CWE-Top-25 SecurityReport

June 20 2017

Sec_App

Version: My Version

PURPOSE OF DOCUMENT:

The purpose of this report is to provide an objective assessment of the health of the application. The primary audience for this report are development and executive teams who are responsible for Training_82.

RevisionHistory

Version	Date	Created	Comment
1.0	MM/DD/YY	User_Name	

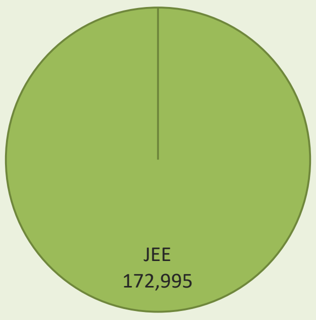
Table of Contents

1. Executive Summary

This Application Assessment evaluates the overall health of the Training_82 application.

Training_82 is a small application and has a good quality with a ***Total Quality Indicator (TQI) of 3.48 on a scale of 4.*** Each of the additional health metrics and their scores along with the other facts about the application are identified below.

1.1. Application Characteristics



Top 5 Technologies

Name	LoC
JEE	172,995

Technical Size

Name	Value
kLoC	173
Files	3,028
Classes	3,685
SQL Art.	0
Tables	0

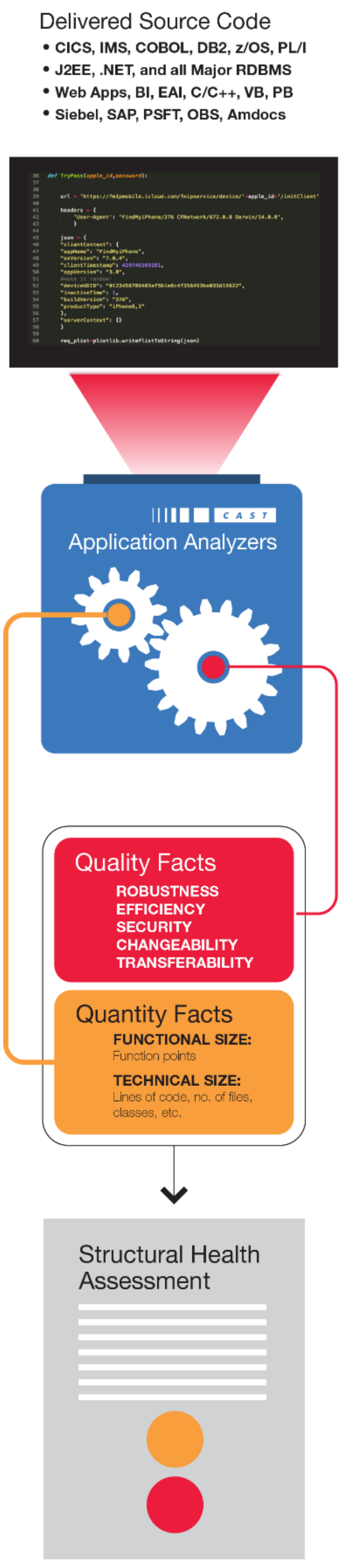
1.2. Assessment Highlights

Statistics on Violations

Name	Value
Critical Violations	2,074
per File	0.68
per kLoC	11.99
Complex Objects	119
With Violations	70

Rule Name	# Violations
Close the outermost stream ASAP	569
Avoid transactions with database resource open capability but without the associated close capability	332
Avoid transactions with too many severe Programming Practices - Error and Exception Handling issues along the path	272
Avoid using Fields (non static final) from other Classes	193
Avoid empty catch blocks	170
Close database resources ASAP	155
CWE-134: Avoid uncontrolled format string	99

