b>LOC / method</b>	7	10	13
<b>NOM / class</b>	4	7	10
<b>NOC / package</b>	6	17	26
<b>CALLS / method</b>	2.01	2.62	3.20
<b>FANOUT / call</b>	0.56	0.62	0.68

Metrics Pyramid for ArtOfIllusion generated in iPlasma is shown on the Figure 7.

## System Detail: **ArtOfIllusion-master**

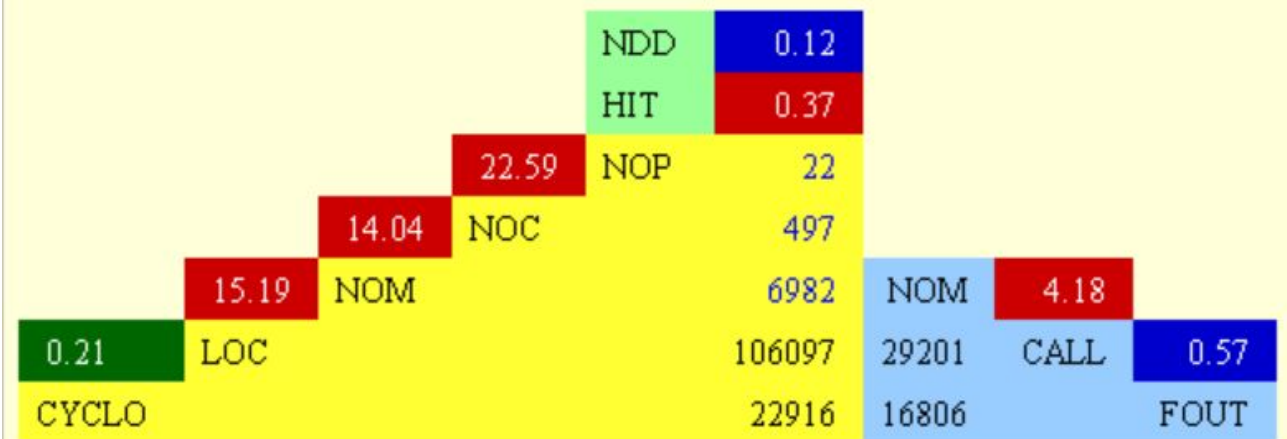


Figure 7. iPlasma Metrics Pyramid for ArtOfIllusion

So, the metrics on this pyramid for ArtOfIllusion project indicate that:

- Class hierarchies tend to be tall and narrow
- Classes tend to be:
  - o rather large (i.e. they define many methods)
  - o organized in rather fine-grained packages
- Methods tend to:
  - o be rather long yet having a rather simple logic (i.e. few conditional branches)
  - o call several methods from few other classes (i.e. low coupling dispersion)

Among indirect metrics, for ArtOfIllusion project I've measured:

- *ATDF* – access to foreign data is the number of attributes from unrelated classes that are accessed directly or by invoking accessor methods. Few (i.e. min value) for this metric is empirically defined as 3.

The value distribution per number of classes for ArtOfIllusion project is shown on Figure 8.

- *WMC* – weighted method count. The sum of the statical complexity of all methods of a class. The CYCLO metric is used to quantify the method's complexity.

The value distribution per number of classes for ArtOfIllusion project is shown

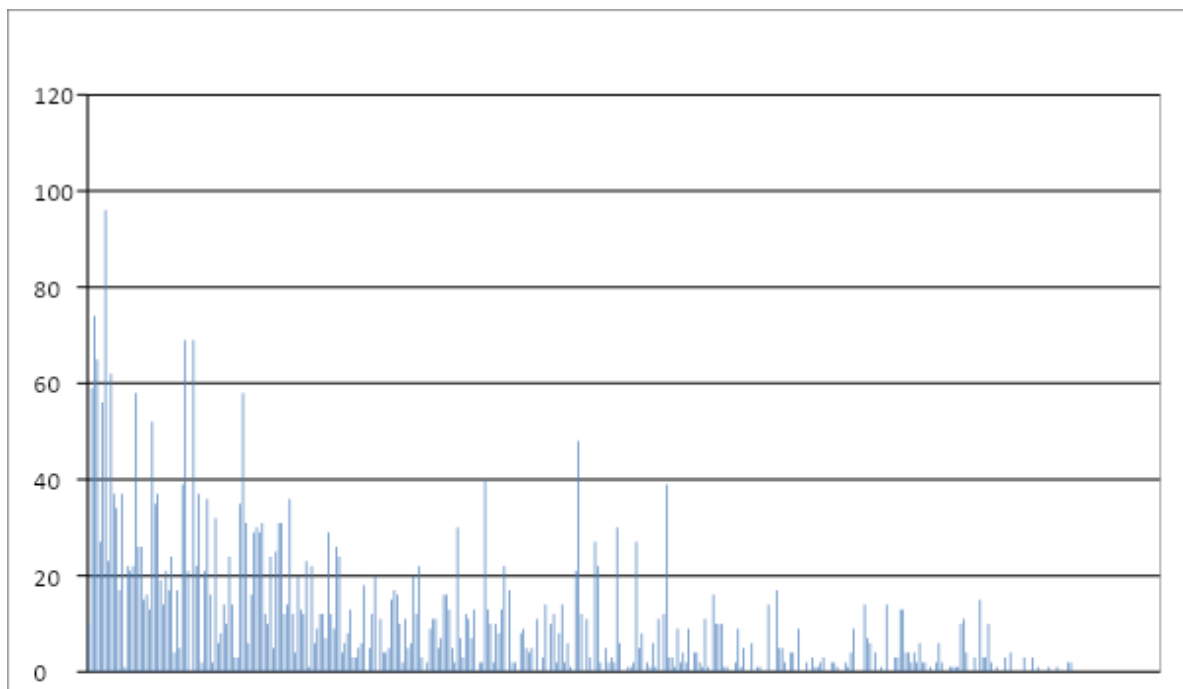
on Figure 9

- *TCC* – tight class cohesion is the relative number of method pairs of a class that access at least one common attribute of that class. Value of this metric is from 0 to 1, so one third equals  $1/3$ .

