University of Warwick

Department of Computer Science

CS310: Third Year Project

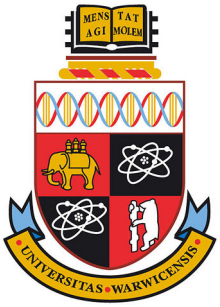
Non-Technical Generation of Object-Oriented Code

Author:
Thomas Allton

Supervisor:
Sara Kalvala

Secondary Supervisor:
Gihan Mudalige

May 25, 2020



Abstract

Object-oriented programming can require a programmer to do repeated work. This type of work is often trivial but time consuming. For example, repeatedly instantiating multiple instances of the same class with different parameters. If the class is used to store data, the parameters are probably determined by some data source. In this case, the programmer must make a decision to either write code to read the data from the source and use it to instantiate the objects during runtime, or copy and paste the data directly into source code and hard code the class instantiation. This kind of work is tedious, and could be done by a non-technical user if automated. A simple intermediate language that contains a representation of class instances, is simple for a non-technical user to understand, and deserialises to full object-oriented code, would solve this problem. This project implements this solution. It takes pure JSON, a simple format of data, parses it using a custom parser, and instantiate classes in an object-oriented language. The objects created are kept ordered in the same order that their corresponding representations had in the file. This allows the objects to be executed in order, forming a kind of program. Additionally, some use cases for the project involving Artificial Intelligence and Game Development are described.

Keywords— Reflection, Java, Compiler, Parser, Representation, Object-Oriented, JSON

Table of Contents

1 Problem Introduction

Automation continues to reduce work for programmers in industry. Software libraries are continually improving, which means programmers can do more with less. However, simple repeated tasks, such as instantiating several data classes, can slow down a developers workflow. Depending on the project, the data used to instantiate these objects can be from some data source, such as a flat file, or the developer could have to seek this data themselves. If the data is stored in some data source, the developer can choose to either read directly from the data source and use it to instantiate classes, or copy and paste the information into code, meaning the data is hard coded. The decision for which approach to choose will probably be made based on the size of the data. If it is the case that the developer must seek and determine the data themselves, this task could also be done by a non-technical user.

The solution solves the problem for both of these cases. For the first case, the solution can be used as an advanced format of data, which when read, results in the instantiation of data classes in an object-oriented language. In the second case, a non-technical user is able to seek and determine the data themselves, and write it in a simple format, which is a representation of objects. These object representations are ordered in a sequence, so they may be executed in order. This is made possible by the classes implementing specific interfaces included in the API, which includes event methods that can be overridden to detect when the class is executed. These types of classes are referred to as 'blocks' to non-technical users, reflecting the ability to combine them together like building blocks, collectively making a program.

An additional use for this project is for it to serve as a bridge between object-oriented languages, and a higher-level representation that is closer to natural language. Rather than the block classes just being classes that store data, they can instead have some useful behaviour. Instantiating these types of classes is therefore similar to calling a function. From the perspective of the non-technical user, this is what they are doing. This concept allows the potential for software libraries to be made more accessible to non-technical users.

In chapter 2 we consider existing methods which partly solve the problem of object deserialisation and examples of high level graphical programming languages. In the next chapter we go through the implementation of the method chosen to solve the problem described above. Tests of this approach are detailed in chapter 4. Chapter 5 describes two example Use Cases which implement this approach. The first is Artificial Intelligence and the second is a tool for game modding. We list possible areas of future development in chapter 6.

2 Background

Several existing high-level and visual programming languages were considered in order to inspire ideas for the project. This also ensured the scope of the project did not include features from existing solutions. This section outlines the main languages considered, and any changes they inspired. Methods to dynamically instantiate objects from data in different languages were also investigated.

2.1 Google Blockly

Blockly is a client-side library for JavaScript for creating block-based visual programming languages and editors. It is used for Android App inventor and Jira AutoBlocks [1]. It represents coding concepts using interlocking blocks, and then generates simple, syntactically correct code in a language you can specify. It is powerful and extendable.

2.1.1 Limitations

Representing coding concepts using drag and drop blocks with configurable properties provides inspiration for class instances to be represented in a similar way. However, Blockly itself does not work well when trying to represent objects in an object-oriented language. It works better for representing code constructs such as conditionals, and variable declaration and modification. The issue with this is it is still fairly difficult to combine these interlocking blocks to create lines of code, and is slower than just writing directly in the language itself, especially in the case of a simple, non-verbose language like Python.

Additionally Blocky does not generate anything that is executable directly, it generates the equivalent lines of code that the blocks represent. The disadvantage of using this for the solution is that it does not produce any intermediate representation of the created program. It directly generates code in the specified language. This means the non-technical user would have to send the source code to a developer, who would then have to manually paste it into some class and ensure it gets run. This would defeat a lot of the purpose of this project, as it is clunky, slows down the workflow, and does not maximise the potential power and usefulness of a non-technical user.

As previously mentioned, this project can be used as a method of reading from large data files, and directly instantiating classes (assuming the data file is in the format of the intermediate language). One major advantage of reading data from a file, rather than hardcoding the data directly in source code, is that it stops source files, and in turn the program as a whole, from getting large. If Blocky is used to generate a lot of source code, it would cause potentially large source files, which is another disadvantage. In summary, Blockly helped realise the idea for representing coding concepts using drag and drop, interlocking, configurable blocks. It also inspired a more user friendly name for this project if it is used in a production environment: 'Blox'. However, the limitations of Blocky detailed above mean that it is not a potential solution to the problem at hand.

2.2 Scratch

Scratch is a block-based visual programming language and website targeted at teaching young people the concepts of basic programming [2]. Like Blockly, it uses drag-and-drop, interlocking blocks that are used together to make a program. The disadvantage of Scratch

for achieving what this project aims to achieve is that it is primarily used to make simple visual games and animations, and can not be hooked into any full object-oriented language in order to produce something more useful. It is not able to produce code in other languages like Blockly, only create graphical behaviour. However, it is the most popular high level language that achieves a somewhat similar goal to what this project aims to achieve: empowering non-technical users to do what could previously be done only by a technical user.

Scratch does have a number of useful blocks that were replicated and implemented in the project. One of these is the repeat block. It is a nested block, meaning you are able to drag blocks inside it. It runs whatever blocks are inside it either a specified number of times, or until a condition is met. A repeat block was implemented to repeat the effect of the succeeding block on a timer. Another useful block that Scratch has is the wait block. It simply waits a specified amount of time before proceeding onto the next block. This block was also implemented using a scheduler.

In Scratch, different types of blocks are colour coordinated. This allows the user to read and write programs faster, as the brain is able to recognise colours more quickly. Although the solution does not implement as many different types of blocks as Scratch, color coding blocks was used in the final solution. Within the Scratch UI, you can run the program, and see it be displayed immediately in a window on the side of the screen. This is useful as it allows fast testing. It would therefore be beneficial to the project to have something like this. Unfortunately, since the project has a broad scope of potential applications, it is impractical to try to implement something like this.

2.3 React

React is a JavaScript library for building user interfaces. It is the most popular user interface framework at the time of writing [8]. React interfaces are built from components, which are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render function [9]. Components can have other components inside them, meaning all components form a tree hierarchy, with a single root component at the top of the tree.

2.3.1 Drag and Drop Components

React has premade frameworks to create drag and drop lists. This makes it a sensible choice for a user interface framework. React has two main variations: ReactJS and React Native. ReactJS is for creation of websites, React Native is for mobile applications. Both use the component structure, however since React Native is not used for websites, it does not use HTML [10]. To make the final product most accessible to users, the interface should be built as a web app, using ReactJS.

2.4 GSON

GSON is a commonly used JSON library made by Google. It allows serialisation of Java objects to JSON, and deserialisation of JSON back to Java objects [4]. The main difference between how objects are instantiated in the solution and GSON is that in GSON, class constructors are not used to instantiate objects. It creates an `ObjectConstructor` with an `UnsafeAllocator` which uses Java reflection to get the `allocateInstance` method of the class `sun.misc.Unsafe` to create the class instance [5]. In simpler terms, GSON uses somewhat of a hack to allocate memory for an object and instantiates it without using any of its constructors.

2.4.1 No Constructor Object Instantiation

Since instantiating a class instance without using a constructor is not supposed to be allowed in Java, it can cause potential issues. Hence the GSON documentation recommends having a default, no-parameter constructor for any objects you wish to serialise. Since the global fields of the object are not set with a constructor, GSON sets them directly using Java reflection. Instantiating objects in this way does avoid dealing with the complexities outlined in the previous section. It also supports nested objects, objects whose properties are other non-primitive objects. Hence GSON recursively serialises the fields of objects till all properties are primitive types that can be stored in JSON.

Because of this, GSON is excellent for serialising and deserialising data objects. However, since class constructors are never invoked, any useful code in the class constructors is never executed. Also, in many cases constructors will call each other, often to convert their parameters to different types. For example, figure 1 shows a `Relationship` class that stores a relationship between two users. It changes the types of the supplied arguments using constructor overloading.

```
class Relationship {
  private final User user1, User user2;

  public Relationship(String userId1, String userId2) {
    this(Database.getUser(userId1), Database.getUser(userId2));
  }

  public Relationship(User user1, User user2) {
    this.user1 = user1;
    this.user2 = user2;
  }
}
```

FIGURE 1: Example class with constructor overloading

It would be useless to use this object in GSON, as it would not be able to invoke the first constructor which takes two string types, and could only create a new instance of the User object, not the one supplied from the data. Also, GSON is only capable of deserialising one object at a time. The solution requires deserialising multiple objects in a list in order to collectively form an executable program, meaning GSON could not be used. For these reasons, GSON is not used in the final solution.

3 Research

3.1 Object Representation

When trying to determine a simpler way of representing class instantiation textually, several formats were considered. The result was that JSON was found to be able to provide a strong representation.

