OBJECT-ORIENTATED AND REFLECTIVE PROGRAMMING

THOMAS WILKINSON

COMPUTER SCIENCE INTEGRATED MASTERS

MSCI FINAL YEAR PROJECT 2023

SUPERVISOR: ADRIAN JOHNSTONE

SCHOOL OF ENGINEERING, PHYSICAL AND MATHEMATICAL SCIENCE

ROYAL HOLLOWAY, UNIVERSITY OF LONDON

DECLARATION

This report has been written based upon my own work. Where other source materials such as textbooks or research papers have been used, they have been referenced.

Word Count: 20888

Student Name: Thomas Wilkinson

Date of Submission: 23/03/2023

Signature: 

TABLE OF CONTENTS

INTRODUCTION

PROJECT AIMS AND RATIONALE

Object-orientated and reflective programming are two powerful paradigms that have greatly influenced the programming world. They are also methodologies which can be naturally used in tandem with each other; many modern object-orientated languages provide reflection APIs which utilise reflective operations in an object-oriented setting. With this project I hope to gain a greater understanding and appreciation for how object-orientated languages really function, as this will be great knowledge to go into the industry with due to object-orientated languages being so commonly used in modern day programming, meaning any knowledge about object-orientation will be widely applicable.

Reflectivity, while not quite as widely utilised, is also a fantastic paradigm to have a deeper understanding for, allowing for the powerful ability to inspect and change code during runtime which most certainly has a place in our modern and dynamic society which requires more and more real-time systems. I will also acquire vital general experience in producing a project, using a version control system, software engineering techniques and working independently, which I will be able to tell future potential employers to my benefit, as these skills are valued highly within the computer science industry.

Throughout this project I will aim to further my exposure to both of these paradigms by performing research and writing two reports, exploring how these methodologies affect design decisions of different programming languages, and how they are implemented from a technical standpoint.

I will also produce multiple 'proof-of-concept' programs, which will aim to implement some ideas for my final language in a smaller setting, both in languages Java and Smalltalk. These will be highly useful as I begin to implement my final language as I will have already produced solutions to problems I may face.

On a technical standpoint, the final objective of this project is to produce an object-orientated and reflective 3D CAD language, which will allow for the creation and augmentation of 3D shapes, utilising the JavaFx and Javassist libraries to write the backend plugin, alongside the university tool ART for syntax and grammar interpretation. The language will provide a basic IDE and will feature elements of object-orientation and reflection such as sub-classing and editing constructors during runtime.

LITERATURE SURVEY

Object-orientated programming has been heavily researched since its conception, and so there is a vast array of papers about each facet. Tim Rentch's "Object-Orientated Programming" was a highly useful paper detailing object-orientated programming, with a heavy lean on the Smalltalk implementation, and also provides a slightly historical perspective due to the paper being written in 1982. It mentions concepts such as sharing that are highly informing and gives a good analysis on the Smalltalk model of objects being the only unit of structure.

To help learn Smalltalk as a language, "Smalltalk-80 The Language and its Implementation" by Adele Goldberg and David Robson was primarily used, which is a cover-all book on the concepts and syntax of Smalltalk, providing plenty of example code that immensely helped learning a language unlike anything else. Considering the Pharo implementation was chosen, "Pharo 9 by Example" by Stéphane Ducasse and Gordana Rakic provided a textbook that covers not only the specific syntax differences of Smalltalk in Pharo, but also how to use the IDE and tools such as inspectors very thoroughly, giving a fantastic foundation to begin to use Pharo, even providing two starter programs to help understand the model.

Reflection is a smaller area of research, yet there are still plenty of papers and resources. “Java Reflection in Action” by Nate and Ira Forman provided a great guide to not only the Java reflection API, but also to the concepts in general such as intercession and serialization, describing the idea and providing specific Java code which executes the given concept. The continual use of “George the Programmer” who has different needs which can be solved by reflection, where the needs are explained in an industry context, was a great writing choice and put into perspective why a certain function would be utilised.

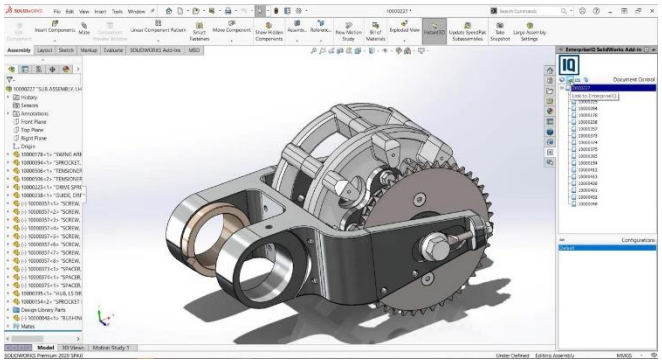
“Understanding and Analyzing Java Reflection” by Yue Li, Tian Tan and Jingling Xue was used as an auxiliary to understand reflection in the theory aspect, especially the problems and issues with reflection, such as operations being considered illegal due to reflection’s power of alter an execution state. Many different descriptive diagrams are featured and are very helpful in understanding flows of information between classes or objects.