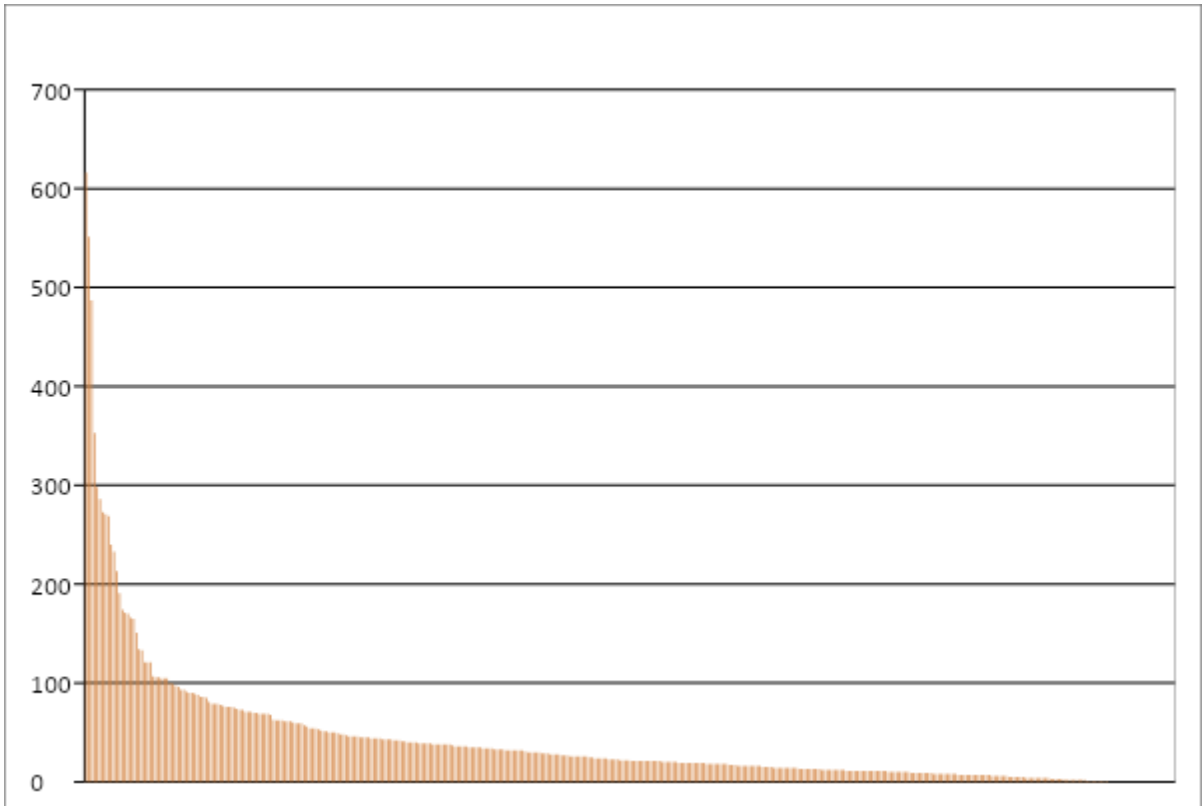
The value distribution per number of classes for ArtOfIllusion project is shown on Figure 10.

- *NOM* – Number of methods per class

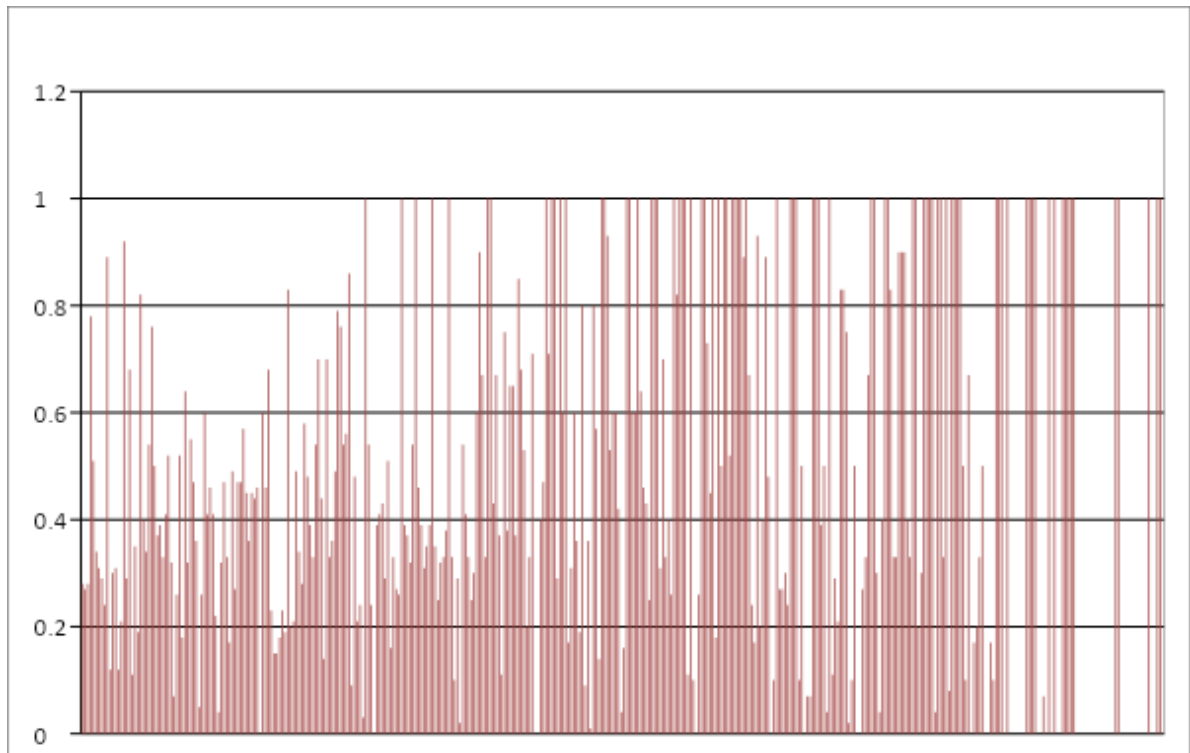
The value distribution per number of classes for ArtOfIllusion project is shown on Figure 11.