3.1.1 C-based Object Representation

It makes logical sense to start with the syntax of class instantiation in many C-based object-oriented languages.

```
new MyObject(argument1, argument2, ...);
```

The arguments outlined here are to be supplied to the parameters of one of the class' constructors. Now a new language, or intermediate representation, can be built with a new object instantiation on each new line.

```
new MyObject1(argument1, argument2, ...);
new MyObject2(argument1, argument2, ...);
new MyObject3(argument1, argument2, ...);
...
```

Since this representation is fairly simple, a parser would not require the new keyword or semicolon to end a statement, so this can be omitted:

```
MyObject1(argument1, argument2, ...)
MyObject2(argument1, argument2, ...)
MyObject3(argument1, argument2, ...)
...
```

The spaces between arguments can be removed, the opening parentheses replaced with a space, and closing parentheses removed, making:

```
MyObject1 argument1,argument2 ...
MyObject2 argument1,argument2 ...
MyObject3 argument1,argument2 ...
...
```

This is the most reduced form possible for the representation, in terms of least number of characters without compression. However, it does lose some readability, as the parentheses are useful for indicating to the user that the arguments belong to the object.

3.1.2 JSON Object Representation

Using a standard format for storing data is obviously very useful. JavaScript Object Notation, or JSON for short, is an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types [3]. It resembles the representation outlined above. The equivalent representation in JSON would use a key-value pair to store the class name and constructor parameters respectively, with all objects collectively enclosed in a JSON object:

```
{
‘‘MyObject1’’: [argument1,argument2, ...],
‘‘MyObject2’’: [argument1,argument2, ...],
‘‘MyObject3’’: [argument1,argument2, ...]
...
}
```

As this ends up being so similar, it makes logical sense to use JSON as the format for the intermediate representation of programs. As mentioned, the biggest reason for this is that it is a standard format of data, which makes it easier to store and distribute to different forms. JSON can be collapsed to be on a single line, with whitespaces between tokens removed. This form is useful for transmission, as it is more compressed. Alternatively, it can be automatically formatted with whitespaces used as indentation to show the hierarchical structure of the data. This form is easier to read and more user friendly. Representing objects in this way also allows the representation of classes with a single constructor parameter to be simplified so that the square brackets do not need to be included.

```
‘‘MyObject1’’: argument,
```

JSON is composed of values, lists, and objects. The previous two examples show how a JSON list can be used to represent multiple arguments of a class constructor, and a JSON value can be used to represent a single argument of a class constructor. A JSON object can also be used to represent the parameters of an object constructor:

```
‘‘MyObject1’’:{
  ‘‘parameter1’’: argument1,
  ‘‘parameter2’’: argument2
}
```

This more verbose form is useful for when the object constructor has many parameters, and remembering the ordering of these parameters can be difficult. The order that the parameters are specified in this form does not matter. The previous example represents the same object as the following:

```
‘‘MyObject1’’:{
  ‘‘parameter1’’: argument1,
  ‘‘parameter2’’: argument2
}
```

So far, the values of the class' constructor parameters have always been single, primitive types (a string, number or boolean). As JSON structures data hierarchically, it allows for the potential of nested data in object representations. If the class constructor has a single parameter of list of array type, it can be represented as:

```
‘‘MyObject1’’: [ [‘‘a’’, ‘‘b’’, ‘‘c’’] ]
```

Representing constructor parameters that are of the type of another non-primitive object is more difficult. Curly brackets must be used in this case as square brackets would indicate the constructor parameter is a list or array type like in the previous example. For example, if the constructor of MyObject1 has a single parameter of type MyObject2, the representation for MyObject2 can be nested inside the representation for MyObject1:

```
‘‘MyObject1’’: [
  {
  ...
  }
]
```

However this is ambiguous in the following case. If MyObject1 has multiple constructors with parameters of different object types. For example, if MyObject1 has two constructors, one with a single parameter of type MyObject2, and another with a single parameter of type MyObject3, the above example is ambiguous as to which constructor it is attempting to instantiate. Attempting to determine whether the specified parameter is of type MyObject2 or MyObject3 introduces cascading complexity. It would be computationally expensive and messy to implement. Due to this, the solution does not attempt to represent objects in this form. However, if the representation for MyObject1 uses curly brackets rather than square brackets, the representation is able to specify that the parameter is of type MyObject2:

```
‘‘MyObject1’’:{
  ‘‘MyObject2’’:{
  ...
  }
}
```

The limitation of this example is that MyObject2 must be both the name of the parameter of MyObject1, and the name of the object type. For example the actual constructor for MyObject1 in code would look like:

```
public MyObject1(MyObject2 MyObject2) {
  ...
}
```

Implementing constructors this way is not good practise, as it violates the variable naming convention of lower camel case, which many object-oriented languages use. The solution supports case insensitive object type names, meaning object constructors can be written in the following way:

```
public MyObject1(MyObject2 myObject2) {
  ...
}
```

And the representation can be written as:

```
‘‘MyObject1’’:{
‘‘myObject2’’:{
  ...
}
}
```

Due to this, the solution does allow representations in this specific form. This is the extent to which it was found object-oriented style objects could be represented in JSON.

Some representations will contain multiple instances of the same object type. For example,

```
{
‘‘MyObject1’’:[argument1,argument2, ...],
‘‘MyObject1’’:[argument1,argument2, ...]
  ...
}
```

JSON objects do not allow duplicate keys, and many parsers implement them using a map data structure. This means the above representation will not be interpreted correctly. To solve this, the solution must implement its own JSON parser, allowing duplicate keys by implementing objects using a list of entries rather than a map data structure. Hence, the format of the intermediate representation is actually a superset of the JSON format. This is because the parser is able to parse everything a JSON parser can parse, as well as representations a JSON parser can not parse.

3.2 JavaCC

As previously explained, a custom JSON parser was needed to be implemented in order to allow many keys of the same name to exist within a JSON object. Java is the preferred language of choice for the solution due to the reasons mentioned in the previous section. JavaCC (Java Compiler Compiler (a compiler that creates compilers)) is an open-source parser generator and lexical analyzer generator written in Java [7]. It generates a parser from a formal grammar written in EBNF notation. Having a parser generated directly from a grammar is logically going to be more performant than a parser written from scratch. This makes it a sensible choice to use to write a JSON parser in Java.

3.3 Reflection

Reflection is the ability of a process to examine, introspect, and modify its own structure and behaviour [6]. It is possible to be used for object instantiation and meta analysis. The solution requires class' constructors to be evaluated to find which one is most suitable. The most suitable constructor is then used to instantiate a new instance of the class.

Not all object-oriented languages support reflection to this extent. C++ does not support reflection. Python does support reflection, however since Python classes can only have one constructor, the techniques discussed earlier will be less effective. Python also does not require types to be specified, which increases the ambiguity during instantiation. Java and C# both support extensive reflection. It is possible to obtain constructor parameter types, and bypass member access rules, meaning private constructors can be used [6]. Java is used in the solution over C# as it is a more popular language.

3.3.1 Reflective Object Instantiation

Before any coding began on the project, it was important to verify reflection is capable of object instantiation in the ways discussed. Figure 2 shows how reflection is able to instantiate the constructor of a class whose parameters are the same type as the supplied arguments.

```
Object instantiate(String packageName, String className,
Object... arguments) throws Exception {
  Class<?> clazz = Class.forName(packageName + "." + className);
  Class<?>[] suppliedTypes = Stream.of(arguments).map(Object::getClass)
  .toArray(Class<?>[]::new);

  for (Constructor<?> constructor : clazz.getDeclaredConstructors()) {
    if (Arrays.equals(suppliedTypes, constructor.getParameterTypes())) {
      constructor.setAccessible(true);
      return constructor.newInstance(arguments);
    }
  }

  return null;
}
```

Figure 2: Reflective Object Instantiation

4 Implementation

From the background research, it is clear there are a number of existing projects which to some extent achieve a component of the desired solution for this project. Scratch and Blockly enable a non-technical user to drag and drop interlocking blocks to make programs, however these programs are limited in usefulness, and are not compatible with objects in an object-oriented language.

Many JSON parsers exist for different languages, however most do not allow duplicate keys in objects, so can not be used. The GSON library enables object serialisation and deserialisation, but the deserialisation does not use object constructors. This means it is limited for use for data objects only, and therefore can not be used in the solution. Also, GSON supports deserialization only of JSON objects, while the solution supports JSON values and JSON arrays as well. A combination of these ideas is used to realise a solution.

4.1 Parser

Once the non-technical user has created their program, it first must be parsed and stored in data structures in Java. The grammar for JSON was obtained from the official JSON website [3]. This is shown in figure 3.