2. Assessment Approach Overview


Delivered Source Code
• CICS, IMS, COBOL, DB2, z/OS, PL/I
• J2EE, .NET, and all Major RDBMS
• Web Apps, BI, EAI, C/C++, VB, PB
• Siebel, SAP, PSFT, OBS, Amdocs
CAST
Application Analyzers
Quality Facts
ROBUSTNESS
EFFICIENCY
SECURITY
CHANGEABILITY
TRANSFERABILITY
Quantity Facts
FUNCTIONAL SIZE:
Function points
TECHNICAL SIZE:
Lines of code, no. of files, classes, etc.
Structural Health Assessment

```
def TryPass(apple_id,password):


    url = 'https://fmipmobile.icloud.com/fmipservice/device/'+apple_id+'/initClient'

    headers = {
        'User-Agent': 'FindMyiPhone/376 CFNetwork/672.0.8 Darwin/14.0.0',
        }

    json = {
    "clientContext": {
    "appName": "FindMyiPhone",
    "osVersion": "7.0.4",
    "clientTimestamp": 429746389281,
    "appVersion": "3.0",
    #make it random!
    "deviceUDID": "0123456789485ef5b1e6c4f356453be033d15622",
    "inactiveTime": 1,
    "buildVersion": "376",
    "productType": "iPhone6,1"
    },
    "serverContext": {}
    }

    req_plist=plistlib.writePlistToString(json)
```

This assessment is an effort to determine the overall health of the application and identify any risks that may be inherent within the application. The assessment determines whether the application is constructed according to industry best practices, follows best practices for software engineering, and is maintainable.

This assessment is focused solely on the source code and database structure with no view to functionality provided by backend services.

The assessment leverages the CAST Application Intelligence Platform (AIP), the leading automated code analysis platform, with coverage of all major development tools and languages. CAST AIP automatically scans and analyzes all of the source code and database elements that are part of an enterprise system. CAST AIP applies over 1,000 metrics based on standards and measurements developed by the Software Engineering Institute (SEI), International Standards Organization (ISO), Consortium for IT Software Quality (CISQ), and Institute of Electrical and Electronics Engineers (IEEE). These metrics objectively measure software quality.

The primary Application Health Factors that are addressed follow on the next page.

Health Factor	Description	Example business benefits
Robustness	Attributes that affect the stability of the application and the likelihood of introducing defects when modifying it	• Improves availability of the business function or service • Reduces risk of loss due to operational malfunction • Reduces cost of application ownership by reducing rework
Efficiency	Attributes that affect the performance of an application	• Reduces risk of losing customers from poor service or response • Improves productivity of those who use the application • Increases speed of making decisions and providing information • Improves ability to scale application to support business growth
Security	Attributes that affect an application's ability to prevent unauthorized intrusions	• Improves protection of competitive information-based assets • Reduces risk of loss in customer confidence or financial damages • Improves compliance with security-related standards and mandates
Transferability	Attributes that allow new teams or members to quickly understand and work with an application	• Reduces inefficiency in transferring application work between teams • Reduces learning curves • Reduces lock-in to suppliers
Changeability	Attributes that make an application easier and quicker to modify	• Improves business agility in responding to markets or customers • Reduces cost of ownership by reducing modification effort

3. How Can Technology Address Application Quality Challenges?

The quality attributes of an application can be characterized by the quality attributes of its component parts no more than the attributes of a molecule can be characterized by the attributes of its constituent atoms. Since high quality components do not equate to a high quality system in any field of engineering, code quality, although necessary, is not sufficient to ensure high quality applications. Organizations need the help of application quality diagnostic tools which can discover inter-component issues and measure the internal quality of the application across its tiers.

There are numerous commercial, freeware, and open source tools available that measure code quality specific to a programming language and are often integrated into Integrated Development Environments (IDEs). These tools are becoming standard components of every developer's toolset since they provide quick feedback during the coding and unit test process. However, these tools are not sufficient to address application quality since they cannot evaluate interactions across the various languages, technologies, and tiers of an application.

Technology that measures application quality analyzes the integrated software produced by a build once the code is checked into a central repository by all the developers. In addition to analyzing each component, application quality technology analyzes their interactions for the types of problems described in earlier sections. Moreover, application quality trends can be compared across builds or releases to monitor the progress against application quality objectives and evaluate the risks posed by the application.

Application quality measurement tools provide several benefits for both the development team and management:

- ***Visibility across application(s):*** Consistent and continuous analysis of all core business applications provides executives with the metrics and information needed to better manage their portfolio of applications and projects.
- ***Analysis of the internal quality of an application***: Reviewing the integrated software system for quality in order to detect architectural and structural problems that hide in interactions between tiers, provides application or project managers with continual status about application quality and risk.
- ***Team performance:*** Since a detailed knowledge of the whole system is usually beyond any individual developer's capabilities, analyzing application quality helps improves developer skills, the team's breadth of application knowledge, and the efficiency of team performance.