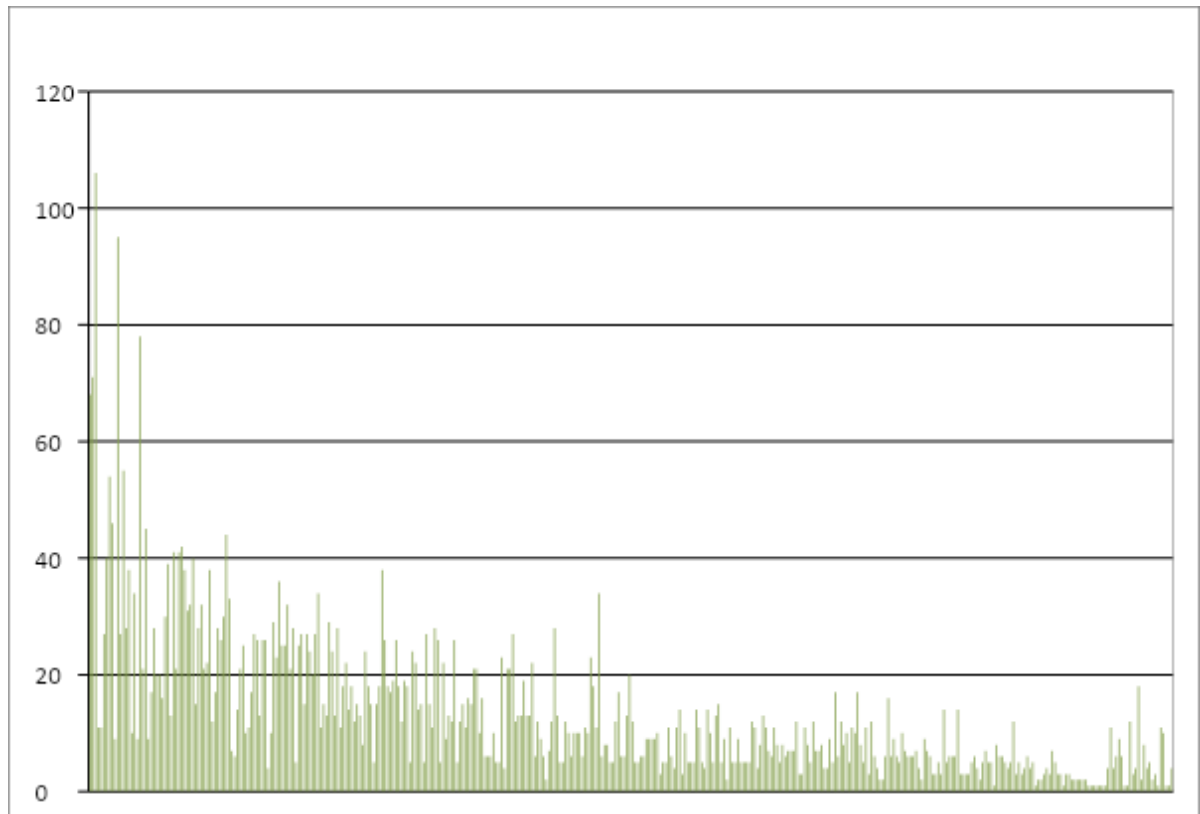
**Figure 8.** *ATFD metric value distribution for ArtOfIllusion project classes*



































































*Figure 9. WMC metric value distribution for ArtOfIllusion project classes*



*Figure 10. TCC metric value distribution for ArtOfIllusion project classes*



*Figure 11. NOM metric value distribution for ArtOfIllusion project classes*

Rule	Metric
 Basic Metrics	
  Average number of comments	447.80
  Average number of methods	14.77
  Comment/Code Ratio	7.52 %
  Lines of code	89376.00
  Number of attributes	1611.00
  Number of comments	6717.00
  Number of constructors	595.00
  Number of lines	118956.00
  Number of methods	5331.00
  Number of parameters	7401.00
  Number of types per package	361.00
 Cohesion Metrics	
  Lack of cohesion 1	81.61
  Lack of cohesion 2	0.70
  Lack of cohesion 3	0.77
 Complexity Metrics	
  Average block depth	1.39
  Cyclomatic complexity	3.12
  Maintainability index	259.29
  Weighted methods per class	18654.00
 Dependency Metrics	
  Normalized Distance	0.86
 Halstead Metrics	
  Difficulty level	18822.20
  Effort to implement	9539156676...
  Number of delivered bugs	6959.09
  Number of operands	282333.00
  Number of operators	310635.00
  Number of unique operands	330.00
  Program length	592968.00
  Program level	0.00
  Program vocabulary size	374.00
  Program volume	5068034.91
 Inheritance Metrics	

**Figure 12. IBM RSA Gathered metrics for ArtOfIllusion project**

All described metrics were used to identify so called God classes in the ArtOfIllusion project

God class problem is characterized in literature this way:

- Top-level classes in a design should share work uniformly.
- Beware of classes with much non-communicative behavior.
- Beware of classes that access directly data from other classes.

We are going to find them using the Goal–Question–Metric (GQM) model that defines the necessary obligations for setting objectives before embarking on any software measurement activity. Main steps are:

- List the major Goals for which metrics are going to be employed.
- From each goal derive the Questions that must be answered to determine if the goals are met.

- Decide what Metrics must be collected to answer the questions.

So questions can be formulated like this:

- Is class excessively complex?
- Is class highly coupled?
- Does class accesses foreign data too much?

First question can be answered by metric WMC, second by metric TCC, third by metric ATFD.

So, iPlasma identified **41 God and 43** classes in ArtOfIllusion project. Some of them are.