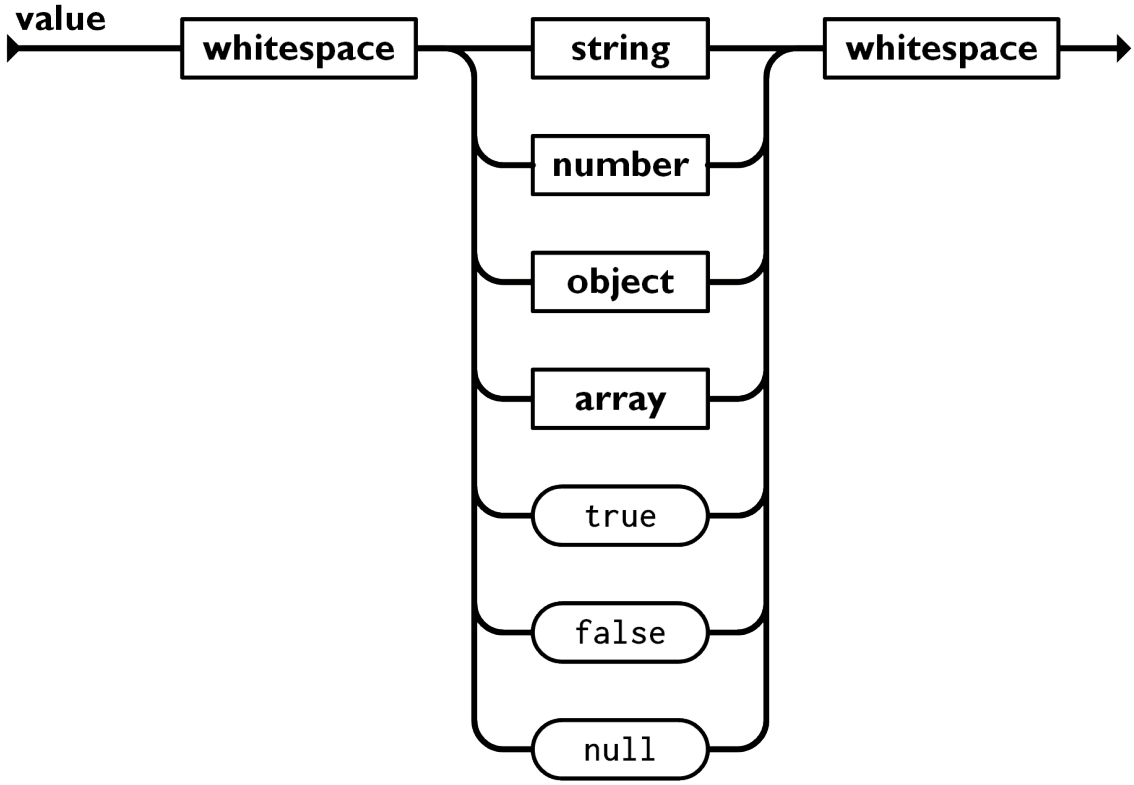


Figure 3: JSON Grammar

4.1.1 Grammar

For the reasons previously discussed, JavaCC was chosen to create the parser. JavaCC accepts a grammar in EBNF form. It has its own file type, with the extension .jj. This file is then used to generate Java source files containing the compiler. First, tokens for quotes, commas, colons, brackets and digits were created. An integer was defined as an optional minus sign, followed by at least one digit:

```
< INTEGER : ("-")? (< DIGIT >)+ >
```

A decimal is defined as an optional minus, followed by any number of digits, followed by a decimal place, followed by at least one digit:

```
< DECIMAL : ("-")? (< DIGIT >)* ‘‘.’’ (< DIGIT >)+ >}
```

A string was defined as a quote, followed by any number of characters that are not quotes, followed by a quote:

```
< STRING : < QUOTE > (~[ "\"" ])* < QUOTE > >
```

The parser was set to skip white spaces, new lines characters, and tabs. The first step was to parse JSON values, which can be a boolean, number, or string. Null values were omitted as non-technical users would not be aware of the concept of null. Boolean values were easily parsed using the true and false tokens. Integers and decimals were parsed using the BigInteger and `BigDecimal` objects respectively. These objects are built into Java, and allow very large numbers, that are above 64 bits, to be stored [17].

It was worth considering whether it was logical to create new data objects specifically for the parser (some parsers have a JsonElement, JsonArray, etc objects). This can make the parser cleaner and more easier for a developer to use as an API. However since this parser is used only internally in the solution to instantiate Java objects, it is not required. It would also add unnecessary overhead to the project, as the Java standard library supports all data structures required. Strings were parsed using the string token, with the opening and closing quotes removed.

4.1.2 List Parsing

The next logical thing to parse was a JSON list or array. JSON lists start with an opening square bracket `[`, have a sequence of JSON elements (can be a JSON value, another JSON list, or JSON object), and end with a closing square bracket `]`. Empty lists are allowed. The ArrayList data structure in Java was used to store the list. It is the most commonly used list data structure in Java, and allows the underlying array to be dynamically resized as elements are added [18]. This is shown in figure 4.

```
List<Object> array() :
{
  List<Object> array = new ArrayList<Object>();
  Object element;
}
{
  < BRACKET_OPEN >
  (
    element = element()
    {
      array.add(element);
    }
    (
      < COMMA >
      element = element()
      {
        array.add(element);
      }
    )*
  )?
  < BRACKET_CLOSE >
  {
    return array;
  }
}
```

FIGURE 4: Array Parsing

4.1.3 Object Parsing

JSON objects are similar to JSON lists, however each element has both a key and corresponding value. As previously discussed, the parser would need to allow duplicate keys in objects. Usually JSON parser implementations use a map or dictionary data structure to store objects, however this can obviously not be used here. Java of course has a HashMap implementation, however two identical keys will generate the same hash, so the second key-value entry would overwrite the first. One solution would be to use both the key of the entry, and its position in the parent JSON object (such as the line number) to generate the hash. Generating hashes in this way allows the map to store multiple entries with keys of the same name, as the hash is generated off both the key and its position in the file.

The Google Guava library implements a `HashBasedTable` [11], which allows a composite key to be generated based off of two keys. However, this data structure does not preserve insertion order, meaning object representations would lose their ordering with respect to each other. Therefore HashBasedTable can not be used in the solution. Also, this would mean any time a lookup was performed on a JSON object to get the corresponding value of a given key, both the key and its position in the file would be needed, which is inconvenient.

A more logical data structure for storing a JSON object is a list of key-value entries. The downside of this is that the lookup for a value of a given key is slower, O(n) in the worst case, instead of O(1) if a hash map was used. However this inefficiency is fairly insignificant, as the data structure will only be used internally. Firstly, it is used to sequentially iterate through the sequence of object representations in the file. And secondly, if the value of an entry is a JSON object, that JSON object will be used to find and instantiate the correct constructor of the Java class.

When this is the case, the number of entries in the object will be the number of parameters in the Java class constructor, which will likely not be a large number. In other

words, for what the parser is used for, there is no case where it will be required to get the value of a given key within a large JSON object, meaning there is not an issue in using this data structure.

The list of entries data structure for an object can be thought of as entries with pointers to the succeeding entry. A linked list stores elements by having each element have a pointer pointing to the next element. This makes it a logical choice to use as a data structure for a JSON object. It also allows the ability to distinguish between whether a list is in fact a JSON list, or JSON object, by checking if it is an instance of ArrayList or LinkedList respectively. JSON objects were parsed similarly to JSON lists, but obviously using key-value entries rather than single elements. This is shown in figure 5.

```
List<Entry<String, Object>> object() :
{
  List<Entry<String, Object>> object =
  new LinkedList<Entry<String, Object>>();
  String key;
  Object value;
}
{
  < BRACE_OPEN >
  (
    key = string() < COLON > value = element()
    {
      object.add(Entry.of(key, value));
    }
    (
      < COMMA >
      key = string() < COLON > value = element()
      {
        object.add(Entry.of(key, value));
      }
    )*
  )?
  < BRACE_CLOSE >
  {
    return object;
  }
}
```

FIGURE 5: Object Parsing

4.1.4 Parser Compilation

At this stage the parser was almost complete. The root of any JSON parse tree is a JSON list or JSON object. The parser was set to start with initially parsing either a JSON list or JSON object, and then recursively parse the remaining structure of the JSON. The Java files for the parser were compiled from the JavaCC file and added to the project.

4.2 Object Instantiation

At this stage, the contents of a JSON file containing the representation of a program can be parsed and stored in data structures in Java. The next stage was to take this data and use it to actually instantiate classes, creating object instances. The first step is to determine what classes this can be done for. As previously discussed, these types of classes are referred

to as 'blocks', to help the understanding of non-technical users. The solution includes a standard library of blocks that can be used universally, such as the `SetAttribute` block, which sets some attribute of the program, such as its name or author. Another example is the `Wait` block, which makes the program wait a specified amount of time before the client is progressed to the next block. While these blocks serve their purpose, the project must have an API to hook into existing software libraries to maximise its usefulness. Hence, the API must allow the ability to add more blocks.

4.2.1 Adding Custom Block Classes

To allow custom block classes to be added, the project is compiled to a jar. Then, other projects are able to use the jar as a dependency, meaning they can add their own block classes. In a Java project, it is not possible to gain a list of all classes in the project [12], even with reflection. If this was the case, it would be possible to simply make every object able to be instantiated. The problem with this, even if it was possible, would be that there are many circumstances where you would not want all classes to be able to be instantiated. For example, if the project has two classes of the same name but in different packages, it would be impossible to determine which class should be instantiated, since the JSON representation specifies only the class name, not the package of the class. Obviously it is possible to register individual classes to be block classes, and the solution does support this. However, this is tedious if there is a large number of classes to be registered. The ability to register all classes in a package to become block classes would solve this. Java classes are organised into packages, corresponding to the folder structure of the project.