ANALYSIS OF EXISTING SYSTEMS

Although not a 3D CAD language, SolidWorks is a 3D CAD piece of software that allows for the creation and manipulation of shapes in a 3-dimensional setting. SolidWorks is an application that I have some previous experience with, as learning how to use SolidWorks was mandatory for my GCSE Design and Technology class, where I used it to model the speaker I wanted to make. SolidWorks is mainly programmed in C++, and features a professional set of tools that allow for the creation and rendering of very complicated objects. This is performed by firstly creating a 2 or 3-dimensional sketch within the software, which defines the vertices, edges and other features. Relations can be used to define attributes such as a tangent or parallel relationship between multiple features in the sketch.

There will be marked differences between SolidWorks and UKCAD, mainly facilitated by the fact that SolidWorks is a piece of 3D CAD software, rather than a programming language. SolidWorks utilises a point-and-click GUI, much like the Microsoft Office line of products, allowing the user to select the exact point or tool they want to use, and to edit shapes in an extremely detailed manner with the wide variety of tools available. Therefore, UKCAD will be different as it is a 3D CAD programming language, which allows the user to build shapes through writing programs using the language’s own syntax and grammar, rather than using the mouse, which means that UKCAD must be designed to offer maximum functionality within that language. Also, SolidWorks is a professional standard piece of software that is powerful and complicated, and therefore UKCAD will not offer nearly as much functionality to create objects, as that would simply be out of the scope of this project.

However, there are still similarities and decisions that can be taken from SolidWorks and applied to UKCAD, as they both are used to model 3D objects. UKCAD should offer panning, scrolling and zooming just as SolidWorks does, allowing for the viewing of shapes from any side. Operations such as texturing, colouring and translating should be present, and as the 3D objects are the sole output, as much emphasis should be placed upon the ease of viewing these shapes; therefore, the user should be able to click on a shape and view its relevant information, and full screen the 3D scene to view the objects created.



An example of SolidWorks being used.

OBJECT-ORIENTATED PROGRAMMING

INTRODUCTION

The idea of having classes and objects, each with their own attributes, methods and possibly inherited structure from a higher class opened up a door into a whole new way of thinking about programming. This concept, although seemingly obvious and natural to a modern-day programmer, was ground-breaking back in the late 1950s and early 1960s when these ideas were being formed. Since then, object-orientated programming has grown into a huge field with different implementations theorised and languages created that supported or even championed object-orientated programming. It is important to note that while this report will detail core concepts and specific implementations of object-orientated programming, it will not be a definitive guide to the idea – the topic of object-orientated programming is vast, a credit to how popular and successful the paradigm has become within the programming sphere.

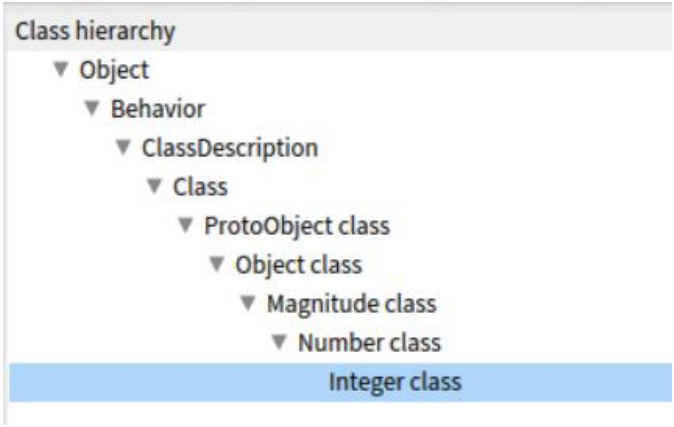
EARLY HISTORY

Although the terms “object” and “orientated” floated around in the 1950s and 1960s at MIT, it was not until Simula-67 was developed that the core ideas were really formed. Classes, objects, subclasses and inheritance all owe their conception to this language [1], and although Simula was used primarily as a research aid, the seeds had been planted. The first more mainstream language that utilised these ideas and the language that will be focused on primarily, was Smalltalk. Smalltalk, or originally Smalltalk-80, took these ideas and extended them; in fact, the term “object-orientated” came from Smalltalk [2]. Smalltalk’s vision was to entirely base the language on the idea of a class as the only unit of structure, with instances of these classes, called objects, providing the functionality of the language [3].