A dynamic business environment, new technology, and multiple sourcing options, amplify the complexity of business application software. Since even the most talented developers can no longer know all the nuances of all the different languages, technologies, and tiers in an application, their capability needs to be augmented by automated tools to evaluate the entire application. Without such assistance, defects hidden in the interactions between application tiers will place the business at risk for the outages, degraded service, security breaches, and corrupted data that are caused by poor quality applications.

3.1. Potential Points of Failures: Critical rules

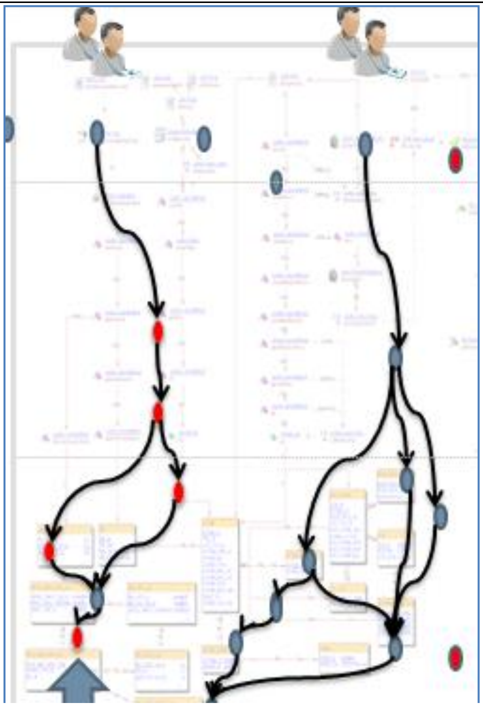
The CAST AIP quality model automatically assesses the application and identifies key issues in the application through a weighted aggregation of more than 1,000 rules across different technologies. The list below shows the various rules where a violation which can create abnornal behavior during the execution of the application has been identified.

TOP 10 CRITICAL VIOLATIONS

Rule Name	# Violations
Close the outermost stream ASAP	569
Avoid transactions with database resource open capability but without the associated close capability	332
Avoid transactions with too many severe Programming Practices - Error and Exception Handling issues along the path	272
Avoid using Fields (non static final) from other Classes	193
Avoid empty catch blocks	170
Close database resources ASAP	155
CWE-134: Avoid uncontrolled format string	99
CWE-89: Avoid SQL injection vulnerabilities	68
Never exit a finally block with a return, break, continue, or throw	59
CWE-73: Avoid file path manipulation vulnerabilities	58

3.2. Potential Points of Failures: Transaction wide Risk Index

Transaction wide Risk Index (TwRI) enables easy identification of the riskiset transactions within the application

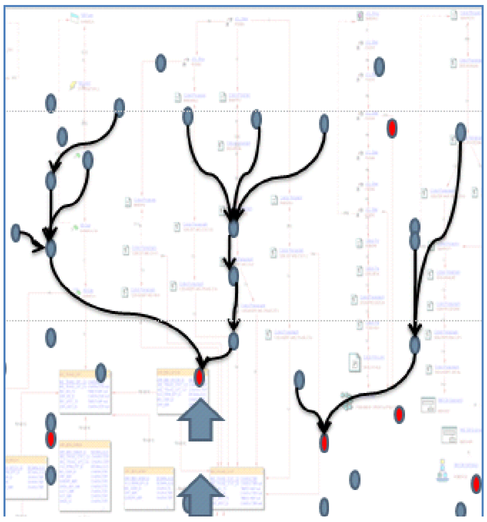


Transaction wide Risk Index (TwRI) is an indicator of the riskiest transactions of the application. The TwRI number reflects the cumulative risk of the transaction based on the risk in the individual objects contributing to the transaction; in the below list the focus is on the efficiency of the application. The TwRI is calculated as a function of the rules violated, their weight/criticality, and the frequency of the violation across all objects in the path of the transaction. TwRI is a powerful metric to identify, prioritize and ultimately remediate riskiest transactions and their objects.

