14/12/2017 Kyle James Watson STC905

Assignment

Software Architectures

**Table of Contents**

[Executive Summary 1](#_Toc501117737)

[Introduction 1](#_Toc501117738)

[Design Patterns Used 2](#_Toc501117739)

[Other Solutions 4](#_Toc501117740)

[Limitations of the Game 5](#_Toc501117741)

[Bibliography 5](#_Toc501117742)

# **Executive Summary**

The game is based on experiments usually conducted in primary school science classes to teach pupils about electrical circuits and components. Users can create components represented by their symbol and connect them up into a circuit. The system responds to changes in the circuit, powering the components allowing them to change state. Users can also remove connections, move components and remove components and the system will update the state of the circuit.

# **Introduction**

After reading about potential areas of primary school science I landed on creating a game around electrical circuits as I felt I could make an application that met the specification well. I created the assets for the components based off diagrams used for teaching primary school pupils. The class and sequence diagrams were then created to loosely model the core gameplay elements using the design patterns taught in lectures and workshops.

To graphically connect components and show electrical flow I used the *strokeLine()* method to draw wires between them. This allowed wires to be of any length in any direction as the line could extend indefinitely.

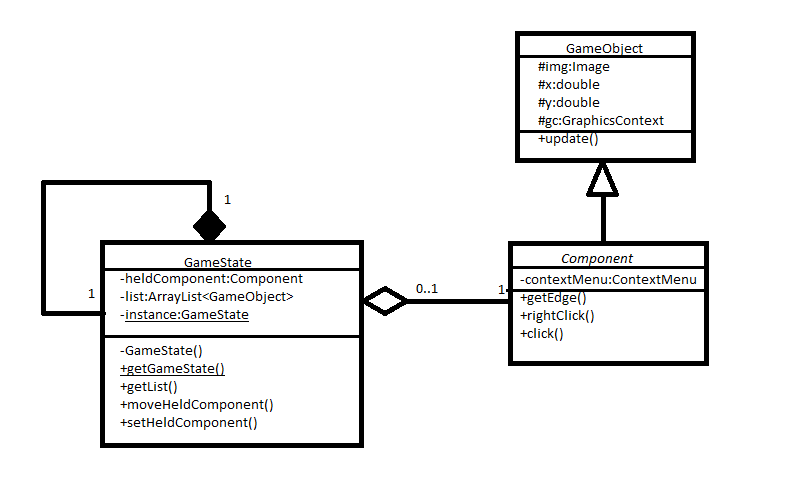
To allow players to modify components after they were placed in the game, a *ContextMenu* object was created in conjunction with a *ContextMenu EventHandler*. This displayed a sub menu after right clicking any component.

A *Slider* in conjunction with a *ChangeListener* was used to allow volume changes. This utilised the *SoundThread setVolume()* method.

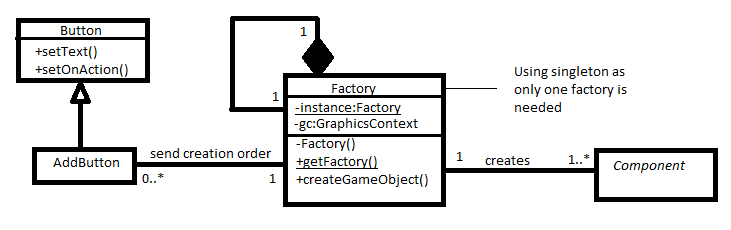
Once the core gameplay was implemented I added *Tutor, Lesson, NullLesson* classes and the *LessonIF* interface to teach pupils basic controls and to get them to experiment with the components.

All images were created by me in paint.net (dotPDN LLC, 2017). The success sound was created by me using Bosca Ceoil (Terry Cavanagh 2015).