CORE CONCEPTS

CLASS AND OBJECTS

Within object-orientated languages, an obviously key concept is an object. Interestingly however, it is seemingly more important to view not what an object is, but how it appears. The outside perspective is important to object-orientated programming; this principle, coined by Tim Rentsch as intelligence encapsulation [4], provides a great metaphor for viewing the intrinsic behaviour of an object. Each object within Smalltalk, and in fact within any object-orientated language, is uniform in the sense that all objects communicate using the same message passing methodology. Quite simply, objects send messages to other objects, no matter what they represent. They are also uniform, at least in the Smalltalk world, in the fact that all objects are not given any real status. This concept of uniformity in relation to objects is known as polymorphism. Integers, for example, while primitive, share the same architecture as a system object, such as a class, which shares the same architecture as a user-created object. In Smalltalk this results in a 'class hierarchy', where each object, such as an integer, is a subclass of a higher class, all the way up to the generic object, and then metaobject class. This is shown well by the means to call or refer to such objects; they are always referred to as a whole, complete atom, using their unique internal name. Logically, this therefore implies that objects cannot be internally looked at or updated, also known as "smashing" its state. While it is possible to replicate these actions, this is only achievable through the object choosing to provide this behaviour; it is not provided by the language itself [5]. This is a key paradigm in object-orientated programming and is known as encapsulation.



The class hierarchy of Integer in Smalltalk

Each subclass will inherit the methods and class variables of the higher class

MESSAGE PASSING

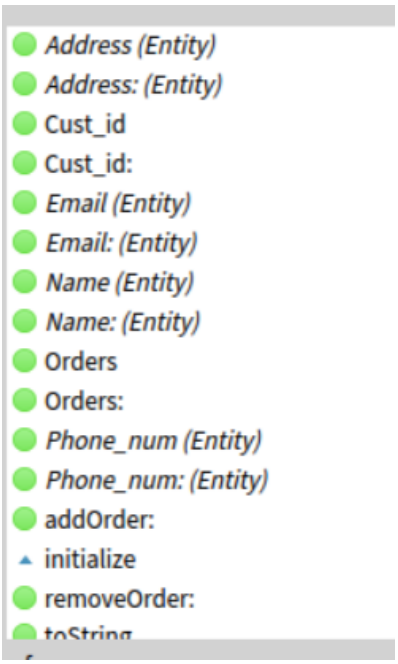
It is not enough to just have objects though; for events to happen, processes must take place. Within Smalltalk, this happens inside the objects themselves as they provide their own computation. Objects must also communicate with each other, as internal computation can only take a program so far. This is done through message sending, which is uniform not in its content or format, but in the fact that all operations are performed through the same message pattern – messages are sent using the same techniques no matter who the sender and recipient are, and what the content of the message is [6]. This design follows naturally from the idea of objects being uniform, as if all objects follow the same template, the messages sent also need to be uniform in their structure. Objects respond to these messages with a 'reply', which in all cases is an object name. The goal of a message is to inform the recipient to begin processing and to request data. This is specified within the text of the message, and any additional parameters consist of information the recipient will need, or to specify the functionality that an object supports [7].

Smalltalk provides an interesting and perhaps unique approach to message passing, at least at the time, in that the sender has ultimate power; it does not care what the receiver can do, it is up to the receiver to handle the information provided and carry out the correct operation [8]. Of course, this requires a certain amount of trust on the sender's part, but this idea is crucial to the object-orientated design. In the event that an object receives a message it does not understand, the virtual machine sends a "doesNotUnderstand" message back to the object, including the reification of the offending message. This reification is produced by creating an instance of the class "Message", which allows for an overview of the message semantics; that an instance of a message can be created and used to debug further displays the power of the "pure" Smalltalk model.

SHARING AND INHERITANCE

The idea of sharing is another core concept within the philosophy. Objects have attributes, but these properties cannot be unique to each class, they must be shared with objects that also have this property. This is best summarised by using real-world objects as an example; a pair of trousers and a t-shirt both have the attribute 'wearable', yet there is not one instance of 'wearable' that is owned by the t-shirt and another by the trousers, it is a shared property. This provides what is known as 'factoring' [9], which enables code to be easy to understand and write, and also allows a degree of modularity; the property can be written and then applied to any object as needed. Sharing, while being relatively straightforward to understand, is a powerful design choice and provides object-orientated languages with the framework to have the best of both worlds; attributes can be shared with a group of objects, while also allowing these classes to redefine this property to be more individual to suit their needs.

Sharing naturally leads onto a corner stone of object-orientated programming, inheritance. Inheritance is a collection of ideas; namely classing, subclassing and superclassing. These tools would not be possible without the concept of sharing, in fact inheritance is simply a method to perform sharing [10]. Inheritance's main goal is to achieve modularity and therefore avoid code reuse, achieved via the inheriting of properties from a superclass. In Smalltalk this is made as clear as possible, as it allows you to not only inspect the class hierarchy of any given object, but it also allows you to view all inherited methods. Inheritance as a paradigm is also exceptional for providing elucidation [11], as it naturally sheds light upon the universe of classes and methods within the system.