Transaction Entry Point	TRI
org.owasp.benchmark.testcode.BenchmarkTest00941	280
org.owasp.benchmark.testcode.BenchmarkTest00332	278
org.owasp.benchmark.testcode.BenchmarkTest00334	278
org.owasp.benchmark.testcode.BenchmarkTest00592	278
org.owasp.benchmark.testcode.BenchmarkTest00674	278
org.owasp.benchmark.testcode.BenchmarkTest01961	278
org.owasp.benchmark.testcode.BenchmarkTest02633	278
org.owasp.benchmark.testcode.BenchmarkTest02730	278
org.owasp.benchmark.testcode.BenchmarkTest00305	268
org.owasp.benchmark.testcode.BenchmarkTest00411	268
org.owasp.benchmark.testcode.BenchmarkTest00559	268
org.owasp.benchmark.testcode.BenchmarkTest00569	268
org.owasp.benchmark.testcode.BenchmarkTest01941	268
org.owasp.benchmark.testcode.BenchmarkTest02148	268

3.3. Potential Point of Failures: Propagated Risk Index

Propagated Risk Index (PRI) enables easy identification of the riskiset objects/artifacts within the application



Propagated Risk Index (PRI) is a measure of the riskiest artifacts or objects of the application along the Health Factors of Robustness, Performance and Security.

PRI takes into account the intrinsic risk of the component coupled with the level of use of the given object in the transaction. It systematically helps aggregate risk of the application in a relative manner allowing for identification, prioritization, and ultimately remediation of the riskiest objects.

The PRI number reflects the cummulative risk of the object based on its relationships and interdependencies. The PRI is calculated as a function of the rules violated, their weight/criticality, and the frequency of the violation.

The Top 15 objects with the highest PRI are:

Artefact name	PRI
Artefact one	PRI value 1
Artefact two	PRI value 2

4. Measures of Security

Most security vulnerabilities result from poor coding and architectural practices such as SQL injection or cross-site scripting. These are well documented in lists maintained by CWE http://cwe.mitre.org/, and CERT.

4.1. Top 25 CWE Rules

List of the Top 25 CWE rules that had any findings in this application

Metrics	Total Violations	Added Violations	Removed Violations
CWE-79: Avoid cross-site scripting DOM vulnerabilities	541	20	98
CWE-78: Avoid OS command injection vulnerabilities	115	19	56
CWE-89: Avoid SQL injection vulnerabilities	233	27	25
CWE-91: Avoid XPath injection vulnerabilities	10	8	6
CWE-73: Avoid file path manipulation vulnerabilities	137	11	32
CWE-117: Avoid Log forging vulnerabilities	45	9	13
CWE-134: Avoid uncontrolled format string	99	17	22
CWE-90: Avoid LDAP injection vulnerabilities	14	6	4

4.2. Technical Criteria

Technical criterion name	Grade	Evolution
Architecture - Multi-Layers and Data Access	1.00	0 %
Architecture - Object-level Dependencies	3.25	0 %
Architecture - OS and Platform Independence	3.63	0 %
Architecture - Reuse	2.42	0 %
Complexity - Algorithmic and Control Structure Complexity	3.98	0 %
Complexity - Dynamic Instantiation	4.00	0 %
Complexity - OO Inheritance and Polymorphism	3.89	0 %
Complexity - SQL Queries	3.67	0 %
Complexity - Technical Complexity	3.00	0 %
Dead code (static)	3.29	0 %
Programming Practices - Error and Exception Handling	1.00	0 %
Programming Practices - OO Inheritance and Polymorphism	4.00	0 %
Programming Practices - Structuredness	2.20	0 %
Programming Practices - Unexpected Behavior	3.63	0 %
Secure Coding - Time and State	4.00	0 %
Volume - Number of Components	3.62	0 %

4.3. Top Non Critical Violations

Rule Name	# Violations
Avoid Artifacts with High Fan-Out	3,341
Avoid unreferenced Classes	2,744
Avoid Classes with High Coupling Between Objects	2,703
Avoid Too Many Copy Pasted Artifacts	2,482
Avoid declaring throwing an exception and not throwing it	2,307

Rule Name	# Violations
Avoid Artifacts with High Fan-Out	3,341
Avoid unreferenced Classes	2,744
Avoid Classes with High Coupling Between Objects	2,703
Avoid Too Many Copy Pasted Artifacts	2,482
Avoid declaring throwing an exception and not throwing it	2,307

5. Appendix: Understanding Quality Indicators, Quality Rules

CAST AIP has 1000+ quality rules and each rule produces a grade. Depending on the impact, the grades are aggregated into high level Indicators: **Quality Indicators** and **Best Practices Indicators**.