Table 3. God and Data Classes

God Classes	Data Classes
Actor	AnimationPrevi
ArtOfIllusion	ewer
Camera	BiasModule
Curve	BMPEncoder
CustomDistortionTr	CosineModule
ack	ExpModule
EditKeyframesDialo	GainModule
g	GridModule
ExprModule	InfoBox
ExternalObject	InterpModule
ImageSaver	IOPort
LayoutWindow	JitterModule
MaterialPreviewer	KeystrokeRecor
MeshEditorWindow	d
Module	Light
Object3D	LogModule
ObjectEditorWindo	Mapping3D
w	Marker
ObjectInfo	MaterialSpec
ObjectPropertiesPan	MeshVertex
el	MeshViewer
ObjectViewer	ObjectMaterial
OutlineFilter	Dialog
ProceduralDirection	OPort
alLight	ParameterModu
ProceduralMaterial3	le
D	PointInfo
ProceduralPointLigh	PowerModule
t	ProductModule
	Property

ProceduralPositionT rack	RandomModule
ProceduralRotationT rack	RatioModule
ProceduralTexture2 D	RenderingMesh
ProceduralTexture3 D	RenderingTrian gle
ProcedureEditor	RenderSetupDia log
Scene	ResetButton
Score	SelectionInfo
SplineMesh	SineModule
SplineMeshEditorWi ndow	SphericalModul e
SpotLight	SqrtModule
ThemeManager	TextureSpec
TreeList	TimeAxis
TriangleMesh	Token
TriMeshEditorWind ow	UVMappedTria ngle
Tube	UVWMappedTr iangle
UVMapping	WireframeMesh
UVMappingViewer	WoodModule
UVMappingWindow	
ViewerCanvas	

Having analysed selected project (ArtOfIllusion) with metrics, I've discovered that:

- *Class hierarchies* tend to be tall and narrow (i.e. inheritance trees tend to have many depth-levels and base-classes with few directly derived sub-classes)
- *Classes* tend to be:
  - o rather large (i.e. they define many methods)
  - o organized in rather fine-grained packages (i.e. few classes per package)
- *Methods* tend to:
  - o be rather long yet having a rather simple logic (i.e. few conditional branches)
  - o call several methods from few other classes (i.e. low coupling dispersion)



### 3.4 Unit tests

Unit testing is a method of testing individual units of source code to determine if they can be used. The device is the smallest part of the program tested. In procedural programming, a unit can be an entire module, but more often a single function or procedure. In object-oriented programming, a unit is often an integer interface, e.g. B. a class, but can be a single method. Test units are created during the development of programmers or periodic white field testers.

Ideally, each test case is independent of the others: Substitutes such as method connectors, models, counterfeits, and test wiring harnesses can be used to test the module in isolation. Block tests are typically written and performed by software developers to ensure that the code conforms to the design and behaves as intended. The implementation can range from very manual (pencil and paper) to formalization in the context of building automation.

#### Unit tests for ArtOfIllusion project

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package sample;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 * Sample JUnit 4 test class.
 *
 * Note that it does not need to be a subclass of any particular class.
 *
 * @author nb
 */
```

```

public class UtilsJUnit4Test {

    public UtilsJUnit4Test() {
    }

    /**
     * Initialization method that is called only once. It is run before
     * any test method is executed. It is run even sooner than any
     * {@code @Before} method. If a test class is a test suite, its
     * {@code @BeforeClass} method is called even sooner than any of test
     * classes contained in the suite. But: If any of superclasses
     * of a test class has some {@code @BeforeClass} method, the superclass
     * {@code @BeforeClass} is called before. There may be only one method
     * annotated with the {@code @BeforeClass} annotation.
     * The name of the method is irrelevant - the only mandatory attributes are:
     * the {@code @BeforeClass} annotation,
     * the {@code public} and {@code static} modifiers
     * {@code void} return type and
     * no arguments.
     */
    @BeforeClass
    public static void setUpClass() throws Exception {
        /**
         * E.g. establish connection to a database so that it does not need to
         * be established for each test separately.
         */
        System.out.println("UtilsJUnit4Test: @BeforeClass method");
    }

    /**
     * Tear down method that is called only once. It is run after all test
     * methods and their {@code @After} methods are executed.
     * If a test class is a test suite, its {@code @AfterClass} method is called
     * even only after all test classes contained in the suite finish running.
     * But: If any of the superclasses of a test class has some
     * {@code @BeforeClass} method, the superclass {@code @BeforeClass} is

```

```

* called later. There may be only one method annotated with the
* {@code @AfterClass} annotation.
* The name of the method is irrelevant - the only mandatory attributes are:
* the {@code @AfterClass} annotation,
* the {@code public} and {@code static} modifiers
* {@code void} return type and
* no arguments.
*/

```

`@AfterClass`

```

public static void tearDownClass() throws Exception {
    /*
     * E.g. disconnect from the database - i.e. the complement to the action
     * performed in setUpClass().
     */
    System.out.println("* UtilsJUnit4Test: @AfterClass method");
}

```

`/**`

```

* Test initialization method. It is executed before each test method
* of this test class. There may be multiple test initialization methods
* - in such a case, all these methods are executed before each test method
* but order of their execution is undefined. The name of the method is
* irrelevant - the only mandatory attributes are:
* the {@code @Before} annotation,
* the {@code public} access modifier
* the {@code void} return type and
* no arguments.
*/

```

`@Before`

```

public void setUp() {
    System.out.println("* UtilsJUnit4Test: @Before method");
}

```

`/**`

```

* Test shutdown method. It is executed after each test method
* of this test class. There may be multiple test shutdown methods

```

```

* - in such a case, all these methods are executed after each test method
* but order of their execution is undefined. The name of the method is
* irrelevant - the only mandatory attributes are:
*   the {@code @After} annotation,
*   the {@code public} access modifier
*   the {@code void} return type and
*   no arguments.
*/

@After
public void tearDown() {
    System.out.println("* UtilsJUnit4Test: @After method");
}

/**
 * Test of concatWords method, of class Utils.
 * Simple test method. It does not need to follow any naming conventions,
 * just the {@code @Test} annotation is important.
 */