The customer class's methods from the Smalltalk proof-of-concept program CustomerOrderingSystem. Each method with (Entity) written beside it is inherited from the superclass Entity.

MODERN IMPLEMENTATIONS

C++

C++ was designed as an extension to the low-level procedural language C. C did not support object-orientated programming as it did not provide functionality to ensure polymorphism, encapsulation and inheritance; without which the paradigm is not possible. While it is technically achievable to simulate object-orientated programming within C through the use of the object-orientated C kit, and although polymorphism in particular was implementable before the release of C++ through a data structure and a list of pointers, C itself is not an object-orientated language as it does not provide native support for classes. Therefore, C++ was developed as a version or superset of C that indeed does offer the ability to program in an object-orientated way [12].

Within C++, static typing is the default. Static typing provides guarantees as to a piece of data's type at compile time; it also therefore implies that the semantics of a concrete class cannot be changed at a later point. While this has its uses, as it requires no framework and time and space costs are eliminated during the use of its member functions, it also means that a derivation from this concrete class is almost self-defeating as a concrete type is self-contained and cannot be easily added to [13]. Therefore, this is where C++ provides functionality for object-orientated programming through the use of "virtual" functions. This type of function enables derived classes to provide their own instances or versions. To facilitate this overriding process, the "protected" keyword is also used to ensure a member of a class is accessible to both the classes' own members and also to the methods of classes derived from it. While this does achieve the goal of a malleable derived class, it also has drawbacks that should be considered, however negligible on modern machines; virtual function calls in C++ are fractionally slower than their static counterparts as the compiler cannot inline them, the use of free store is now a necessity as derived classes can vary in size and accessing requires pointers or references to ensure polymorphism [14]. These costs do place into perspective why static dispatch is the default, especially considering C++ is a derivation of C which is a language that is often used on resource-limited systems.

```
class Date2 {
public:
     // public interface, consisting
     // primarily of virtual functions
protected:
     // representation and other
     // implementation details
};
```

An example of a C++ class, which uses virtual functions to allow derivations to provide their own versions. The protected keyword provides derived classes' member functions access.

C++ as a language was inherently designed to facilitate backwards compatibility with C. C++ mostly introduced new language features that did not interfere with C's existing language structures, for example classes, constructors, exception handling and templates which, along with other abilities, allowed C++ to support data abstraction [15]. This notion, along with concerns about efficiency, resulted in a unique implementation of object-oriented programming. A template object of a derived class within C++ has the functionality to pass operations as parameters, this is paramount in terms of efficiency as it is easy for the compiler to inline all uses of a certain function, a notable advantage of C++ over C where operations must be passed as pointers, causing a possibly significant overhead.

```
template<class C>
class std_coll {
public:
    bool eq(C a, C b)
        { return a==b; }
    bool lt(C a, C b)
        { return a<b; }
};

template<
    class C,
    class Coll = std_coll<C>
        >
int cmp(
    string<C>& s1,
    string<C>& s2
       )
{
    // compare s1 with s2
    // using  Coll::eq and Coll::lt
    // to do character comparisons
}
```

Two independent templates in C++ for a collating sequence and string comparison. The cmp template takes a derived class from std_coll as a parameter, allowing access to the eq and lt functions.

C#

Although C++ was created to be an object-orientated language, it still was considered a relatively difficult language to learn as the user was required to manage various low-level tasks such as memory management and security [16]. In the 1990s Java became popular which ran on top of a set of class libraries, meaning that these low-level tasks were automatically handled. This influence resulted in Microsoft creating C#, called C# as it incorporated syntax from C, mainly to appeal to C++ and Java developers. C# was designed to incorporate at-the-time modern concepts, one of which was object orientation. This implementation however differed from the implementation in C++. C++ offered multiple inheritance, where a class can inherit features from multiple superclasses, but the use of which was optional. Within C#, multiple inheritance is not possible, but instead allows the implementation of multiple interfaces. This shows the fundamental difference in design decisions; even when writing an application within C# that is not object-oriented, some object-oriented functionality must be used [17]. C# also draws a clear line between a class and a struct, where structs cannot inherit from any class, and do not provide the ability to have multiple constructors compared to C++, where the two are almost identical, illuminating the fact that C++ was a language built as an extension to a language that did not support the object-oriented paradigm.

GENERICS

Many object-oriented languages by default incorporate some type of support for generics; for example C++ offers the previously mentioned template structure. This is due to generics fitting nicely in with the philosophy; generics allow for a class or method to be defined with no specific type on its parameters. They further the idea of uniformity within objects and message sending structure. However, it was not until JDK 1.5 that Java released native support for generics [18]. This implementation differed significantly from the design of generics within C++; Java generics were designed to provide backwards compatibility with older Java code. C++'s translation model of generics is based on instantiation – the idea of no static type checking being done on generic code. Instead, the compiler will instantiate copies of a class or method with a distinct type, and type checking is then performed. This design results in no type information being lost during runtime as types are assigned by the compiler, although it does imply a certain amount of excess code [19]. To contrast, the Java methodology pushes retrofitting distinct types with the generic data structure, which are replaced with the bounding type; this type-erasure model loses run-time information, removing the need for operations which require this data.