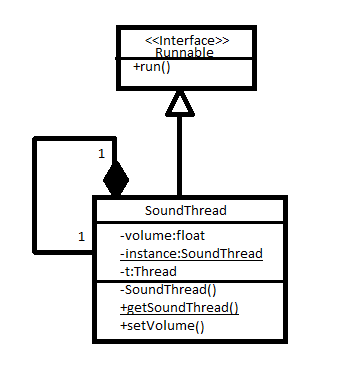
# **Design Patterns Used**

I decided to use a singleton design for 3 different classes where I felt it was only necessary to have one object initialised:

The *GameState* class would need to be accessed by multiple static classes as it is holding the display *list*. Having this as a singleton alows a static instance to be accessed by all the classes which keeps the display *list* synchronous for updates amongst the system.

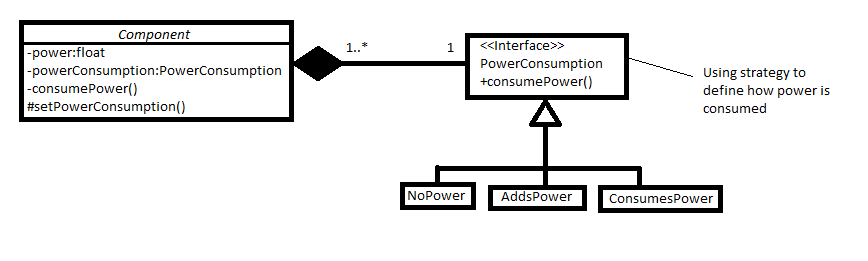


The *Factory* is accessed by multiple *AddButton* objects as they are the handlers used to create *Components*. There is no need to have multiple *Factories* for creating objects as they all would have the same state. The only field is *gc* which would only need to be different game wide if there were multiple *Canvasses*.

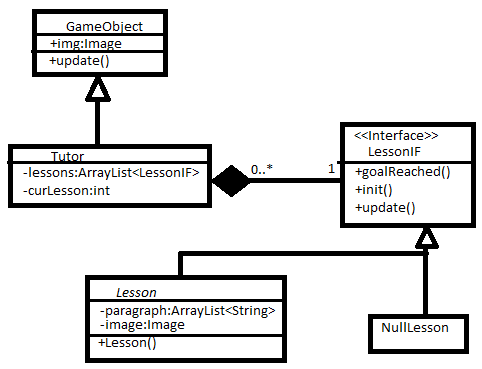
Due to the *Buzzer* being a component in the sandbox I decided a beep generating class would be needed.

After finding a solution from Stack Overflow (tangens 2009), I realised the generation would need to be multi-threaded otherwise the entire update would stutter whilst waiting for the sound to play.

To stop multiple threads playing over eachother and crackling I created the *SoundThread* as a singleton that controlled the looping of the playback.

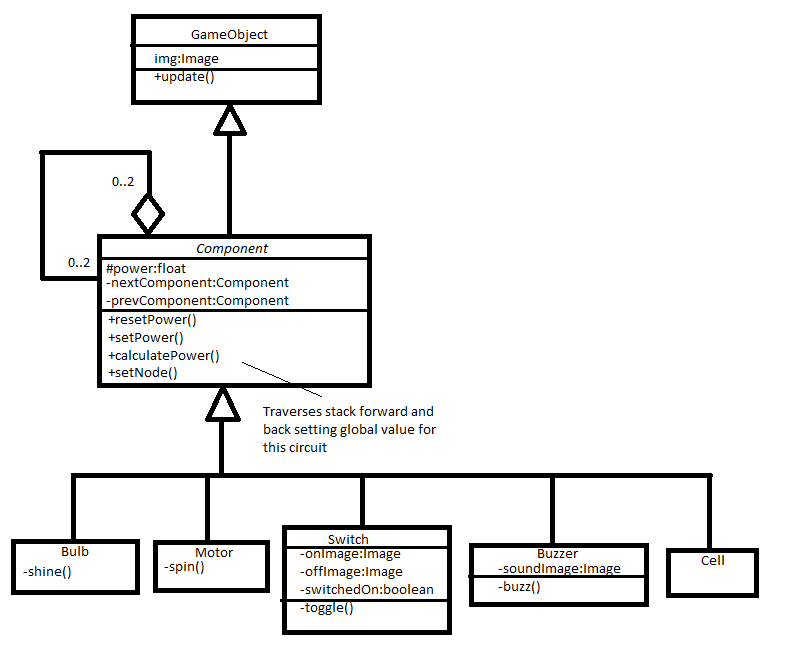


All *Components* in the circuit manipulated the power in one way or another so I decided to use the strategy design pattern to abstract this responsibility.

