B.U.B.O.L.O. Software Quality Assurance Plan

BU CS673 Clone Productions
Approved February 1, 2014
Revisions Approved February 8, 2014

1. Purpose

This Software Quality Assurance Plan describes the processes and procedures that will be used by the CS673 B.U.B.O.L.O team ("Clone Productions") to ensure that the B.U.B.O.L.O software product is developed on time and meets all requirements. It covers the full development lifecycle, from initial planning, through release, and into the initial support phase. This plan was accepted by the full team on February 1, 2014.

2. Reference Documents

- Software Configuration Management Plan (SCMP)

3. Management

3.1 Organization

B.U.B.O.L.O. is organized and managed using integrated product teams (IPTs) and cross product teams (CPTs). IPTs are multidisciplinary groups of people who are collectively responsible for delivering a defined product or process. CPTs are groups of people with different functional expertise working toward a common goal.

To work more efficiently, the team has been divided into three subteams:

- Alpha Team -- Menu & UI
    - Team Lead: Debbie (Initially led by Abhinav, who dropped the class after the first week of Sprint 1)
    - Asif
    - Secondary Resource: Chris (added after Abhinav dropped the class)

- Bravo Team -- Game Model (Game World Data & Structure)
    - Team Lead: Aaron
    - Patrick
    - Brandon

- Charlie Team -- Graphics

- Team Lead: Chris
- Aditi
- Ethan
- Chennan

As we complete subsystems and transition to the next set of subsystems, such as networking & audio, subteams may change based on each engineer's knowledge and interests. All engineers are expected to have an understanding of all subsystems of the application, regardless of which specific subsystem his or her team is tasked with completing.

3.2 Tasks

3.2.1 Development Life cycle

B.U.B.O.L.O. involves the complete software development life cycle: requirements, design, development, code, unit test, integration, system-level test, release and maintenance.

3.2.2 Development Tasks

Development tasks include those activities associated with a typical software development life cycle.

Activity	Entry Criteria	Exit Criteria
Requirements	Planning Document	Requirements Peer Review, Action Item Work-off, Authorization to proceed
Design	Requirement Concept, Previous stag authorization to proceed	Design Peer Review, Action Item Work-off, Authorization to proceed
Implementation	Design Completion, Previous stage authorization to proceed	Code Peer Review, Action Item Work-off, Authorization to proceed
Testing	Code merged, Test environment established, Test procedures complete	Test Complete
Maintenance	Product released, Bugs need to fix	Update Complete, Bug fixed

Because we are following an agile methodology, the Requirements, Design, Implementation and Testing phases will typically occur in every sprint.

3.3 Roles and Responsibilities

Responsibilities of the Project Manager:

- Act as the project facilitator
- Ensure that work on the project meets the team's schedule
- Organize project meetings and provide an agenda
- Perform final document reviews
- Ensure that resources, such as training, are available so that each group can function effectively

Responsibilities of the Quality Assurance Manager:

- Ensure that all artifacts meet the standards specified in this and other documents
- Provide final review of all code and other assets that are checked into the version control system's master branch
- Ensure that overall project quality is maintained

Responsibilities of the Configuration Manager:

- Please see the Software Configuration Management Plan

Responsibilities of the Team Leads:

- Act as a facilitator among team members on his or her subteam
- Ensure that all artifacts produced by the subteam meet the standards set by the team
- Provide assistance to subteam members
- Organize the work of the subteam

Responsibilities of all Engineers:

- Follow the standards and guidelines that the team has specified in this and other documents
- Read and understand all documents
- Understand all aspects of the application, including areas that the engineer is not directly involved with
- Perform research and suggestions, beyond what is specifically requested
- Perform periodic audits of artifacts, particularly code reviews. Project quality is ultimately the responsibility of all members of the team.
- Volunteer and provide assistance to other team members when requests for assistance are sent

3.4 Quality Assurance Estimated Resources

- One engineer (the Quality Assurance Manager) devoting 3-5 hours per week for quality assurance reviews
- One engineer (the Project Manager) devoting 1-3 hours per per week for final artifact reviews
- Three engineers (the Team Leads) devoting 3-5 hours per week for team artifact reviews
- All engineers devoting 2-4 hours per week for artifact audits and reviews

These estimates are averaged across the project's development cycle, and may be modified as the project and engineers' abilities are better understood.

4. Documentation

4.1 Purpose

This section indicates the documents that will be created during the development, verification, validation, use and maintenance of the game software.

4.2 Minimum Documentation Requirements

The minimum documents necessary are listed below.

- Software Quality Assurance Plan (SQAP, which is this document)
- Software Configuration Management Plan (SCMP)

4.3 Other

The following documentation is covered in other parts of SQAP. Separate documents need not be maintained.