@Test
public void helloWorldCheck() {
    System.out.println("* UtilsJUnit4Test: test method 1 - helloWorldCheck()");
    assertEquals("Hello, world!", Utils.concatWords("Hello", " ", "world", "!"));
}

/**
 * Test with timeout. If the testing routine does not finish in one second,
 * it is interrupted and the test fails. The timeout is declared by an
 * argument to the {@code @Test} annotation.
 */

@Test(timeout = 1000)
public void testWithTimeout() {
    System.out.println("* UtilsJUnit4Test: test method 2 - testWithTimeout()");
    final int factorialOf = 1 + (int) (30000 * Math.random());
    System.out.println("computing " + factorialOf + "!");
    System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
}

```

```

/**
 * Test of expected exception. If a given exception is not thrown during the
 * tested routine, the test fails. If the exception is thrown, the test
 * passes. The expected annotation is declared by an argument to the
 * {@code @Test} annotation.
 */
@Test(expected = IllegalArgumentException.class)
public void checkExpectedException() {
    System.out.println("* UtilsJUnit4Test: test method 3 -
checkExpectedException());
    final int factorialOf = -5;
    System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
}

/**
 * Test that is temporarily disabled simply by the special {@code @Ignore}
 * annotation.
 */
@Ignore
@Test
public void temporarilyDisabledTest() throws Exception {
    System.out.println("* UtilsJUnit4Test: test method 4 -
checkExpectedException());
    assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));
}
}

```

### 3.5 Continuous Code Analysis with the SonarQube

The SonarQube uses an evolved SQALE model.

Bugs, Vulnerabilities and Code Smells are main metrics to measure.

- Bugs: Code that is demonstrably wrong or highly likely to yield unexpected behaviour.
- Vulnerabilities: Code that is potentially vulnerable to exploitation by hackers.
- Code Smells: Will confuse maintainers or give them pause. Not only ratings, but also approximate remediation efforts.

#### 3.5.1 Three Lines of Analysis

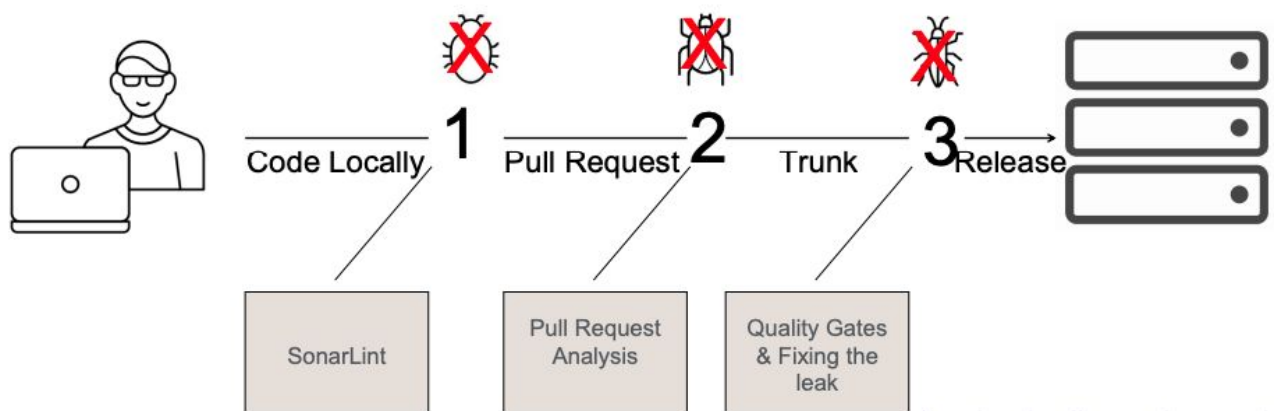


Figure 13. SonarQube Three Lines of Analysis

First line is a SonarLint, that runs in IDE and analyses new code and highlights issues if any.