Java generics support type requirements being defined on arguments as a sequence of abstractions – known as genericity. Interfaces are Java generic data structures used to model a concept through subtyping, leading to the problem of increased coupling between the generic and the constraints as the type will depend on a supertype [20]. C++ on the other hand does not offer any way to constrain these parameters, although it is possible to check constraints in C++ through the use of a library utilising compile time assertions. Java generics in fact ship without functionality for type aliasing and associated types, two traits of generic programming that are offered by C++ templates, through typedef and the traits mechanism. The traits mechanism facilitates the use of encapsulated types, Java classes only allow for encapsulation of methods and data. Therefore, Java associated types become extra parameters, resulting in code bloat and repetition as they need to be manually passed. Java generics also feature some notable disadvantages due to the backwards compatibility focus, such as primitive types being unable to be passed as type parameters in a generic method or class and generic type variables not having the ability to appear in catch clauses [21].

```java
public interface GenericAnimal {

    public enum Sex {
        MALE,
        FEMALE
    }

    public void setName(String name);

    public String getName();

    public void setSex(Sex sex);

    public Sex getSex();

    public void setWeight(int weight);

    public int getWeight();

    public void setColour(String colour);

    public String getColour();

    public String toString();
}
```

An example of a generic Java interface.

REFLECTIVE PROGRAMMING

INTRODUCTION

The modern-day concept of reflection, where reflection is built into a high-level procedural programming language, as opposed to the innate reflection that assembly and other low-level languages provide, was conceptualised by Brian Cantwell Smith in 1982 as a constituent piece of the language Lisp, more specifically the dialect 3-Lisp [22]. This report will detail what reflection is, along with key concepts such as introspection and serialization, outlining different implementations in modern programming languages. It is important to note that reflection as a paradigm is wide, with each language having a slightly different idea of what reflection means, usually defined by the languages design, and therefore this report will not be a definitive guide to the concept; rather a discussion of the different implementations.

CORE CONCEPTS

Reflection as a concept can be the solution to seemingly very different problems, for example implementing a component after a product has been released or increasing security by changing an already deployed system to only accept requests from certain packages. Both of these very real scenarios contain an element in common; a change in code based on the pre-existing structure to ensure an update of requirements. This adaptation process is methodical and could be illustrated in an algorithm [23]:

1. Examine a program's data and structure.
2. Make decisions based on the findings
3. Implement these decisions through augmenting the behaviour, structure, or data of the program

The true power of reflection comes from the realisation that these tasks do not need to be performed by a programmer manually, they can be realised by the program itself. Therefore, to ensure these specifications are met, reflection needs to be able to provide functionality to allow a running program to examine itself, and to choose what action it performs based upon its findings. This self-examination is dependant on "metadata"; data which denotes the program and its structure, rather than the data within the program itself. In an object-orientated setting, this metadata follows the paradigm and is grouped into "metaobjects", differentiated from "base-level objects" which implement the purposes of the system, and which can be examined during runtime to provide what is known as "introspection" [24]. An example of a metaobject within Java is a Class or a Method object, containing their respective metadata. Instances of these metaobjects represent the program itself.

```
Class cls = obj.getClass();
```

A Class metaobject in Java

INTROSPECTION AND INTERCESSION

Introspection provides the information about the program needed to facilitate the most useful step of reflection, the behaviour change, or where the program modifies itself. In order to support this powerful ability, there are three different techniques that can be utilised [25]. Direct metaobject modification is where the metaobject can be augmented to change the properties of the objects that are implemented from it, such as a type or a class to be derived from. Operations for using metadata allow for the calling of operations which perform some function using the metadata supplied; a great example of this is dynamic method invocation, where a method is called but the object ID or class are unknown. Intercession is the ability of a program to alter its own execution state or interpretation, this requires functionality to realise its own execution state as data; the process which results in this data is known as “reification”.