4.2.2 Retrieving Classes in a Package

It is not possible to get all classes contained in a package in Java, even with reflection. However since packages correspond to the folder structure of the source code, or of the jar if it's compiled, a workaround is possible. After some research and experimentation, two solutions were devised, depending on whether the code is compiled in a jar, or run directly from an integrated development environment. The first step is to check if the code is running in a jar, by checking if a jar file exists:

```
new File(ClassUtils.class.getProtectionDomain().getCodeSource()
.getLocation().toURI())
```

If this returns true, the internal structure of the jar can be traversed using the `java.util.jar.JarFile` class, and the names of all .class files in a given package can be extracted as shown in figure 6.

```
Set<Class<?>> getClassesFromJar(File jarFile, String packageName) {
  Set<Class<?>> classes = new HashSet<Class<?>>();
  try {
    JarFile file = new JarFile(jarFile);
    for (Enumeration<JarEntry> entry = file.entries();
entry.hasMoreElements();) {
      JarEntry jarEntry = entry.nextElement();
      String name = jarEntry.getName().replace("/", ".");
      if (name.startsWith(packageName) && name.endsWith(".class")) {
        classes.add(Class.forName(name.substring(0, name.length() -
        ".class".length())));
      }
    }
    file.close();
    } catch (Exception exception) {
      exception.printStackTrace();
    }
    return classes;
}
```

FIGURE 6: Retrieval of classes in a package in a jar

The second solution is required if the code is not compiled in a jar. This solution uses the reflections library, which extends the capabilities of standard Java reflection. It allows for runtime metadata analysis, scanning of the classpath, indexing metadata, and allows the ability to query on runtime [13]. At this point, the project needed to depend on this library. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information [14]. The project was converted from a standard Java project to a Maven Java project. By adding the following XML to the project's pom.xml file, the reflections library was automatically compiled into the project jar whenever it was built:

```
<dependency>
  <groupId>org.reflections</groupId>
  <artifactId>reflections</artifactId>
  <version>0.9.12</version>
  <scope>compile</scope>
</dependency>
```

The classes in a package can be retrieved using the reflections library and `ClassLoader` as shown in figure 7.

```
Set<Class<?>> getClassesFromClassLoader(String packageName) {
    List<ClassLoader> classLoadersList = new LinkedList<ClassLoader>();
    classLoadersList.add(ClasspathHelper.contextClassLoader());
    classLoadersList.add(ClasspathHelper.staticClassLoader());

    ConfigurationBuilder config = new ConfigurationBuilder();
    // don't exclude Object.class
    config.setScanners(new SubTypesScanner(false), new ResourcesScanner());
    // filter package
    config.filterInputsBy(new FilterBuilder()
    .include(FilterBuilder.prefix(packageName)));
    config.setUrls(ClasspathHelper.forJavaClassPath());

    return new Reflections(config).getSubTypesOf(Object.class);
}
```

FIGURE 7: Retrieval of classes in a package outside a jar

Initially, the line

```
config.setUrls(ClasspathHelper.forJavaClassPath())
```

was

```
config.setUrls(ClasspathHelper.forClassLoader(classLoadersList
.toArray(new ClassLoader[0])));
```

However, the latter did not work when testing on a macOS due to differences in the Java Virtual Machine and class loader. After brief research, the corrected version was found.

The two solutions outlined above were combined, so that the classes in a package could be retrieved in any circumstance. This allowed the API to have a method to register all classes in a given package as block classes.

4.2.3 Program Loading

Now that the block classes could be determined, the next stage was to instantiate these classes using the JSON representation. Initially, the API required setting a folder containing all JSON files to be loaded, and then a `load()` method called. However, it made more intuitive sense to have a single `load` method which takes a folder as an argument. All files contained in the folder are retrieved recursively using the method shown in figure 8.

```
Set<File> getFiles(File folder) {
  Set<File> files = new HashSet<>();

  if (!folder.isDirectory()) {
    return files;
  }

  for (File file : folder.listFiles()) {
    if (file.isDirectory()) {
      files.addAll(getFiles(file));
    } else {
      files.add(file);
    }
  }

  return files;
}
```

FIGURE 8: Obtaining files in a folder recursively

The files with a `.json` or `.blox` extension are loaded in alphabetical order, which allows non-technical users to ensure a given program gets loaded after any programs it depends on. If the file does not have valid JSON (the parsing fails) or it is not a JSON object, an `IllegalStateException` is thrown. Each key-value entry is parsed and converted into an object instance. The key is used to determine the block class used for instantiation. In order to allow maximum flexibility, the key is converted to lowercase (as block classes are registered using a lowercase key), and spaces are removed. In other words, both:

```
{
‘‘MyObject1’’:argument
}
```

and...

```
{
‘‘my object 1’’:argument
}
```

work and instantiate the same object. This allows for representations to be closer to natural language. For example:

```
{
‘‘Show home screen’’:[]
}
```

Will result in the instantiation of a block class named `ShowHomeScreen`.

Each key-value entry is then used to instantiate a new object using the representations previously discussed. If a class of the given name is not found, an exception is thrown. First, a list of parameters is created. If the value of the entry is a single JSON primitive value (boolean, number or string) and not a collection (JSON list or JSON object), then that single value is extracted and added to the parameter list.

Values are extracted by checking if its class type is a boolean, number or string. Primitive wrapper classes are converted into the equivalent primitive class when the type of a constructor parameter is primitive. For example, if a constructor has a parameter of type `int`, and the extracted value is of type `Integer`, the value will be converted from being an `Integer` type to an `int` type. Additionally, if a constructor parameter type is a decimal

type, such as float or double, and the supplied parameter is an integer, the parameter is converted to a decimal type. This is shown in figure 9.

```
Object castNumber(Class<?> requiredParameter, Object parameter) {
    Class<?> suppliedParameter = parameter.getClass();

    // casts numbers to the specified type
    if (requiredParameter == double.class) {
        if (suppliedParameter == int.class ||
suppliedParameter == Integer.class) {
            return (double) ((int) parameter);
        }
    } else if (requiredParameter == float.class) {
        if (suppliedParameter == int.class ||
suppliedParameter == Integer.class) {
            return (float) ((int) parameter);
        } else if (suppliedParameter == double.class ||
suppliedParameter == Double.class) {
            return (float) ((double) parameter);
        }
    }

    return parameter;
}
```

FIGURE 9: Number Casting

These kinds of operations are performed by the Java compiler itself, which gives the language more flexibility. However, since it is obviously not possible to access elements of the Java compiler from Java runtime, these operations must be replicated at a higher level.

Once the parameters have been collected, a search is performed to find a constructor of the class whose parameter types match the closest to the supplied parameter types. This is done using the algorithm shown in figure 10.

```
Constructor<?> getConstructor(Class<?> clazz, List<Object> parameters,
boolean lenient) {
    search: for (Constructor<?> c : clazz.getConstructors()) {
        int numParameters = c.getParameterTypes().length;

        // don't require array parameter
        if (numParameters > 0 &&
c.getParameterTypes()[c.getParameterTypes().length - 1].isArray()) {
            numParameters--;
        }

        // continue if not enough parameters
        if (parameters.size() < numParameters) {
            continue;
        }

        // check parameter types equal
        for (int i = 0; i < Math.min(c.getParameterTypes().length,
parameters.size()); i++) {
            Class<?> requiredParameter = c.getParameterTypes()[i];

            // fix parameter if it is an array
            boolean array = requiredParameter.isArray() &&
            i == c.getParameterTypes().length - 1;
            requiredParameter = array ? requiredParameter.getComponentType() :
            requiredParameter;

            // if array, check all remaining parameters equal the array type
            for (int j = i; j < (array ? parameters.size() : i + 1); j++) {
                Class<?> suppliedParameter =
                castClass(parameters.get(j).getClass());

                if (!suppliedParameter.equals(requiredParameter)) {

                    // if lenient, try number casting to make parameter types match
                    if (lenient) {
                        Object number = castNumber(requiredParameter,
                        parameters.get(j));
                        suppliedParameter = castClass(number.getClass());

                        if (suppliedParameter.equals(requiredParameter)) {
                            // update parameters with casted number
                            parameters.set(j, number);
                            continue;
                        }
                    }

                    continue search;
                }
            }
        }

        return c;
    }
    return null;
}
```


FIGURE 10: Best constructor retrieval

4.2.4 Representation Flexibility

The algorithm allows for constructors which have an array type for their final parameter to be instantiated. In other words, this allows constructors such as:

```
public DisplayMessage(String... messages) {
...
}
```

to be instantiated using any of the following representations:

```
‘‘DisplayMessage’’: []
```

(with no arguments)

```
‘‘DisplayMessage’’: ‘‘Message’’
```

(with a single argument)

```
‘‘DisplayMessage’’: [ ‘‘Line 1’’, ‘‘Line 2’’ ]
```

(with multiple arguments)

One of the core philosophies decided whilst developing the project was to allow maximum flexibility, and enforce minimal verbosity on the non-technical user. Giving non-technical users this leeway maximises the amount of time they spend on creating programs, as they are not wasting time fixing formatting problems. It also allows the representation to be slightly closer to natural language, which as previously mentioned, is one of the goals of the project.

The lenient parameter of the algorithm is optional, but if specified, will make a greater effort to find the correct parameter type. This is achieved by attempting to cast number types to match the types of the constructor parameters. The solution runs the algorithm once with lenient set to false, to try and find the correct constructor with maximum precision. If the algorithm is unsuccessful, it is run again with lenient set to true. If it is still unsuccessful in finding a constructor, the provided representation does not match the class, and an `IllegalArgumentException` is thrown.

If the constructor found has an array type as its last parameter as mentioned, the arguments are converted accordingly, with the last n arguments converted to a single array argument. This is achieved using the algorithm shown in figure 11.

```
// if last argument of constructor is an array, convert
Class<?>[] paramTypes = constructor.getParameterTypes();
if (paramTypes.length > 0 && paramTypes[paramTypes.length - 1].isArray()) {
    List<Object> typeArray = new ArrayList<>();

    // remove parameters that form an array
    if (parameters.size() >= paramTypes.length) {
        for (int i = paramTypes.length - 1; i < parameters.size(); i++) {
            typeArray.add(parameters.get(i));
        }
        for (int i = parameters.size() - 1; i >= paramTypes.length - 1; i--) {
            parameters.remove(i);
        }
    }

    // create array and add to the parameters
    Class<?> arrayTest = paramTypes[paramTypes.length - 1];

    if (arrayTest.getComponentType().isPrimitive()) {
        parameters.add(createPrimitiveArray(typeArray, arrayTest));
    } else {
        parameters.add(typeArray.toArray((Object[])
Array.newInstance(arrayTest.getComponentType(), typeArray.size())));
    }
}
```

FIGURE 11: Multiple argument to single array conversion

This is represented visually in figure 12.