- Maintenance Plan: Covered in section 11
- Software Test Documentation (STD): Covered in section 7
- Software Project Management Plan (SPMP): Covered in section 3

5. Standards, Practices, Conventions, and Metrics

5.1 Purpose

The quality requirements and metrics described in section 5.2, as well as the tools and methodologies described in section 9, are designed to ensure that defects are caught at the earliest possible time, and that the project is completed on time, meets all requirements, and is maintainable. Any member of the team can identify an artifact for nonconformance with this document's quality standards, regardless of who created the artifact. Formally, the initial conformance responsibility lies with the artifact's creator, secondary responsibility lies with the reviewer (typically a Team Leader), and final responsibility lies with the Quality Assurance Manager.

5.2 Content

The following technical, design and programming standards will be used:

1. Documentation Standards: Please see section 4, Documentation, for a description of the documentation that will be required. As a general guideline, if something should be documented, but there is no corresponding official document described in section 4, engineers should add the information to the project's GitHub wiki.

2. Design Standards: The design of each subsystem must be described in detail on the project's GitHub wiki. The description should use a mix of prose, Unified Modeling Language, and code. User stories

should drive requirements as well as design. Large design decisions must be discussed by the team.

3. Code Standards: The Code Conventions for the Java Programming Language, described at http://www.oracle.com/technetwork/java/codeconv-138413.html, will be followed. A standard code convention can ease maintenance and integration, by reducing mental friction. The team has made the following exceptions to the Oracle conventions:
    - Exception to 6.4 and 7.4-7.9: Use symmetrical brace style (also known as "Allman Style"). See Appendix C for more information.
    - Exception to 6.3: Declare local variables where they are first used, *not* at the top of blocks

   Additionally, the guidelines and best practices described by Joshua Bloch in *Effective Java (2nd Edition)* should be followed, since they represent years of research on the Java language. Our team benefits when we learn from the work of others.

4. Comment Standards: All classes and non-overridden, non-private methods, including public, protected and package-private, will be documented using Javadoc comments, as described at http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html and http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html. This standard will ensure that the internal API of the program is fully documented. Additionally, while code should be written in a way that is as obvious and self-documenting as possible, comments should be used to describe complex or long algorithms, or anything else that the engineer or reviewer believes may not be fully understood by an outside party in the future. When in doubt, comment.

5. Testing Standards & Practices: See section 7, Test

6. Software Quality Assurance Product & Process Metrics: The following metrics will be used to determine whether the product meets its quality goals:
    - No known Critical defects remain
    - All artifacts are delivered on time

6. Software Reviews

6.1 Purpose

Standards and procedures for Reviews and Audits are described in sections 3.3 and 8. In addition to formal reviews, all engineers are expected to periodically perform code reviews.

6.2 Minimum Requirements

6.2.1 Software Requirements Reviews

These reviews ensure that project code meets the standards set in sections 5 and 7. Team Leads should review artifacts as they are checked into the subteam's Git branch. Artifacts that will be merged from subteam development branches into the Production branch must be reviewed by the Quality Assurance Manager. If the Quality Assurance Manager is also a Team Lead, one of the other Team Leads should review the Quality

Assurance Manager's code before it is merged into Production. Additionally, engineers will perform periodic audits of the code-base, which should be logged to the project's wiki. The results of these reviews will be sent to the engineer(s) responsible for the artifact, as well as any other interested members of the team.

6.2.2 Architecture Design Reviews

At least once per sprint, the full team will review the architecture design of the application.

7. Test

The testing methodologies that will be followed for the project are as follows:

Each engineer will be responsible for testing his or her code. The Team Leads and the Quality Assurance Manager will verify that test coverage is adequate, and will rerun all tests to ensure that they pass. It consists :

Unit Testing

- The application will be written in Java and therefore JUnit 4 will be used for Unit testing
- In general, one unit test class should be created per new class
- If committing new code to the master branch, the unit tests for that code must be committed at the same time
- If fixing a bug, write a failing test proving the bug, then fix the code to make the test pass

8. Problem Reporting and Corrective Action

Problems will be logged and tracked using GitHub's issue tracker. While this tool does not have as many features as many other issue tracking tools, it is also much simpler and trivial to set up. Whenever possible, our issue tracking process should focus on simplicity and clarity over complexity.

