**Software Design Document**

**for**

Postscript Generator

**Version 2.0**

**Prepared by Max Hesser-Knoll, jesse Zhang, James Lang, and Scott Corcoran**

**UAF CS 372**

**April 15, 2016**

**Table of Contents**

**Table of Contents**

**Revision History**

**1.** **Introduction**

**2.** **System Architecture**

2.1 Application Architecture

2.2 System Overview

2.3 System Diagram

2.4 Assumptions and Dependencies

**3.** **System Design**

3.1 Design Overview

3.2 System Components

**4.** **Testing Strategy**

4.1 Development Testing

4.2 Post Development Testing

**5.** **Appendix**

5.1 Glossary

5.2 References

**Revision History**

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Version** |
| April 15, 2016 | 1.0 | Rough draft for submission. |
| April, 2016 | 2.0 | Final draft for submission. |

# **Introduction**

This documentation is for providing an outline of the program we have created to generate Postscript. Our design implementations will be shown and better understood from it. Documentation here is intended to be used by parties designing and implementing the our Postscript generator.

# **System Architecture**

## **Application Architecture**

The application architecture for the shape generator will be similer to the factor method architrcture. Like an factor method architrcture the generator will be done by creating objects by calling a factory method to easly create and modify new or exsiting objects.

## **System Overview**

This generator is built to have a framework for easy adding of additional features. The primary version of the software features several shapes that can be implemented by the user. These shapes will take a small set of parameters appropriate for the shape itself. THere is no user interface the program will require the api to be used instead.

This program will be implemented using C++.

The advantages of our design choices means that adding additional features will take significantly less work. The starting ground work however will be more time exspensive. With this implemented though future requirements will not be as difficult to meet.

## **System Diagram**

The following diagram displays the system architecture the Postscript generator.

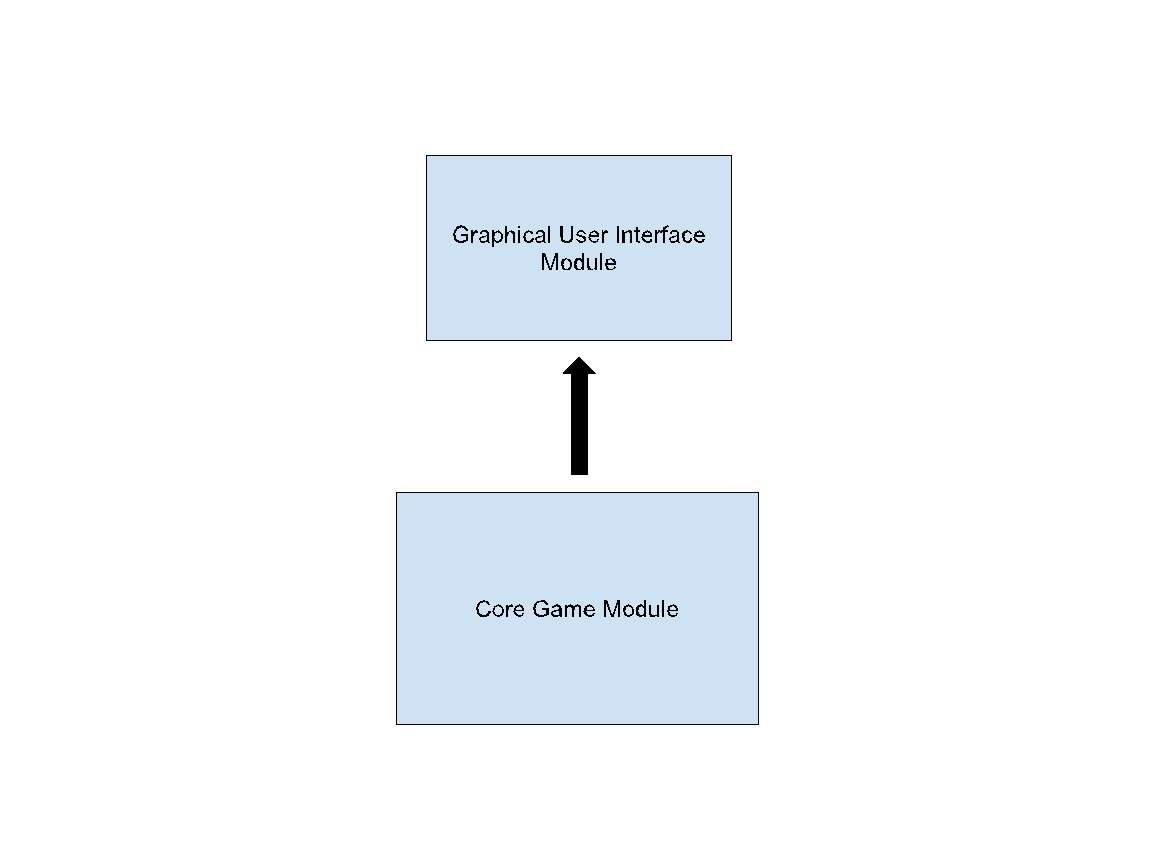


Figure 2.3.1

## **Assumptions and Dependencies**

The Postscript generator is intended to run on a modern x86 Windows, Linux, and Mac OS computers. The generator will output strings, this will require some software or software in combination with hardware to confirm the results.

This program alone will not generate what is needed to print, it will give the strings to be able to.

# **System Design**

## **Design Overview**

This section outlines the basic structure and functionality of the Postscript generator. See Section 2.2 for a high level overview of how the classes work together to form the complete Postscript generator. The functionality outlined for each class in this section is targeted toward satisfying the requirements in the software requirements document for the Postscript generator.

## **System Components**

This system is a C++ API that is able to easily generate Postscript code. The Postscript code that is generated is a series of base shapes that one can then use to create more complex shapes. More shapes can be added due to the structure of our design, with much of the functionalitly already in place.

# **Testing Strategy**

## **Development Testing**

The testing of the system will begin while the system is under development. The basic principles of test driven development will be used at every level of development. Each module, and every component of each module, will be developed concurrently with its testing package. Once each element of a module passes its tests the module can be assembled and tested. The tests that are generated during development can be retained to prevent regression and potentially recycled as elements of other tests.

## **Post Development Testing**

Once the initial development is complete, the post development testing can begin. The post development testing consists of two parts, Basic Functionality Testing and Stress Testing.

### **Basic Functionality Testing**

The purpose of the basic functionality testing is to confirm that the entire system functions properly under normal usage. The testing will mimic the actions of an average user which makes automation of some aspects of the test impractical due to the difficulty of automating a user’s interaction with the system’s user interface. The basic functionality testing strategy can confirm that the development produced a functioning, usable product.

### **Stress Testing**

The purpose of the stress testing to subject the system to unusual datasets in an attempt to determine its limitations or bugs that were not encountered under the normal usage testing in the basic functionality testing stage. Automation can be applied during this stage of testing in order to push the program to its limits. Stress testing can facilitate the location of obscure bugs and helps to determine the performance and usage guidelines for the system as a whole.

# **Appendix**

## **Glossary**

### **Placeholder**: This is a placeholder for future glossary entries.

## **References**

This document uses the IEEE 1016-1998 template as outlined in the following document as a guide:

"Software Design Description." Wikipedia, Web. 20 Nov 2011. <http://en.wikipedia.org/wiki/Software\_Design\_Description>.

The following site provided additional formatting information:

Appleton, Brad. "A Software Design Specification Template." . N.p., n.d. Web. 20 Nov 2011. <http://www.cmcrossroads.com/bradapp/docs/sdd.html>.