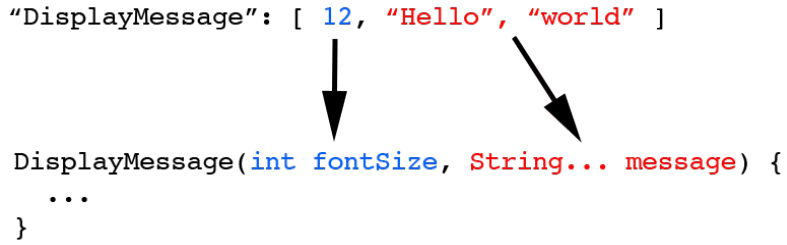


FIGURE 12: Multiple argument to single array conversion

Finally, the constructor is instantiated and added to a list of the objects that have been instantiated from the file.

As previously discussed, a JSON object is stored using a Java `LinkedList`, a list of key-value entries linking to each other. This means the representation can be checked to see if it is a JSON object by simply checking if it is an instance of `LinkedList`. In this case, the correct constructor for the class can be determined more accurately. This is because it is possible to search for a constructor based on the names of its parameters as well as the types of its parameters. The parameter names are checked first, as they are string types, and strings can be compared with less ambiguity and complexity than types. This is done using the algorithm displayed in figure 13.

```
Map<String, Object> object = new HashMap<>();
for (Entry<String, Object> keyValue : (List<Entry<String, Object>>) value) {
    // add parameter name-value pairs to map
    object.putIfAbsent(keyValue.getKey().toLowerCase(),
extractParameter(keyValue.getValue(), true));
}

// search for suitable constructor
constructors: for (Map.Entry<Constructor<?>, List<String>> entry :
blockConstructors.get(blockClass).entrySet()) {
    // check if parameter names equal
    if (!object.keySet().equals(entry.getValue().stream()
    .collect(Collectors.toSet()))) {
        continue;
    }

    // clear parameters from previous attempts
    parameters.clear();

    // check if parameter types equal
    Class<?>[] paramTypes = entry.getKey().getParameterTypes();
    List<String> paramNames = entry.getValue();

    for (int i = 0; i < paramTypes.length; i++) {
        String paramName = paramNames.get(i);
        Object suppliedParam = object.get(paramName);
        Class<?> suppliedClass = castClass(suppliedParam.getClass());

        // nested block
        if (suppliedParam instanceof LinkedList) {
            suppliedParam = getBlock(paramTypes[i], suppliedParam);
        } else {
            // cast number to match type
            suppliedParam = castNumber(paramTypes[i], suppliedParam);
        }

        // check parameter types equal
        if (!suppliedClass.equals(paramTypes[i])) {
            continue constructors;
        }

        // add parameter
        parameters.add(suppliedParam);
    }

    // both parameter names and type equal, save the discovered
    constructor and stop searching
    constructor = entry.getKey();
    break;
}
```

FIGURE 13: Instantiation from JSON object representation

First, the list of entries is converted to a map (since this JSON object will never need to

have keys with the same entries). The keys are converted to lowercase so that they can be compared to the parameters names of the constructor with case insensitivity.

The JSON object keys must now be compared to the parameter names of the block classes' constructors. It is not possible to determine constructor parameter names in Java within the standard Java library, even using reflection [15]. Again, an external library must be used. ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form [16]. It enables the ability to extract constructor parameter names using the algorithm shown in figure 14.

```
List<String> getConstructorParameterNames(Constructor<?> constructor) {
    Class<?> declaringClass = constructor.getDeclaringClass();
    ClassLoader declaringClassLoader = declaringClass.getClassLoader();

    Type declaringType = Type.getType(declaringClass);
    String constructorDescriptor = Type.getConstructorDescriptor(constructor);
    String url = declaringType.getInternalName() + ".class";

    InputStream classFileInputStream = declaringClassLoader.getResourceAsStream(url);
    if (classFileInputStream == null) {
        throw new IllegalArgumentException("The constructor's class loader cannot
        find the bytecode that defined the constructor's class (URL: " + url + ")");
    }

    ClassNode classNode = null;
    try {
        try {
            classNode = new ClassNode();
            ClassReader classReader = new ClassReader(classFileInputStream);
            classReader.accept(classNode, 0);
        } finally {
            classFileInputStream.close();
        }
    } catch (Exception exception) {
        throw new IllegalArgumentException("Error getting constructor parameter
        names for " + constructor.getClass());
    }

    List<MethodNode> methods = classNode.methods;
    for (MethodNode method : methods) {
        if (method.name.equals("<init>") && method.desc.equals(constructorDescriptor)) {
            Type[] argumentTypes = Type.getArgumentTypes(method.desc);
            List<String> parameterNames = new ArrayList<String>(argumentTypes.length);

            List<LocalVariableNode> localVariables = method.localVariables;
            for (int i = 0; i < argumentTypes.length; i++) {
                // the first local variable actually represents the "this" object
                parameterNames.add(localVariables.get(i + 1).name);
            }

            return parameterNames;
        }
    }

    return null;
}
```

FIGURE 14: Retrieval of constructor parameter names

The ASM library was added to the project build in a similar fashion to the reflections library, by adding the following XML to the projects pom.xml:

```
<dependency>
    <groupId>org.ow2.asm</groupId>
```

```
    <artifactId>asm</artifactId>
    <version>7.1</version>
</dependency>
<dependency>
    <groupId>org.ow2.asm</groupId>
    <artifactId>asm-tree</artifactId>
    <version>7.1</version>
</dependency>
```

The parameters names for a given class' constructors are cached for efficiency when the class is registered as a block class. The cache is accessed when trying to find a constructor of a given block class with given parameter names. Once a constructor that has parameter names that match the keys of the JSON object is found, the types of the parameters are compared to the values of the JSON object. If the types match, the constructor is used to instantiate a new instance of the class.

As previously discussed, JSON object representations of Java objects support nested objects. For example, the following case instantiates a new instance of the `MyObject1`, using a constructor with a parameter of type `MyObject2`:

```
‘‘MyObject1’’:{
  ‘‘MyObject2’’:{
  ...
  }
}
```

This is implemented by recursively calling the method to instantiate a block if a value inside a JSON object is another JSON object.

4.3 API

In order to maximise the usefulness of instantiated classes, an API was included to increase the capabilities a developer has to design classes specifically to be used by non-technical users. Once a file is parsed and deserialised to objects, these objects are stored in a list, which is stored within a `Script` object. This object is used to run the program the file represents. The reason for naming the class `Script` rather than `Program` is to reflect the fact that the program will usually be small but flexible, and be dynamically interchangeable in runtime, much like scripts are. However since these programs are compiled, not interpreted, from a technical perspective they are still programs. The API also allows for programs to be loaded and unloaded during runtime.

4.3.1 Block Class Interfaces

The `Block` interface can optionally be implemented by classes registered as block classes. This allows the class to override the methods `onLoad` and `onUnload`, which are called when the program is loaded and unloaded respectively. It also gives the class access to its parent `Script` objects, which can be used to access other blocks in the program. This allows multiple blocks to be used together to provide combined functionality.

When programs are run, they often provide some functionality to the client. When this is the case, the idea is that the client progresses through the program one block at a time, with each block providing some behaviour for the client. Client is referred to here in abstract terms. The definition depends on what the project is being applied to. In the case of some graphical interface, the client will represent the user using the interface. In the case of a game, the client will be a player of the game. When the API is instantiated, the desired class to represent the client is supplied as a generic type.

The `ClientBlock` interface is an extension of the `Block` interface which allows classes to implement these behaviours. The `onEnter` and `onExit` can be overridden to detect when

the client enters and exits the block respectively. After the client has entered the block, it can be progressed to the next block using the `progress` method. The `onEnterScript` and `onExitScript` can be overridden to detect when a client enters and exits the program respectively. In other words, when they enter and exit any block in the program.

Any class can be used as a block class; they do not need to implement any interfaces from the API. However, classes that do not implement any of these interfaces will have limited functionality, and will most likely be used as data classes. Classes that implement `Block` are able to access their parent program, and classes that implement `ClientBlock` are able to provide behaviour to the client when they run the program. Hence there are three main types of block classes: data blocks that have no behaviour, program blocks that have behaviour that affects the program and other blocks within the program, and client blocks that have behaviour that affects the client.

5 Project Management

The project was stored on a private Git repository on a personal GitHub account. Any changes made to the project were frequently committed to the repository, which was on average every half hour. This was beneficial as it allowed there to be a constant backup of the work, despite there being only a single developer working on the project. If any mistakes were made that broke something in the project, the latest commit was reverted.

The Agile development strategy was followed throughout the projects development due to the ambiguity of its scope. It proved beneficial in keeping the project on track, as it enabled the experimentation of different ideas. The project timeline that was followed can be seen in the Gantt chart in Appendix A. Each of these milestones was tracked as an epic on Trello. An epic is a large user story that cannot be delivered as defined within a single iteration or is large enough that it can be split into smaller user stories [20]. Trello is a web-based Kanban-style list-making application [19].

5.1 Testing

Although features were unit tested as they were developed, several tests were conducted in order to ensure the project worked correctly. All API methods were tested, ensuring they behave as expected. Each variation of the different representations previously discussed were also tested to ensure they could be parsed correctly.