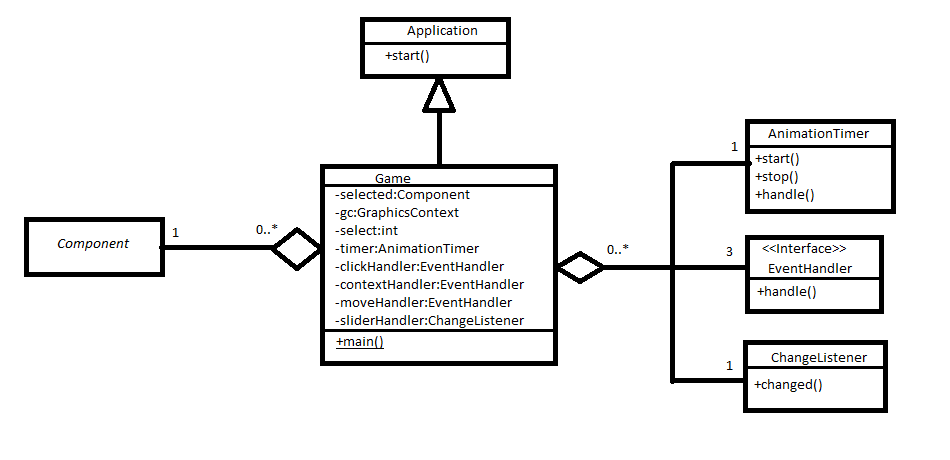
As *Tutor* objects can have an indefinite amount of *Lessons* to loop through it made sense to use the Null Object design pattern so the *Tutor* does not have to check if it is out of *Lessons* each frame.

I added a success sound to play after each lesson using a solution from Stack Overflow (pek 2008).

# **Other Solutions**



Using an algorithm loosely based off a doubly linked list I created recursive functions to calculate, and then set, power for any potential circuits. This allows the *Components* to handle their own circuits resulting in multiple circuits being calculated regardless of their setup.

It was ideal for the *Game* class containing the entry point to the program to set up the UI as well as handle events fired by the UI. Any UI elements that needed to access components could do this through the *GameState* singleton.

# **Limitations of the Game**

The sandbox does not allow for parallel circuits which would be possible with a real life set of components. I felt implementation of this was not needed as, according to STEM Learning Limited (2016), primary school science focusses on how changing elements in a single series circuit can cause other changes for example “the brightness of bulbs, the loudness of buzzers and the on/off position of switches” (STEM 2016, para. 1).

The circuits are not accurately simulated as current, voltage and resistance are not considered. This does not affect the usage of the game as a learning tool because it can still model changes in a circuit not unlike the effects on a real circuit.

# **Bibliography**

dotPDN LLC. (2017). *paint.net.* Retrieved 24th July 2017, from https://www.getpaint.net/index.html

Cavanagh, T (2015). *Bosca Ceoil.* Retrieved 28th August 2017, from <http://boscaceoil.net/>

tangens (2009). Java generating sound. Retrieved 14th December 2017, from https://stackoverflow.com/questions/1932490/java-generating-sound/1932537#1932537

pek (2008). How can I play sound in Java?. Retrieved 8th December 2017, from https://stackoverflow.com/questions/26305/how-can-i-play-sound-in-java

STEM Learning Limited. (2016) Year 6: Electricity. Retrieved 1st December 2017, from https://www.stem.org.uk/resources/community/collection/12390/year-6-electricity