Each aggregation is a weighted average of the contributing metric grades where certain metric grades are flagged critical, i.e. it is nearly a defect. We label these **Critical Violations**.

Quality Indicators

The structure, classification, and terminology are from the ISO 9126-3 specification and the subsequent ISO 25000:2005 quality model. The main focus is on internal structural quality. Subcategories have been created to handle specific areas like business application architecture and technical characteristics such as data access and manipulation or the notion of transactions. The dependence tree between software quality characteristics and their measurable attributes is represented in the following diagram, where each of the 5 characteristics that matter for the user or owner of the business system depends on measurable attributes: Application Architecture Practices, Coding Practices, Application Complexity, Documentation, Portability, and Technical & Functional Volume.

Quality Indicator	Description
Security	A measure of the likelihood of potential security breaches due to poor coding and architectural practices. This quantifies the risk of encountering critical vulnerabilities that damage the business and provides a list of prevention measures.
TQI	A Total Quality Index (TQI) is computed on all the measures made by CAST AIP.

Best Practices Indicators

Best Practice	Description
Programming Practices	Measures the level of compliance of the application to coding best practices. Compliance to best practices reduces risks of failures in production and improves productivity through increased readability and reduced debugging.
Architectural Design	Measures the level of compliance of the application to software architecture and design rules. Compliance to architecture rules improves productivity through better use of existing frameworks and code as well as reduced debugging.
Documentation	Measures the level of compliance of the application to code documentation best practices. Compliance to documentation best practices improves productivity through increased readability and faster understanding of source code.

The risk level is assessed according to the below scale:

Scale	Risk Level
4	🟩 Low Risk
3	🟨 Moderate Risk
2	🟧 High Risk
1	🟥 Very High Risk

6. Appendix: Importance of measuring all layers of an application

Measuring the technical quality of business software applications is evolving from an art to a science with the availability of software tools that automate the process of code analysis. However, it is critical to understand that there are two categories of software quality with very different implications for operational performance. The first category is Code Quality which measures individual or small collections of coded components written in a single language and occupying a single tier (e.g., user interface, logic, or data) in an application. The second category, Application Quality, analyzes the software across all of the application's languages, tiers, and technologies to measure how well all an application's components come together to create its operational performance and overall maintainability.

Although the code quality of individual components is important, by itself it will not ensure the overall quality of the application. Quality is not an intrinsic property of code: the exact same piece of code can be excellent in quality or highly dangerous depending on the context in which it operates. Ignoring the larger context in which the code operates – the multitude of connections with other code, databases, middleware, and APIs – will often generate a large number of false positives.

Today's business applications are complex, built in multiple languages on multiple technologies. Even more challenging, these applications usually interact with other applications built on different technologies. Analyzing the quality of modern applications is monstrously complex and can only be accomplished with automated software that analyzes the inner structure of all components and evaluates their interactions in the context of the entire business application.

Typical application quality problems are listed below to clarify the distinction between application and code quality. Performance testing alone is not sufficient to detect these application quality problems.

6.1. Bypassing the Architecture

Components in one tier of a multi-tier application are typically designed to access components in another tier only through an intermediate "traffic management" component. Bypassing this traffic management component will usually result in a cascade of problems.

6.2. Failure to Control Processing Volumes

Applications can behave erratically when they fail to control the amount of data or processing they allow. This problem is often caused by a failure to incorporate controls in each of several different architectural tiers.

6.3. Application Resource Imbalances

When database resources in a connection pool are mismatched with the number of request threads from an application, resource contention will block the threads until a resource becomes available. This ties up CPU resources with the waiting threads and slows application response times.

6.4. Security Weaknesses

Applications are vulnerable to security attacks when they lack appropriate sanitization checks on user inputs in all relevant tiers of the application.

6.5. Lack of Defensive Mechanisms

Since the developers implementing one tier cannot anticipate every situation, they must implement defensive code that sustains the application’s performance in the face of stresses or failures affecting other tiers. Tiers that lack these defensive structures are fragile because they fail to protect themselves from problems in their interaction with other tiers. Each of these application quality problems will result in unpredictable application performance, business disruption, data corruption, and make it difficult to alter the application in response to pressing business needs. Reliably detecting these problems requires an analysis of each application component in the context of the entire application as a whole – an evaluation of application rather than code quality.