5.1.1 API Tests

To use the API, the `Blox` class must first be instantiated. This instance was checked to see what blocks it has by default.

```
System.out.println(new Blox<>().getBlocks());
```

which gave output

```
{name=class com.tomallton.blox.blocks.Name, setattribute=class
com.tomallton.blox.blocks.SetAttribute}
```

which is correct as these are the standard block classes that are added by default. It is possible to not have these default block classes registered by setting `registerDefaultBlocks` to `false` when `Blox` is instantiated:

```
System.out.println(new Blox<>(false).getBlocks());
```

which correctly gave output of an empty map, signifying no block classes are registered:

```
{}
```

The API methods to register new block classes were then tested. When this was tested, a small change to the API was made to make it more convenient to use. The `addBlock` methods were changed to return the instance of the `Blox` class. This allowed multiple blocks to be added in a single line of code as follows:

```
System.out.println(new Blox<>(false).addBlock(Foo.class).addBlock(Doo.class).getBlocks());
```

which as expected showed the two block classes had been successfully registered as follows:

```
{doo=class test.blocks.Doo, foo=class test.blocks.Foo}
```

This showed that individual block classes could be registered successfully. The `Foo` and `Doo` classes both were in the same `test.blocks` package. Hence registration of all classes in a package as block classes was tested as follows:

```
System.out.println(new Blox<>(false).addBlocks("test.blocks").getBlocks());
```

which correctly gave output to show all classes in the `test.blocks` were successfully registered:

```
{doo=class test.blocks.Doo, foo=class test.blocks.Foo}
```

As previously discussed, the API provides interfaces for block classes to implement in order to give them more functionality. These are the `Block` interface and `ClientBlock` class. First, the `BlockTest` class was created to implement `Block`:

```
class BlockTest implements Block {

    public void onLoad(Script<?> script) {
        System.out.println(getClass().getSimpleName() + " block loaded");
    }

    public void onUnload(Script<?> script) {
        System.out.println(getClass().getSimpleName() + " block unloaded");
    }
}
```

And the `ClientTest` class was created to extend `ClientBlock`:

```
public class ClientTest extends ClientBlock<ConsoleClient> {

    public void onEnterScript(ConsoleClient client) {
        System.out.println(client.getName() + " started running program");
    }

    public void onEnter(ConsoleClient client) {
        System.out.println(client.getName() + " entered " +
        getClass().getSimpleName() + " block");

        progress(client);
    }

    public void onExit(ConsoleClient client) {
        System.out.println(client.getName() + " exited " +
        getClass().getSimpleName() + " block");
    }

    public void onExitScript(ConsoleClient client) {
        System.out.println(client.getName() + " stopped running program");
    }
}
```

In order to create a `ClientBlock`, a class representing the client was required. Since the tests were being run from a console, a `ConsoleClient` dummy object was created:

```
public class ConsoleClient {

    public String getName() {
        return "CONSOLE";
    }

    public void message(String... message) {
        for (String a : message) {
            System.out.println(a);
        }
    }
}
```

The `Block` interface was tested by loading, and then unloading, a file with a single `BlockTest` stage:

```
Blox<ConsoleClient> blox = new Blox<>();
blox.addBlocks("test.blocks");

// load program
Script<ConsoleClient> script = blox.load(new File("src/main/resources")).get(0);

// unload program
blox.unloadScript(script);
```

This printed the following output, showing that the `Block` interface functioned correctly:

```
BlockTest block loaded
BlockTest block unloaded
```

Next, the `ClientBlock` class was tested, using the `ClientTest` class. To do this, the file was simply changed to contain a single representation for a `ClientTest` class rather than a `BlockTest` class, and the program was run for a `ConsoleClient`:

```
Blox<ConsoleClient> blox = new Blox<>();
blox.addBlocks("test.blocks");

// load program
Script<ConsoleClient> script = blox.load(new File("src/main/resources")).get(0);

// make client 'run' program
script.enter(new ConsoleClient());

// unload program
blox.unloadScript(script);
```

This produced the following output, as expected, verifying the overridden event methods of `ClientBlock` run as expected:

```
CONSOLE entered ClientTest block
You entered the ClientTest block!
CONSOLE stopped running program
CONSOLE exited ClientTest block
CONSOLE started running program
```

5.1.2 Parsing Tests

Once it was asserted that the API was functioning correctly and block classes could be registered successfully, different representations were created to ensure they could be parsed. Also, some invalid representations were created, to ensure they were detected as invalid and not parsed.

All files that contained invalid JSON caused and error or exception such as

```
Exception in thread "main" com.tomallton.blox.parser.TokenMgrError:
Lexical error at line 1, column 11.  Encountered: "e" (101), after : "t"
at com.tomallton.blox.parser.ParserTokenManager
.getNextToken(ParserTokenManager.java:447)
at com.tomallton.blox.parser.Parser.jj_ntk(Parser.java:320)
```

A class with many constructors was created in order to test different representations. This is shown in figure 15.

```
public class TestBlock {

    public TestBlock(boolean a) {
        System.out.println("Instantiated with boolean constructor: " + a);
    }

    public TestBlock(int a) {
        System.out.println("Instantiated with int constructor: " + a);
    }

    public TestBlock(int... a) {
        System.out.println("Instantiated with int array constructor: " +
        String.join(",", Stream.of(a).map(Object::toString).collect(Collectors.toList())));
    }

    public TestBlock(float a) {
        System.out.println("Instantiated with float constructor: " + a);
    }

    public TestBlock(double a) {
        System.out.println("Instantiated with double constructor: " + a);
    }

    public TestBlock(double[] a) {
        System.out.println("Instantiated with double constructor: " +
        String.join(",", Stream.of(a).map(Object::toString).collect(Collectors.toList())));
    }

    public TestBlock(String a) {
        System.out.println("Instantiated with string constructor: " + a);
    }

    public TestBlock(String... a) {
        System.out.println("Instantiated with string array constructor: " +
        String.join(",", a));
    }

    public TestBlock(List<String> a) {
        System.out.println("Instantiated with string list constructor: " +
        String.join(",", a));
    }
}
```

FIGURE 15: Test block class for testing constructor instantiation

Many instances of this classes were instantiated using the following program

```
{
"TestBlock": false,
"TestBlock": 9,
"TestBlock": [11, 12],
"TestBlock": "Hello",
"TestBlock": [ "Hello", "world" ],
"TestBlock": [[ "Hello", "world" ]]
```

```
}
```

Which produced the expected output:

```
Instantiated with boolean constructor: false
Instantiated with int constructor: 9
Instantiated with int array constructor: [11, 12]
Instantiated with string constructor: Hello
Instantiated with string array constructor: ["Hello", "world"]
Instantiated with string list constructor: Hello,world
```

Instantiation of class constructors with a parameter of a non-primitive type was also tested. The following constructor requires a `Location` type. The `Location` class is registered as a block.

```
public TestBlock(String a, Location location) {
  System.out.println("Instantiated with: " + a + ", (" + location.x + ", "
  + location.y + ", " + location.z + ")");
}
```

The following representation was parsed and deserialised successfully

```
{
"TestBlock": {
"a": "test",
"location": { "x": 1.0, "y": 2.0, "z": 3.0 }
}
}
```

which printed correctly out

```
Instantiated with: test (1.0, 2.0, 3.0)
```

5.1.3 Edge Cases

Some edge cases were tested to ensure the parsing was stable under a variety of conditions. Strings with escape characters were tested:

```
[ ‘‘Hello \’’world’’ ]
```

However this threw a parsing exception, as the quote character is recognised as the end of the string. This was fixed by changing the string grammar from

```
< STRING : < QUOTE > (~[ "\"" ])* < QUOTE > >
```

to

```
< STRING : < QUOTE > ("\\" ~[] | ~["\"", "\\"])* < QUOTE > >
```

which allows escape characters to exist in strings.

Next, representations which contained very large numbers were parsed (larger than a 64 bit `long` or `double`).

```
‘‘TestBlock’’: [ 9999999999999999, 9999999999999999.999999999 ]
```

however these did not pose any problems, as the parser stores numbers in `BigInteger` and `BigDecimal`, meaning numbers of any size can be stored.

Chinese and unicode characters were also parsed successfully. This is because the parser uses UTF-8 encoding.

6 Use Cases

The broad and general scope of this project can make it difficult to understand potential applications. As the project is effectively a library, it is useful to have some example applications to see some of the potential of what it can be used for.

First, a small, lightweight neural network library is created with the backpropagation algorithm implemented. The classes in this library are registered as block classes, giving non-technical users access to the power of Artificial Intelligence. They are able to build their own neural networks, train them using a data set, and then use the model for classification of new data.

Multiple layers of abstraction are provided. At the top level of abstraction, the non-technical user simply provides a dataset, and the neural network model is automatically created, with parameters and hyperparameters automatically determined. The model may then be used for prediction. At the bottom level, the user is able to set individual layers of the neural network, specifying the number of neurons in layers, whether the layer should have a bias, and the activation function of the layer. While it is true this low-level of abstraction is fairly technical, it is still useful for users with mathematical knowledge but little programming experience.

Then, the project is used as a tool to assist in game development. Since it is not worthwhile creating a game from scratch just for a demonstration, an existing game is used and built upon. As the project is written in Java, the chosen game must use the same language. Minecraft is the most popular video game of all time in terms of sales [21], and the original PC edition is written in Java. This makes it a logical choice to use. Several block classes are created to provide content to the client, which in this case is a Minecraft player. These include blocks to set the text of different parts of the player's UI, create in-game entities, and give the player items. A combination of these blocks is used to create in-game challenges.

6.1 Artificial Intelligence

Artificial Intelligence has been one of the greatest technological innovations of the past decade. As previously discussed, this project allows non-technical users greater access to software libraries and tools. The library in question was not originally written for the purpose of being used this way [22]. I created it simply to assert I could implement, and therefore understand the backpropagation algorithm. As hoped, it was very easy to convert to having classes that could be used as blocks.