An example of dynamic method invocation, where a “Monkey” object is instantiated, and its setter methods are found and invoked. The class of the parameters must be specified so that the methods can be found successfully. It is important to note that each method is identified by both its declaring class and its signature, meaning that a method can only be invoked on an instance of its declaring class, and not on any object with the same method.

```
Monkey monkey = Monkey.class.getDeclaredConstructor().newInstance(); // Finds constructor for class and creates new instance
monkey.getClass().getDeclaredMethod("setName", String.class).invoke(monkey, name);
monkey.getClass().getDeclaredMethod("setWeight", int.class).invoke(monkey, weight);
monkey.getClass().getDeclaredMethod("setSex", Sex.class).invoke(monkey, sex);
monkey.getClass().getDeclaredMethod("setColour", String.class).invoke(monkey, colour);
monkey.getClass().getDeclaredMethod("setType", MonkeyType.class).invoke(monkey, type);
```

SERIALIZATION

Reflection can provide flexibility through the invocation of methods discovered at runtime. However, the accessing and modification of fields can also provide significant advantages to a programmer interested in flexibility. Serialization is a process to represent an object as text or another form of data, such as binary, and is an example of an operation that is made possible by reflection, as this representation is dependant on accessing the object’s metadata, specifically interested in the data held and structure of the fields of that object [26]. This overview of an object can be utilised in different ways; it can be used to allow the finding and invocation of that object’s methods to facilitate dynamic method invocation for example, as the representation will specify what methods an object has access to.

```
public static Field[] getFields(Object obj) {
    Class cls = obj.getClass();
    List accum = new LinkedList();
    while (cls != null) {
        Field[] f = cls.getDeclaredFields();
        for (int i = 0; i < f.length; i++) {
            accum.add(f[i]);
        }
        cls = cls.getSuperclass();
    }
    Field[] allFields = (Field[]) accum.toArray(new Field[accum.size()]);
    return allFields;
```

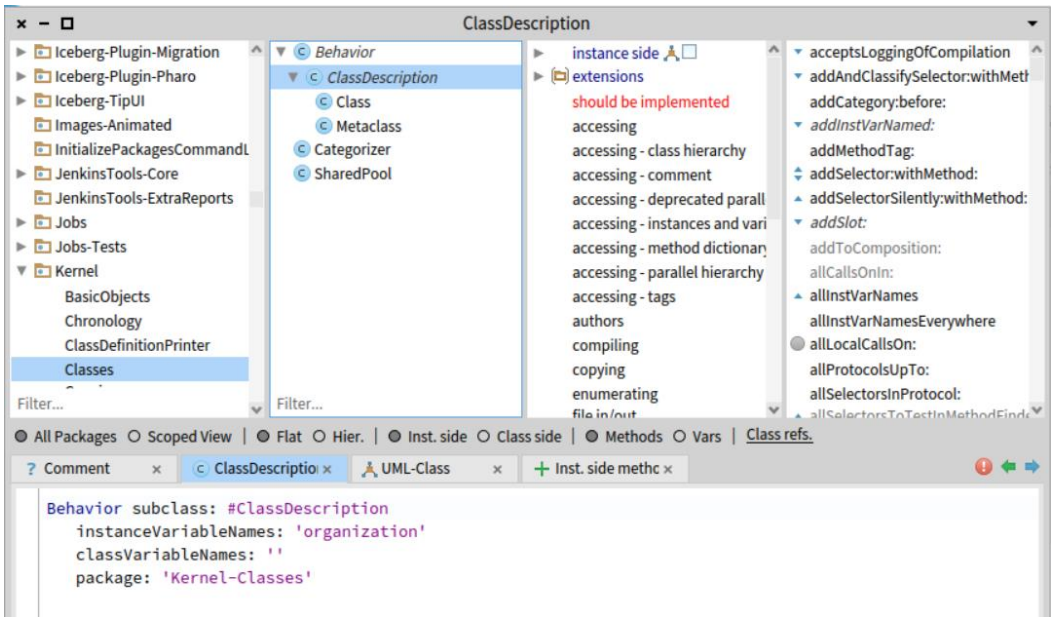
A method which obtains the values of all an object’s fields, both declared and inherited. This is paramount for serialization as it must access all information about an object.

SMALLTALK

Within Smalltalk, the paradigms of reflectivity and object-oriented programming are interlinked. At the meta-level, or the level at which how operations are performed is described, the techniques to implement inheritance, message passing, instance creation and other actions are specified. As Smalltalk is a partially reflective language, these specifications are enabled through message passing at the meta-level, with metaclasses controlling the instantiation and maintenance of classes and therefore objects. A class is simply an instance of a metaclass, differentiated as the metaclasses' instance variable environments are host to instance variables and methods that define a class. Smalltalk however is not a fully reflective language as it contains several intransigent language features, or a feature which has been built in at all levels, implying it cannot be extended or modified [27]. Two examples of this are the representation of objects and the specifications of message passing, as these are uniform protocols that apply to every object and message.