Second, pull request analysis. When a pull request (PR) is submitted to the repository user comment the changed code with any new issues. After it is up to the manual reviewer to decide if the issues are critical or accepted in the context before merging the PR.

Finally, quality gates and leak management. If previous prevention fails, there is an ability to set up a collection of go/no-go conditions that indicate whether or not the project is releasable - Quality Gate. In case the project fails the Quality gate then the tool automatically informs about it.

Security and Reliability ratings are based on the severity of the worst open issue in that domain [24]:

- E - Blocker
- D - Critical
- C - Major
- B - Minor
- A - Info or no open issues

For Maintainability the rating is based on the ratio of the size of the code base to the estimated time to fix all open Maintainability issues [24]:

- $\leq 5\%$  of the time that has already gone into the application, the rating is A
- between 6 to 10% the rating is a B
- between 11 to 20% the rating is a C
- between 21 to 50% the rating is a D
- anything over 50% is an E

### 3.5.2 SonarQube results for ArtOfIllusion project

The SonarQube provides comprehensive code analysis. The tool found 223 bugs, 125 vulnerabilities, 15 security hotspots. All these categories are marked with rating E, which means Blocker and must be fixed before next release (Figure 14.).

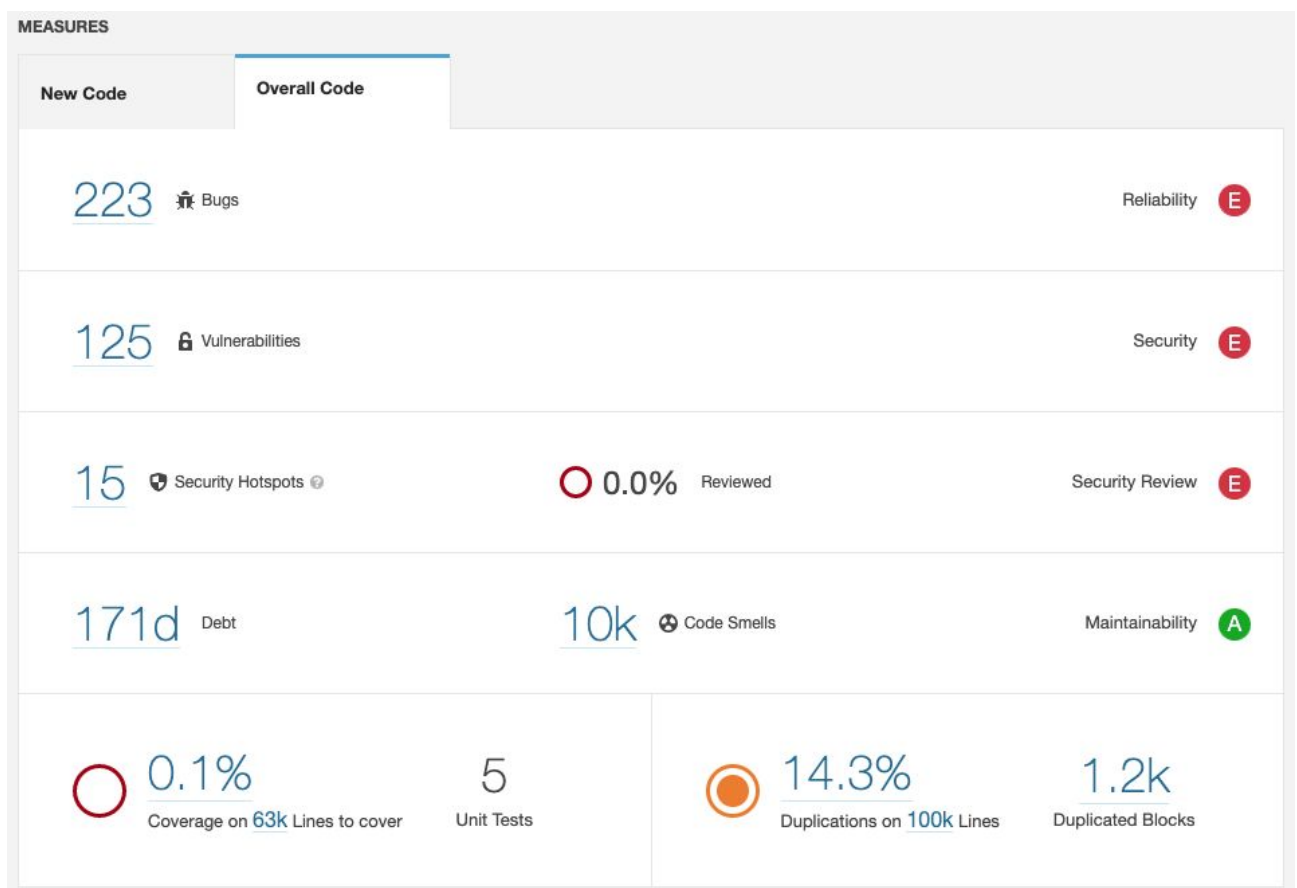
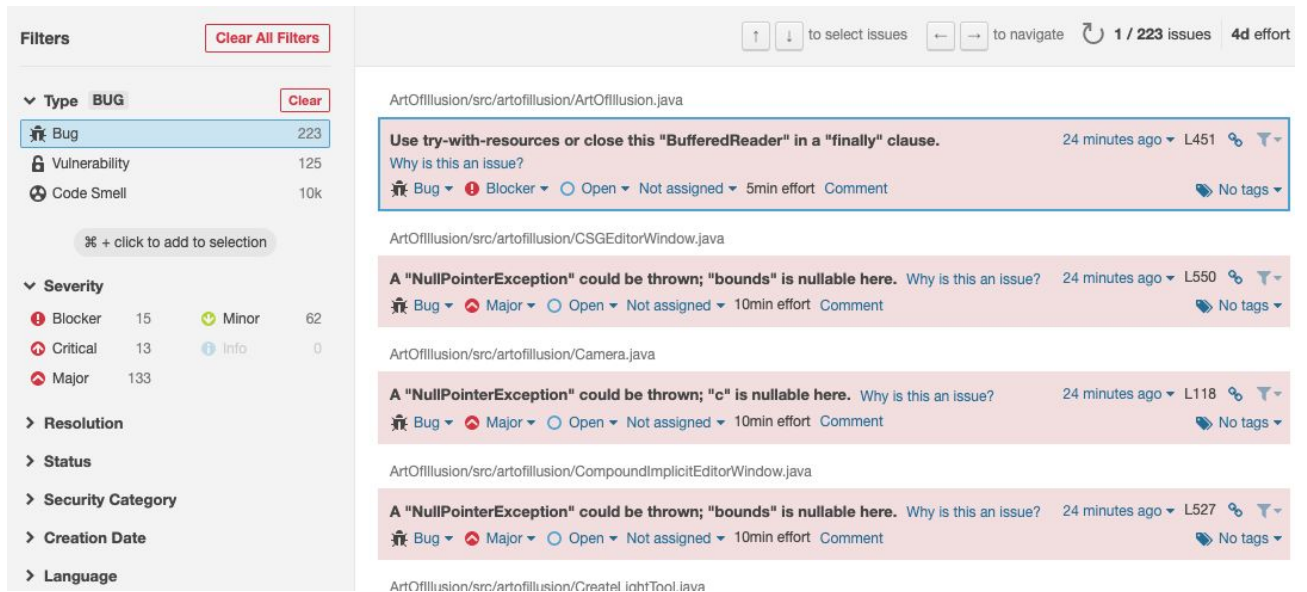


Figure 14. SonarQube Overall Code Analysis Results



**Filters** [Clear All Filters](#)

▼ **Type** **BUG** [Clear](#)

- 🐛 Bug 223
- 🔒 Vulnerability 125
- 💩 Code Smell 10k

🔍 + click to add to selection

▼ **Severity**

- 🔴 Blocker 15
- 🟡 Minor 62
- 🔴 Critical 13
- 🟢 Info 0
- 🔴 Major 133

► **Resolution**

► **Status**

► **Security Category**

► **Creation Date**

► **Language**

1 / 223 issues 4d effort

ArtOfIllusion/src/artofillusion/ArtOfIllusion.java

**Use try-with-resources or close this "BufferedReader" in a "finally" clause.** 24 minutes ago L451

Why is this an issue?

🐛 Bug 🔴 Blocker 🔵 Open ⚪ Not assigned 5min effort Comment No tags

ArtOfIllusion/src/artofillusion/CSGEditorWindow.java

**A "NullPointerException" could be thrown; "bounds" is nullable here.** Why is this an issue? 24 minutes ago L550

🐛 Bug 🔴 Major 🔵 Open ⚪ Not assigned 10min effort Comment No tags

ArtOfIllusion/src/artofillusion/Camera.java