6.1.1 High Level Abstraction

To create neural network models with the most ease, the non-technical user should use the highest level of abstraction available. This is the `CreateNeuralNetwork` block. It requires the non-technical user to specify only the CSV file containing their data, and the columns of that file that are feature (input) and label (output) columns. Also a name for the model must be specified, so that the model can be referenced and used in other blocks later on in the program, or in blocks in other programs. The following simple example trains a model to detect whether a person has cancer:

```
{
"Create neural network": {
```

```
  "name": "cancer_model",
  "file": "src/main/resources/cancer.csv",
  "featureColumnStart": 2,
  "featureColumnEnd": 32,
  "labelColumn": 1,
  "trainProportion": 0.8
}
}
```

Notice the name used for the block being `Create neural network`, which is intuitive and user-friendly for a user with little technical experience. The CSV file provided is the Breast Cancer Wisconsin (Diagnostic) Data Set [23]. The features for a patient in the dataset are between columns 2-32, and the label or classification of whether they have cancer or not is in column 1. The `trainProportion` parameter is optional, but specifies how much of the data is used to train the model (training set), and how much is used to predict its accuracy (validation set). In this case, it is set to `0.8`, meaning the model will be trained on 80% of the data, and validated on the remaining 20%. If not specified, the default value for `trainProportion` is 70%.

The result of the instantiation of this block is a trained neural network that is capable of making predictions. The fields of the dataset are automatically converted to numerical values if they are strings, using one-hot encoding. The values of the features are normalised, by subtracting each value by the mean value for that feature, and diving by the variance for that feature:

$$z = \frac{x_i - \mu}{\sigma} \tag{6.1}$$

The resulting neural network created by the block has 3 layers, each with their own bias. The input layer of course has a number of neurons equal to the number of features in the dataset (`featureColumnEnd - featureColumnStart = 32 - 2 = 30` in this case). The output layer has a number of neurons equal to the number of of expected labels in this case (`1` neuron in this case, which will output `0` for not having cancer and `1` for having cancer). The middle, hidden layer has a number of neurons halfway between these two values, or the average of the input and output size of the network. This collectively forms a dimensionality reduction in the network. Aside from the output layer, each of the layers use the sigmoid activation function to normalise their output. This is because the sigmoid function is generally best at allowing complex patterns in data to be learned.

The learning rate and epochs for training is locked at 0.1 and 100 respectively. While it is possible to determine these hyperparameters more intelligently, this was omitted as the focus of this project is not Artificial Intelligence.

Once trained, the model prints the classification accuracy on the training set. In this case, the accuracy was consistently over 90% after multiple runs.

```
Trained neural network model, accuracy: 90.4%
```

The model can now be used for classification for other data using the `Predict` block.

```
"Predict": [ "cancer_model", "src/main/resources/cancer2.csv", 2, 32 ]
```

This prints out the prediction vector for the dataset using the `cancer_model` model created.

At the time of writing, the neural network library that has the highest abstraction (in other words, easiest to use, with minimal technical knowledge) is probably Scikit-learn. Scikit-learn is a free machine learning library for Python [24]. However, even Scikit-learn requires the data to be processed, normalised, and training and validation data sets separated for models to be trained. And obviously requires knowledge of Python. Therefore, this solution is arguably the most accessible artificial Intelligence library to non-technical users.

6.1.2 Low Level Abstraction

While the high level abstraction of the neural network library is very easy to use, it offers no flexibility to the user when creating models. The low level abstraction allows users to specify the hyperparameters of the model, such as the number of layers in the network, and the number of neurons, bias and activation function for a given layer. It also allows for the learning rate and training epochs to be specified.

The existing `Layer` class, which represents a layer of a neural network, was registered as a block class. Only one small change was needed. Its main constructor is

```
public Layer(int neurons, boolean bias, ActivationFunction activationFunction) {
```

Since `ActivationFunction` is not a primitive type, this constructor would not be able to be instantiated using a JSON representation. To resolve this, an additional constructor was simply added to overflow to this main constructor, converting a `String` type to an `ActivationFunction` type.

```
public Layer(int neurons, boolean bias, String activationFunction) {
  this(neurons, bias,
  ActivationFunction.valueOf(activationFunction.toUpperCase()));
}
```

This example exemplifies how easy it is to adapt existing software to be parsed and instantiated from a JSON representation. Individual layers can be created as blocks. Then, the `CreateNeuralNetwork` block can be used to create a neural network from these layers. It works by scanning all blocks in the program, and checking if the block is a `Layer`. Once a model is created, it can be trained using the `Train` block. This is put together in the following example:

```
{
"Layer": [ 30, true, "Sigmoid" ],
"Layer": [ 15, true, "Sigmoid" ],
"Layer": [ 1, true ],
"Create neural network": "cancer_model",
"Train": {
  "modelName": "cancer_model",
  "file": "src/main/resources/cancer.csv",
  "featureColumnStart": 2,
  "featureColumnEnd": 32,
  "labelColumn": 1,
  "trainProportion": 0.8
}
}
```

This example results in the creation and training of exactly the same network as the previous example, but this time the hyperparameters of the network are manually specified. This more verbose form allows more flexibility and control over the network. It enables a user to experiment with hyperparameter values by re-running the program and checking the classification accuracy of the trained model. As before, the program outputs the accuracy of the training, which is consistently over 90%.

```
Trained model, accuracy: 93.0 %
```

From here it would be easily possible to create block classes to serialise and deserialise models. This would involve the storing of all of the models parameters (weights, biases, activation functions). This has the advantage of allowing models to be loaded and used without having to be trained each time. However as this would be basic yet quite time consuming to setup, it is not covered here.

6.2 Game Development

Minecraft is the most popular video game of all time in terms of sales [21] and conveniently is written in Java. Extensive modding of both the client and server code has greatly helped its development. Here, we create blocks in the server side code, allowing non-technical users to create behaviour in-game. The Spigot API makes this trivial, as it exposes an API and allows creation of server plugins [25].

6.2.1 Trampoline Program

The Minecraft world is 3D world made from blocks. The world utilises a coordinate system. A block class named `RegionEntry` was created. This takes a 3D cuboid region as input (six numeric coordinates, three for one corner, three for the other corner). It detects when a player enters this region, and then makes them run the program.

Additionally, a block class named `SetPlayerVelocity` was created. As the name suggests, this sets the velocity of the player when they enter the block. These two blocks can be used together to make a trampoline as shown in figure 16.

```
{
"RegionEntry": [ 100, 100, 100, 105, 101, 105 ],
"SetPlayerVelocity": [ 0, 1, 0 ]
}
```

FIGURE 16: Minecraft Trampoline Program

The `RegionEntry` block defined here specifies an area that is 5 Minecraft blocks wide, and 1 Minecraft block tall. This is the area above the trampoline. The `SetPlayerVelocity` block sets the player's velocity vertically upwards, as the y velocity is 1, and the x and z velocity is 0. Together this means each time the player enters the trampoline region, they are bounced vertically upwards. So each bounce of the trampoline is one run of the program.

6.2.2 MMO Style Quest Program

Massively multiplayer online (MMO for short) games often incorporate quests into the game world, where the player must achieve a series of objectives to receive a reward. If block classes could be created to require a player to meet these objectives, and then give them rewards, it would be possible to make it easy for a non-technical user to make quests of this style in Minecraft.

These objectives often involve finding and clicking some entity in the world. It is possible to spawn entities in the Minecraft world using the Spigot API. These can be animals (pigs, cows, etc) or non-player characters (NPCs for short). The `CreateEntity` block was created to allow this. Once an entity is created, the player must be required to click said entity. The `ClickEntity` block was created to do this. Also, a `SendPlayerMessage` block was created to send a message to the player. This allows the entities to talk to the player during interactions. Finally, to allow the player to be rewarded, the `GivePlayerItem` stage was created to add a Minecraft item to the player's inventory. These blocks can be combined together to make an MMO style quest as shown in figure 17.

```
{
"CreateEntity": [ "Alice", "Human", 100, 100, 100 ],
"CreateEntity": [ "Bob", "Human", 200, 200, 200 ],

"ClickEntity": "Alice",
"SendPlayerMessage": "Alice: Hello, I am Alice. Go and find Bob for a reward.",
"ClickEntity": "Bob",
"SendPlayerMessage": "Bob: Hello, I am Bob. Take this diamond as a reward.",
"GivePlayerItem": "Diamond",
"SendPlayerMessage": "Quest complete!"
}
```

FIGURE 17: MMO Style Quest Program

This very simple program creates an MMO style quest or challenge for a Minecraft player. The `CreateEntity` blocks create two human entities named `Alice` and `Bob`, at coordinates `(100, 100, 100)` and `(200, 200, 200)` respectively. They do not effect the rest of the program, which is reflected by the blank space between the `CreateEntity` blocks and the rest of the program. The block `"ClickEntity": "Alice",` refers to the entity previously created in the `CreateEntity` block. The rest of the program is self explanatory. The player must find the entity named Alice, which makes them start the program. Alice tells them to find Bob. The player must then find Bob to receive a diamond as a reward. In many cases, these kinds of programs would take less time to create than the average time it takes for a player to complete them. This is a testament to the power of the project, as it enables a lot of behaviour and content to be created from very little work.

While these examples are small and simple, they can be expanded to be much larger and more useful. An earlier iteration of this project is used on the PokéFind Minecraft Network [26]. It runs hundreds of programs, ranging from small challenges, to MMO style quests that take hours to complete. These have been enjoyed by hundreds of thousands of players, and are all made by individuals with little to no technical knowledge. It has 249 block classes at the time of writing. Hence, it is this that actually inspired the original idea and concept for this project.

7 Future Work

Since this project is a library, it has a huge variety of potential, both as a tool for automation, and a tool to make technical work more accessible to non-technical users. The two example use cases discussed previously demonstrate some of its power, but there are many more potential applications.

7.1 User Interface

A Scratch-like user interface that allows a non-technical user to collate blocks would make it even easier to create programs. The user would simply have to drag and drop blocks rather than typing out their names and structuring the program according to the JSON specification. The interface could have a button to export to a JSON file. This would create a higher level abstraction for the creation of programs. The full flow of program creation would then become `User Interface -> JSON Representation -> Java Objects`.