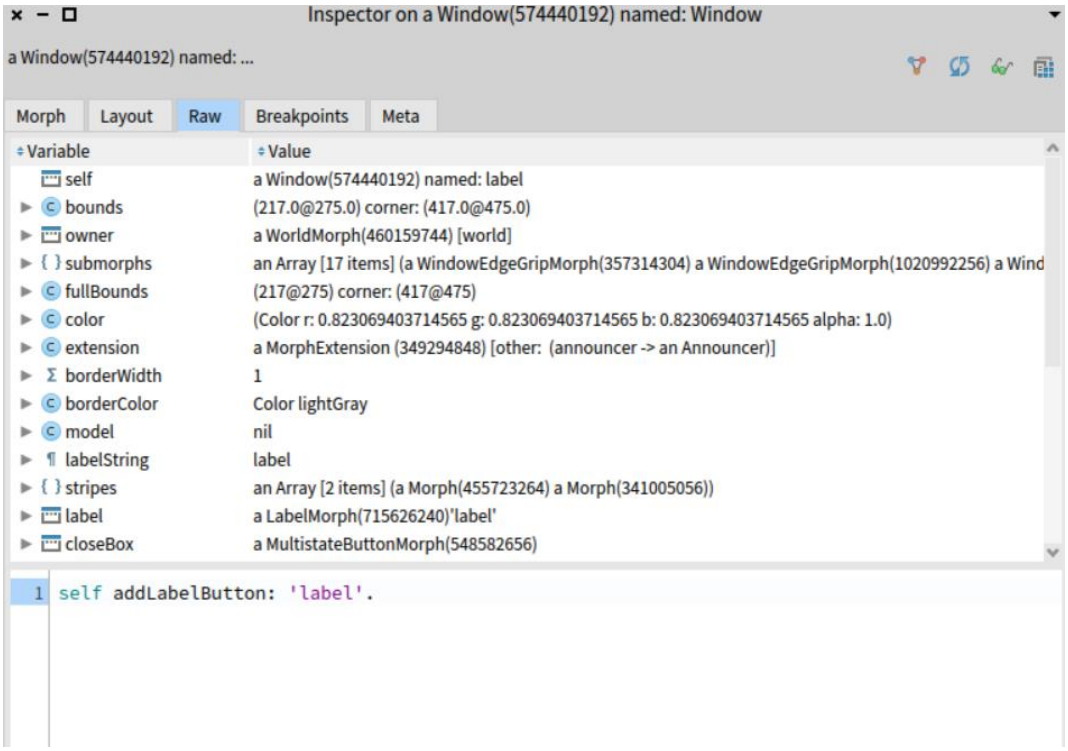
As metaclasses are a higher level than a normal class, an inevitable recursive query could be to the existence of a "metametaclass", or a class which defines the properties of a metaclass. However, these classes are not featured as they are not necessary, made possible by the fact that metaclasses can be defined in terms of themselves. This therefore implies that metaclasses are the highest-level class, indeed by design as otherwise the language designers would face a recursive problem as to where to stop. Therefore, these definitions can be said to be "meta-circular"; the classes are presumed to already exist during their own definitions, which is made possible by "boot-strapping" initial values so they may assemble themselves [28].

Smalltalk naturally lends itself to reflection partially due to the design decision to allow the viewing and indeed changing of these "metaobjects" through "meta-operations", which would normally be considered illegal access in many other programming languages. This is made possible by the uniformity of objects in Smalltalk, as the metaobjects are no different structure-wise to user-defined objects, simply containing content such as methods which define that metaobject.



The definition of the metaobject “ClassDescription”, which specifies the instance variables and the meta-operations that can be performed. This metaobject therefore defines how classes can operate and is called each time a new class is created, as all user-defined classes are a subclass eventually of ClassDescription. The fact that this metadata is viewable and modifiable provides natural introspection and direct metaobject modification.

Another feature that provides introspection and fantastic elucidation within Smalltalk is an “inspector”. Inspectors allow for the user to browse and interact with any object during runtime, showing the structure and the raw values, along with all metadata of any object that has been inspected, displayed as a class hierarchy and methods available to each class. Inspectors can be used to facilitate reflection very well, as code can be ran within the inspector to call or change the properties of an object while the program is running.



An inspector on a user-created Window object, displaying the raw values which describe the object. The code in the bottom box can be ran during runtime, which will produce a button to change the label of the running window to the string argument.

JAVA

The implementation of reflection in Java is intriguing compared to the Smalltalk model, as Smalltalk offers natural reflection based on the object being the only unit of structure, therefore implying that metaclasses can be examined and extended just as a user-defined class could be. Java however supports introspection and only very limited intercession, yet still can be used to remove the constraints of encapsulation and static typing, allowing for dynamic behaviour. One of the key reasons Java does not support reflection to the same extent as for example Smalltalk is that it does not contain a reification function to represent the running system or execution state as a data structure [29], to be passed to the meta-operations. To replace this, a pseudo-representation is created at the start of the program's runtime, is based on the system's metaobjects, and exists until the execution is complete. This lack of a reification function has led to the existence of intercession within Java to become a relatively controversial topic.