Problems will be classified into three severity levels:

- Critical: The problem is preventing the software from meeting its requirements
- Medium: Neither Critical nor Low
- Low: The problem causes inconvenience or annoyance, but does not prevent the software from meeting its requirements

To help improve the quality assurance process over time, problems will be further divided into categories. In this context, artifact refers to any item that may cause a bug, including code, comments, and documentation:

- Bug: All software bugs that do not meet the definitions for any of the categories listed below
- Performance: A software performance problem
- Integration: The artifact causes failure when combined with existing artifacts
- Omission: A necessary component is missing from the artifact
- Standards Nonconformance: The artifact does not meet the standards set by this and other project

documents
- Unnecessary: The artifact is unnecessary

Because all members of the team have the right and responsibility to periodically audit code, documents and other artifacts that are created over the course of this project, any team member should be comfortable opening an Issue in the tracker for any purpose. Once an issue has been opened, it should be assigned to either the creator of the artifact in question, or a qualified alternative team member. It is the responsibility of the Quality Assurance Manager to ensure that all Critical bugs are resolved before our product ships.

9. Tools, Techniques, and Methodologies

Because identifying and resolving problems earlier in the software engineering process leads to reduced cost, our tools, techniques and methodologies are design to catch problems at the earliest possible moment. The following tools and techniques will be used to help meet our quality requirements:

1. Eclipse Warnings: As the compiler acts similarly to a spell checker, Eclipse warnings act as a grammar checker. These warnings help catch problems very early, at relatively little burden to the engineer, and teach best programming practices. Code must compile with zero warnings, or if this is not possible, a comment above the line(s) of code in question should explain why the warning can be safely ignored. The @SuppressWarnings annotation should not be used to suppress warnings, because this hides potential problems from reviewers. See Appendix A for the specific list of warnings.

2. FindBugs Static Code Analysis Tool: The FindBugs Eclipse plug-in (http://findbugs.sourceforge.net/) is a static code analysis tool that was initially designed by The University of Maryland and has been used by companies such as Google to improve code quality. As with warnings, problems identified by FindBugs should either be resolved or explained; they should never be ignored. Note that FindBugs must be explicitly run, unlike Eclipse Warnings, which are automatically displayed. See Appendix B for the specific FindBugs settings that our team will use.

3. JUnit 4: Units tests should be written using the JUnit 4 test framework, which is included with Eclipse. A unit is defined as the smallest testable piece of code. We require that all non-private methods have tests associated with them, unless an exception is granted by the team. Tests are allowed to have warnings, unlike our production code, because at times it is logical to do things in a test that would not be correct in production code.

Code in the bubolo-prototypes repository of the version control system is not required to follow the same rules as code that is used in the main project. Code here should be self-contained, and should be viewed as throwaway code that expands our knowledge, but not our active code base.

10. Media Control

The Quality Assurance Manager ensures that the procedures and standards as described in the Software Configuration Management Plan are handled properly.

11. Records Collection, Maintenance, and Retention

All documents are computerized, available online, and accessible to all group members. All data will be retained until at least the end of the project. The code, wiki and issue tracker, at a minimum, will be available to the public, since this is an open source project.

All records will be maintained by the Configuration Manager. Please see the Software Configuration Management Plan for additional information regarding records collection, maintenance and retention.

12. Training

The team conducted an initial quality assurance training meeting on February 1, 2014, where these standards were reviewed and accepted. On February 8, the team provided training that covered Eclipse setup, Git and GitHub, importing Eclipse preferences files, and more. Additionally, the team's wiki has been updated to include information on the setup and use of these tools and code conventions, and this information will continue to be expanded. Training issues raised by engineers throughout the life of the project will be addressed on an ad-hoc basis.

13. Risk Management

These general rules shall be followed for the entirety of the project to ensure smooth development process and to minimize the effect of potential problems:

- Should any problems arise, the Project Manager should be notified immediately so the appropriate action may be taken. This extends to anyone on the team being unavailable for any given period of time.
- The team must frequently communicate via weekly in-person meetings and daily online interaction so that everyone understands all aspects of the project
- The scope of the project must be reasonable
- Ensure that the initial design of the system is simplistic and user-friendly. Additional functionality and complexity may be added during the development process but not at the expense of core features.

14. SQAP Change Procedure and History

14.1 Change Procedure

This document is produced by the Software Quality Assurance Plan Group of B.U.B.O.L.O. It will be reviewed by the whole members of the project before release. Thereafter, if there's a necessary changes of this document, it will proposed and discussed in the weekly group meeting.

14.2 Modification History

Date	Revision	Change Overview
2/1/2014	Initial	Creation of SQAP
2/8/2014	A	Made adjustments based on our experience developing the application over the past week. Changed group leader for Menu & UI Team, since Abhinav dropped the class.