This would effectively would allow drag and drop object oriented programming, and give non-technical users the power of the object oriented paradigm. A logical design would be to have a window on the left containing all blocks (similar to Scratch). A search function would allow blocks to be located quickly. The program would occupy the rest of the screen. The user would drag blocks into the program, and then set their parameters. It would also be possible to reorder blocks in the program. Programs could also be imported into the interface, where the blocks in the JSON representation of the program would be loaded into the interface. This would enable programs to be edited once created.

7.2 Other Language Support

It is possible to add support for languages other than Java. As previously discussed, Python supports reflection to the same extent as Java, and has great flexibility. This makes it a logical choice for the next object-oriented language to implement JSON object deserialisation. The algorithm to instantiate objects would be quite different, since Python classes can not have multiple constructors. Instead, constructors are able to have optional arguments.

7.3 Improved Error Handling

Exceptions that are thrown when the project has an error parsing JSON or instantiating objects. These exceptions cause the program to halt during runtime. It would be beneficial to change this, as errors caused by non-technical users should be expected. A simple solution to this would be to make sure all exceptions thrown are custom exceptions that extend `RuntimeException`, as runtime exceptions do not halt the program if caught. Another solution would be to print out errors in a form friendly to non-technical users, instead of throwing exceptions.

7.4 Further Use Cases

The two example use cases mentioned could be expanded upon greatly. The artificial intelligence library could implement blocks for more neural network models (convectional

neural networks and recurrent neural networks to name a couple). More flexibility could be added to the training block, to allow the user more control of the hyperparameters used for training. In general, this project works best when applied to anything that involves specifying data (such as training neural networks), or any process that is sequential (such as a quest) in nature. There are many more potential applications aside from the two already mentioned.

7.4.1 User Interface Creation

It would be possible to create blocks to allow the creation of user interfaces. Blocks could be made to create user interface components, and create behaviours when those components are clicked. Swing is a GUI widget toolkit for Java. It is part of Oracle's Java Foundation Classes – an API for providing a graphical user interface for Java programs [27]. It would be possible to extend upon a library like Swing and make it accessible to non-technical users. As user interface creation can be a tedious process of trail and error, allowing it to be done be a non-technical user would be very beneficial.

8 Conclusion

Providing a higher level representation of classes of an object-oriented language in a simple form is useful as it allows the classes to be used by users will little technical knowledge. This representation can be used as a tool for automation, as it automates a task for a developer, allowing it to be done by a non-technical user. It also can be used as a data format, where the data in a flat file is deserialised and results in the direct instantiation of objects. JSON is a sensible format for these representations, as it is a standard format for data, and allows great flexibility. Google's GSON library is a good solution for object serialisation and deserialisation of JSON objects to Java class instances. However it avoids the complexities associated with determining the right class constructor to instantiate. This project takes a JSON representation of a series of objects, and uses it to instantiate those objects. It does this by finding the most suitable constructor, and instantiating that constructor using Java reflection. The order of these objects is preserved, meaning once converted into Java code, they can be used collectively as a program that can be run by the client. The client is referred to here in general terms; the API allows it to be any interface representing a client depending on what the project is being used for. The project's usefulness is demonstrated with two example use cases that are very different from each other: an artificial intelligence library, and a tool for game development. The artificial intelligence library allows non-technical users to train and use complex neural network models. The game development tool allows non-technical users to create behaviour in a Minecraft world, but could be adapted to any 3D environment.

8.1 Author's Assessment of the Project

`What is the (technical) contribution of this project?`
The project implements a JSON parser written in JavaCC. It implements algorithms which use Java reflection to convert a JSON representation of a Java object to the actual object. In order to demonstrate a use case, a small neural network library is implemented. It implements the backpropagation algorithm and gradient descent for training. The project has also been used as a game modding tool for Minecraft. There have been 249 block classes created at the time of writing. The programs created range from small challenges to large MMO style quests that take hours to complete. They have been enjoyed by hundreds of thousands of players.

`Why should this contribution be considered relevant and important for the subject of your degree?`
It is the most powerful and flexible JSON to Java object deserialisation tool out of any open source solutions found. It also explores how the complexities of object oriented programming can be represented at a higher level of abstraction, that is closer to natural language, and therefore more accessible to non-technical users.

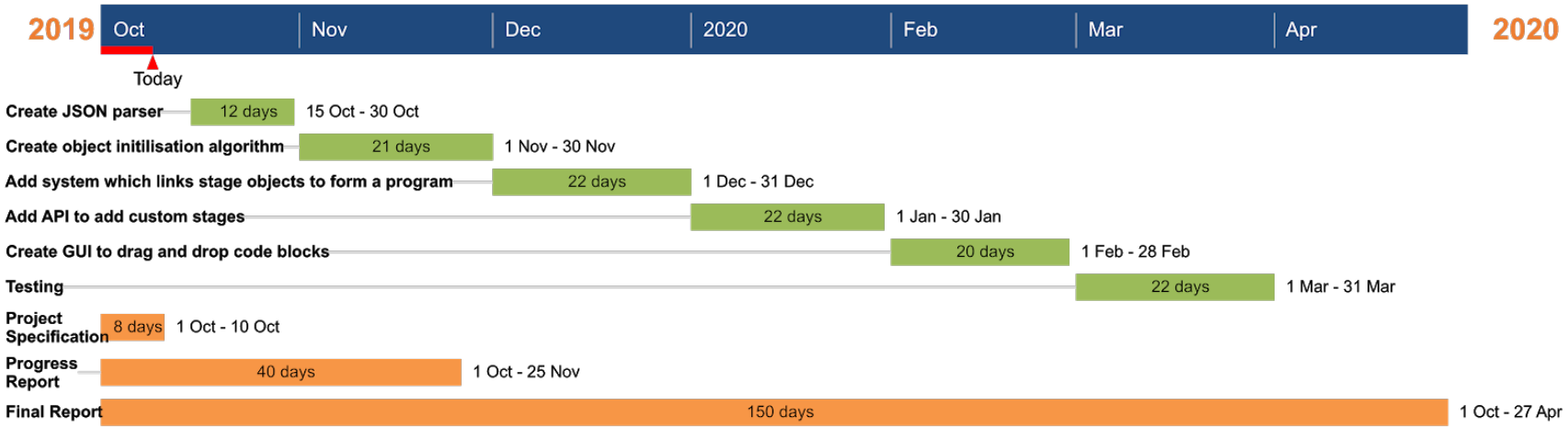
`How can others make use of the work in this project?`
See 'Use Cases' chapter.

`Why should this project be considered an achievement?`
It empowers users who are not technical to be able to take advantage of the power of object-oriented programming.

`What are the limitations of this project?`
It is restricted to Java. It can only be used to instantiate class constructors that have parameters with primitive types, or types that have been registered as block classes. Some classes may need their constructors to be modified to allow them to be represented.

9 Appendices

A Project Timetable Gantt Chart


2019
Oct
Nov
Dec
2020
Feb
Mar
Apr
2020
Today
Create JSON parser
12 days
15 Oct - 30 Oct
Create object initilisation algorithm
21 days
1 Nov - 30 Nov
Add system which links stage objects to form a program
22 days
1 Dec - 31 Dec
Add API to add custom stages
22 days
1 Jan - 30 Jan
Create GUI to drag and drop code blocks
20 days
1 Feb - 28 Feb
Testing
22 days
1 Mar - 31 Mar
Project Specification
8 days
1 Oct - 10 Oct
Progress Report
40 days
1 Oct - 25 Nov
Final Report
150 days
1 Oct - 27 Apr

References

[1] Google Blockly https://developers.google.com/blockly

[2] Scratch https://scratch.mit.edu/

[3] JSON https://www.json.org/json-en.html

[4] GSON https://github.com/google/gson/blob/master/UserGuide.md

[5] GSON Instance Creator https://www.javadoc.io/doc/com.google.code.gson/gson/2.6.2/com/google/gson/InstanceCreator.html

[6] Reflection https://en.wikipedia.org/wiki/Reflection_(computer_programming)

[7] JavaCC https://javacc.github.io/javacc/

[8] Top JavaScript User Interface Frameworks https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/

[9] React https://reactjs.org/

[10] React Native vs React https://stackoverflow.com/questions/34641582/what-is-the-difference-between-react-native-and-react

[11] Hash Based Table https://guava.dev/releases/19.0/api/docs/com/google/common/collect/HashBasedTable.html

[12] Java Classes in a package https://stackoverflow.com/questions/520328/can-you-find-all-classes-in-a-package-using-reflection

[13] Reflections Library https://github.com/ronmamo/reflections

[14] Maven https://maven.apache.org/

[15] Java constructor parameter names https://stackoverflow.com/questions/2729580/how-to-get-the-parameter-names-of-an-objects-constructors-reflection

[16] ASM https://asm.ow2.io/

[17] BigInteger and BigDecimal https://www.baeldung.com/java-bigdecimal-biginteger

[18] ArrayList documentation https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

[19] Trello https://trello.com/

[20] Agile Alliance https://www.agilealliance.org/glossary/epic/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'epic))~searchTerm~'~sort~false~sortDirection~'asc~page~1)

[21] List of best-selling video games https://en.wikipedia.org/wiki/List_of_best-selling_video_games

[22] Thomas Allton Java Neural Network Library https://github.com/tomallton/neural-network

[23] Breast Cancer Wisconsin (Diagnostic) Data Set http://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+%28diagnostic%29

[24] Scikit-learn https://scikit-learn.org/stable/

[25] SpigotMC API https://www.spigotmc.org/

[26] PokéFind https://www.pokefind.co/

[27] Swing https://en.wikipedia.org/wiki/Swing_(Java)