**A "NullPointerException" could be thrown; "c" is nullable here.** Why is this an issue? 24 minutes ago L118

🐛 Bug 🔴 Major 🔵 Open ⚪ Not assigned 10min effort Comment No tags

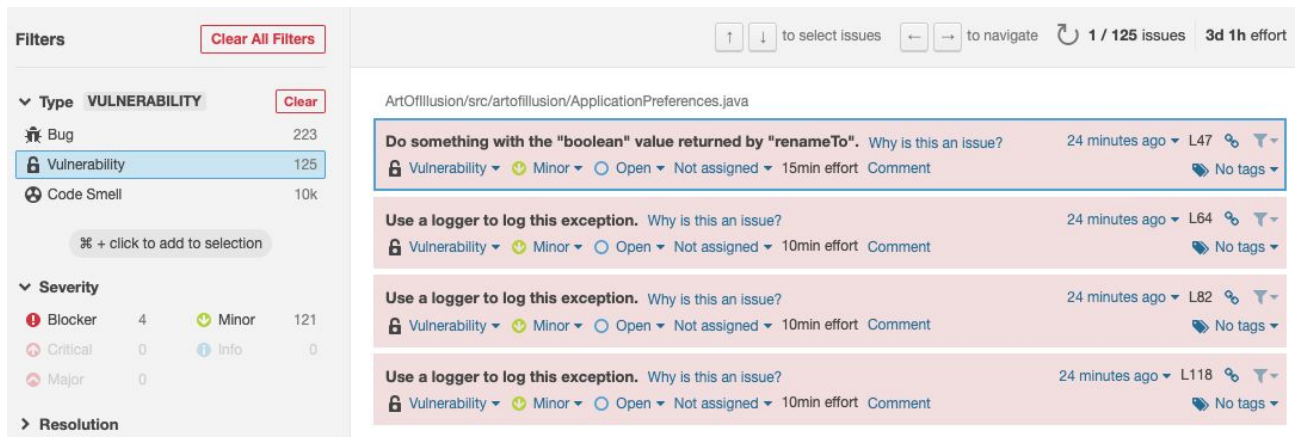
ArtOfIllusion/src/artofillusion/CompoundImplicitEditorWindow.java

**A "NullPointerException" could be thrown; "bounds" is nullable here.** Why is this an issue? 24 minutes ago L527

🐛 Bug 🔴 Major 🔵 Open ⚪ Not assigned 10min effort Comment No tags

ArtOfIllusion/src/artofillusion/CreatelightTool.java

Figure 15. SonarQube Bug Analysis Results



**Filters** [Clear All Filters](#)

▼ **Type** **VULNERABILITY** [Clear](#)

- 🐛 Bug 223
- 🔒 Vulnerability 125
- 💩 Code Smell 10k

🔍 + click to add to selection

▼ **Severity**

- 🔴 Blocker 4
- 🟡 Minor 121
- 🔴 Critical 0
- 🟢 Info 0
- 🔴 Major 0

► **Resolution**

1 / 125 issues 3d 1h effort

ArtOfIllusion/src/artofillusion/ApplicationPreferences.java

**Do something with the "boolean" value returned by "renameTo".** Why is this an issue? 24 minutes ago L47

🔒 Vulnerability 🔴 Minor 🔵 Open ⚪ Not assigned 15min effort Comment No tags

**Use a logger to log this exception.** Why is this an issue? 24 minutes ago L64

🔒 Vulnerability 🔴 Minor 🔵 Open ⚪ Not assigned 10min effort Comment No tags

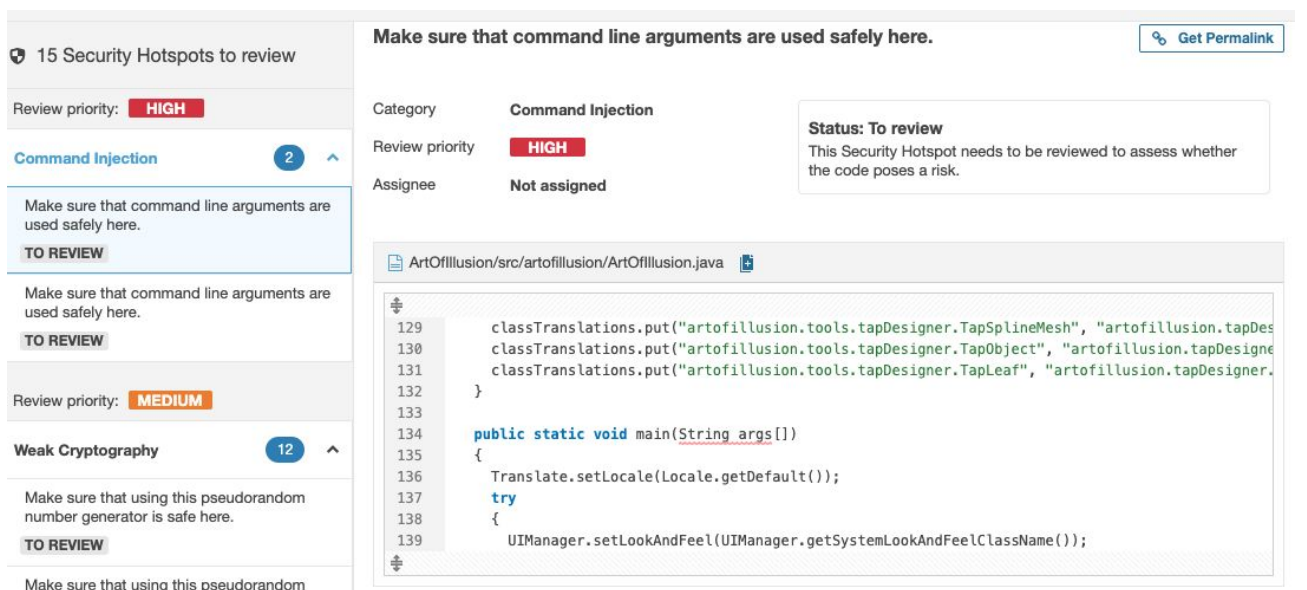
**Use a logger to log this exception.** Why is this an issue? 24 minutes ago L82

🔒 Vulnerability 🔴 Minor 🔵 Open ⚪ Not assigned 10min effort Comment No tags

**Use a logger to log this exception.** Why is this an issue? 24 minutes ago L118

🔒 Vulnerability 🔴 Minor 🔵 Open ⚪ Not assigned 10min effort Comment No tags

Figure 17. SonarQube Vulnerabilities Analysis Results



15 Security Hotspots to review

Review priority: **HIGH**

**Command Injection** 2

Make sure that command line arguments are used safely here.

**TO REVIEW**

Make sure that command line arguments are used safely here.

**TO REVIEW**

Review priority: **MEDIUM**

**Weak Cryptography** 12

Make sure that using this pseudorandom number generator is safe here.

**TO REVIEW**

Make sure that using this pseudorandom

**Make sure that command line arguments are used safely here.** [Get Permalink](#)

Category: **Command Injection**

Review priority: **HIGH**

Assignee: **Not assigned**

**Status: To review**  
This Security Hotspot needs to be reviewed to assess whether the code poses a risk.

ArtOfIllusion/src/artofillusion/ArtOfIllusion.java