Appendix A: Eclipse Warnings

Eclipse warnings are set in the following location: Window >> Preferences >> Java >> Compiler >> Errors/Warnings. At a minimum, the following warnings must be set in Eclipse:

Section: Code Style

- Parameter assignment: Warning
- Resource not managed via try-with-resource: Warning
  Why: Managing resources using the Java 7 try-with-resource block is simpler and less error-prone than using try-finally or other methods.
- Method with a constructor name: Warning
- Method can be static: Warning
  Why: Small scopes lead to smaller problems. Static methods, including private static methods, have reduced scope compared to non-static methods, since no internal state can be accessed by the method. Additionally, declaring a method static is a form of documentation and generally makes it trivially simple to understand what impacts a method will have.

Section: Potential Programming Problems

- Comparing identical values: Warning
- Assignment has no effect: Warning
- Possible accidental boolean assignment: Warning
- Using a char array in string concatenation: Warning
- Inexact type match for vararg arguments: Warning
- Unused allocation: Warning
- Incomplete ‘switch’ cases on enum: Warning
  Note: “Signal even if ‘default’ case exists” should be unchecked.
- ‘switch’ is missing ‘default’ case: Warning
  Explanation: A default case is useful for unexpected input, or to identify situations where the valid input

has changed, but the method has not yet been updated. If there is a valid reason for not having a default case, that is fine: just document it.

- 'switch' case fall-through: Warning
  Explanation: This is more controversial than many other warnings, because there are valid reasons for why a case falls through. However, it is also a common source of accidental bugs. For this reason, if the case fall through is valid, please document it.
- Hidden catch block: Warning
- 'finally' does not complete normally: Warning
- Dead code (e.g. 'if (false)'): Warning
- Resource leak: Warning
- Potential resource leak: Warning
- Serializable class without serialVersionUID: Warning
  Note: this is trivially generated using Eclipse.
- Class overrides 'equals()' but not 'hashCode()': Warning

Section: Name Shadowing and Conflicts

- Field declaration hides another field or variable: Warning
- Local variable declaration hides another field or variable: Warning
  Explanation: This warning can be controversial, because it can conflict with the most natural name for a variable. However, this is another source of real-world bugs.
  Note: "Include constructor or setter method parameters" should be unchecked. This reduces the negative impact of this rule.
- Type parameter hides another type: Warning
- Method does not override package visible method: Warning
- Interface method conflicts with protected 'Object' method: Warning

Section: Deprecated and Restricted API

- Deprecated API: Warning
- Forbidden reference: Error
- Discouraged reference: Warning

Section: Unnecessary Code

- Value of local variable is not used: Warning
- Value of parameter is not used: Warning
  Note: "Ignore in overriding and implementing methods" should be checked.
- Unused type parameter: Warning
- Unused import: Warning
- Unused private member: Warning
- Unnecessary cast or 'instanceof' operation: Warning
- Unused 'break' or 'continue' label: Warning

Section: Generic Types

- Unchecked generic type operation: Warning

- Usage of a raw type: Warning
  Explanation: Generics have been part of Java since 1.5, and provide additional type safety that raw alternatives do not provide. Use the generic version if one is available.
- Generic type parameter declared with a final type bound: Warning

Section: Annotations

- Missing '@Override' annotation: Warning
  Why: The @Override annotation helps identify situations where the overridden method's interface has changed, but the overriding method has not been updated to reflect that change. It provides additional safety at little cost to the engineer.
- Annotation is used as super interface: Warning
- Enable '@SuppresssWarnings' annotations **Unchecked**

Section: Null Analysis

- Null pointer access: Warning

Apendix B: FindBugs Settings

FindBugs settings are set in the following location: Window >> Preferences >> Java >> FindBugs. The following settings should be used:

- Minimum rank to report: 15 (Of Concern)
- Minimum confidence to report: Medium
- Reported (visible) bug categories
  - Dodgy code: True
  - Bad practice: True
  - Correctness: True
  - Performance: True
  - Multithreaded correctness: True
- Mark bugs with … rank as
  - Scariest: Warning
  - Scary: Warning
  - Troubling: Warning
  - Of concern: Warning

Appendix C: Code Style

The team voted on a standard code style on February 1, 2014. This uses the *Code Conventions for the Java Programming Language*, revised April 20, 1999 (http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html), with a modification to the brace style. Instead of K&R style, our team voted to use symmetrical style (also known as

“Allman” style).

Examples are listed below. Note that these do not contain comments, which are also required for all classes and non-private methods. Additionally, this is an example for code style purposes only, and may not reflect the final design of the application.

```
public class Tank extends Entity
{
	private int health;
	private boolean dead;

	public Tank(int x, int y)
	{
		super(x, y);
	}

	public int getHealth()
	{
		return health;
	}

	public void setHealth(int health)
	{
		this.health = health;
		if (this.health <= 0)
		{
			dead = true
		}
	}

	public boolean isDead()
	{
		return dead;
	}

	@Override
	public String toString()
	{
		return “Health: “ + health;
	}
}
```