```

129     classTranslations.put("artofillusion.tools.tapDesigner.TapSplineMesh", "artofillusion.tapDes
130     classTranslations.put("artofillusion.tools.tapDesigner.TapObject", "artofillusion.tapDesigne
131     classTranslations.put("artofillusion.tools.tapDesigner.TapLeaf", "artofillusion.tapDesigner.
132 }
133
134 public static void main(String args[])
135 {
136     Translate.setLocale(Locale.getDefault());
137     try
138     {
139         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

```

Figure 18. SonarQube Security Hotspots Analysis Results



## Conclusions

As a summary of techniques for software analysis, represented on Figure 1, due to the computability barrier, no technique can provide fully automatic, robust, and complete analyses. Testing sacrifices robustness. Assisted proving is not automatic (even if it is often partly automated, the main proof arguments generally need to be human provided). Model-checking approaches can achieve robustness and completeness only with respect to finite models, and they generally give up completeness when considering programs (the incompleteness is often introduced in the modeling stage). Static analysis gives up completeness (though it may be designed to be precise for large classes of interested programs). Last, bug finding is neither robust nor complete. Another important dimension is scalability. In practice, all approaches have limitations regarding scalability, although these limitations vary depending on the intended applications (e.g., input programs, target properties, and algorithms used).

The application must be analysed on all stages during the development life cycle.

During the design phase, tools like IBM RSA one can easily find architectural bottlenecks in the software and prevent them cheaply and quickly before the start of development.

Already implemented code could be analysed in a continuous integration environment by a tool like SonarQube. Property configured metrics and quality gates provide scalability and continuous analysis that fits modern Agile development life cycle.

## Sources

1. Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, "Using Static Analysis to Find Bugs," IEEE Software, vol. 25, no. 5, pp. 22-29, Sep./Oct. 2008
2. Brian Chess, Jacob West (Fortify Software) (2007). Secure Programming with Static Analysis. Addison-Wesley.
3. Flemming Nielson, Hanne R. Nielson, Chris Hankin (1999, corrected 2004). Principles of Program Analysis. Springer.
4. Integrate static analysis into a software development process  
<http://www.embedded.com/shared/printableArticle.jhtml?articleID=193500830>
5. Code Quality Improvement - Coding standards conformance checking (DDJ)  
<http://www.ddj.com/dept/debug/189401916>
6. Kolawa, Adam. "When, Why, and How: Code Analysis". The Code Project. Retrieved 19 October 2010.
7. Kaner, Dr. Cem, Software Engineer Metrics: What do they measure and how do we know? <http://www.kaner.com/pdfs/metrics2004.pdf>
8. IBM Rational Software Architect family of products information centers  
<http://www.ibm.com/support/docview.wss?uid=swg27010908>
9. Rational Software Architect Wiki on IBM developerWorks  
<https://www.ibm.com/developerworks/wikis/display/RSA/Home>
10. Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. Communications of the ACM, 53(2):66–75, 2010.
11. David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. IBM Systems Journal, 22(3):229–245, 1983.
12. Proceedings of Metrics 2005, Como, 2005.
13. Introduction to UML 2 Package Diagrams by Scott W. Ambler  
<http://www.agilemodeling.com/artifacts/packageDiagram.htm>
14. Scott W. Ambler (2009) UML 2 Class Diagrams. Webdoc 2003-2009. Accessed Dec 2, 2009  
<http://www.agilemodeling.com/artifacts/classDiagram.htm>

15. David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.
16. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM Press, 2005.
17. Rival, Xavier. *Introduction to Static Analysis*. The MIT Press, 2020.
18. Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95– 120, 1988.
19. Edmund C. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
20. Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, forthcoming.
21. Junghee Lim and Thomas W. Reps. A system for generating static analyzers for machine instructions. In *International Conference on Compiler Construction (CC)*, pages 36–52. Springer, 2008.
22. Art of Illusion, <http://www.artofillusion.org/>
23. SQALE Model, <https://en.wikipedia.org/wiki/SQALE>
24. SonarQube Metric Definitions  
<https://docs.sonarqube.org/7.1/MetricDefinitions.html>