The pointer-affecting methods of the Java reflection API are the functions that can resolve control and data flow information [30], the ability of which is crucial as operations that are reflectively invoked must be able to be carried out statically, otherwise values or control flow that other modules of the program depend upon may not exist. Therefore, pointer-analysis, which resolves these issues statically, must be affected through the reflective methods in the Java reflection API, which can be divided into three groups:

- Class-retrieving methods – concerned with the creation of Class objects

Class-retrieving methods are the natural start of reflection in Java, as they are responsible for creating class objects, from which reflection can begin. There are many class-retrieving methods; the most widely-used functions are forName(), which utilises string analysis to return a Class object represented by the string and getClass() or simply class, which return the dynamic type (or class) of the object which the method is performed on. The use of string analysis to facilitate reflection can be controversial as it may result in diminished soundness and precision, due to string arguments being useless when the value is not known, resulting in the target not being resolved statically [31].

An example of forName() being used.

```
Class cls = Class.forName(className: "VehicleSubClass");
```

- Member-retrieving methods – use introspection and retrieve metaobjects

Member-retrieving methods includes numerous functions that can be performed on a Class object to fetch its metaobjects, such as a constructor or a field. These metaobjects can also be used to examine the metaobjects of the target class. Member-retrieving methods are so numerous as each type of metaobject has four different methods [32], one to fetch the specified declared metaobject, one to fetch the specified declared or inherited metaobject, one to fetch all declared metaobjects of that type for that class, and one to fetch all metaobjects of that type from the given class.

```
Object obj = cls.getDeclaredConstructor().newInstance();
cls.getDeclaredMethod(name: "setWheels", ...parameterTypes: int.class).invoke(obj, (int) args[3].value());
cls.getDeclaredMethod(name: "setDoors", ...parameterTypes: int.class).invoke(obj, (int) args[4].value());
```

Two different member-retrieving methods; getDeclaredConstructor() and getDeclaredMethod() which rely on string analysis. Note that if a metaobject has arguments, to be found the class of the type of the arguments must be specified.

- Reflective-action methods – change the pointer information reflectively

These methods (nine in total) [33] can modify or utilise the pointer information, through their various side effects. An example of this is newInstance(), the side effect of which is to allocate an object with the type given by the metaobject and to create it with the given constructor, or the default if none is given. Invoke()'s side effect is a virtual call when an object is given, or a static call when there is none as then it will be calling to access a class method or variable.

PROGRAMS AND TECHNICAL ACHIEVEMENTS

ORDERING SYSTEM

A program written in Smalltalk which simulates a customer ordering system, where a customer can place an order of a certain item from a certain retailer.

IMPLEMENTATION FEATURES

The system comprises of four main classes; being "Customer", "Retailer", "Product" and "Order", which are used to derive instances to represent their respective entities. Customer and Retailer share fields and therefore are subclasses of the superclass Entity, a subclass of the generic class "Object". These instance variables and methods are therefore inherited by Customer and Retailer, where Customer adds methods to add and remove an Order object from the orders OrderedCollection, a dynamically sized array. Customer also has an "initialize" message, which creates a new OrderedCollection for each new Customer object.

The class definition of Customer

```
Entity subclass: #Customer
    instanceVariableNames: 'cust_id orders'
    classVariableNames: ''
    package: 'CustomerOrderingSystem'
```

Order's initialize message, which stores the current dateTime as an OrderedCollection due to its format.

```
initialize
    dateTime := DateAndTime now asOrderedCollection.
```

Product is a subclass of Object and contains an OrderedCollection of Retailer objects that sell this product. Upon initialisation, retailers is created.

Product's message to add a Retailer object

```
addRetailer: retailer
    retailers add: retailer.
```

"OrderingSystem" contains the functionality for the system. Upon instantiation, OrderingSystem creates the OrderedCollections "orderList", "productList", "retailerList" and "customerList", which are instance variables.

```
addCustomer: name Address: address Phone_num: phone_num Email: email
    | customer |
    customer := Customer new Name: name.
    customer Address: address.
    customer Phone_num: phone_num.
    customer Email: email.
    customer Cust_id: customerList size printString. "Sets size of customerList as id"
    customerList add: customer.
    ^ customer.
```

Message to add Customer. Ids for each object are set to be the size of the list before the item is added, ensuring that each id starts at zero upon the start of the program.

Upon the adding of an Order object, it fetches the given Customer via their id and then sets the address of the order to be their address.

```
   tempCust ifNotNil: [ order Address: tempCust Address ] ifNil: [ Transcript show: 'Customer not found'; cr ]. "Checks if customer exists and if it does sets address to address of customer"
```

Upon adding a Product, the given retailer is found and then the product is added to that retailer's products array.

```
    retailerList do: [ :retailer | retailer Retailer_id = id ifTrue: [item := retailer] ].
    item ifNotNil: [item addProduct: product] ifNil: [Transcript show: 'Retailer not found'; cr].
"Checks if retailer exists and adds product to retailer"
```

Upon the removal of an object which has been stored in an array, such as an order in a customer's orders list, this item is also found and removed.

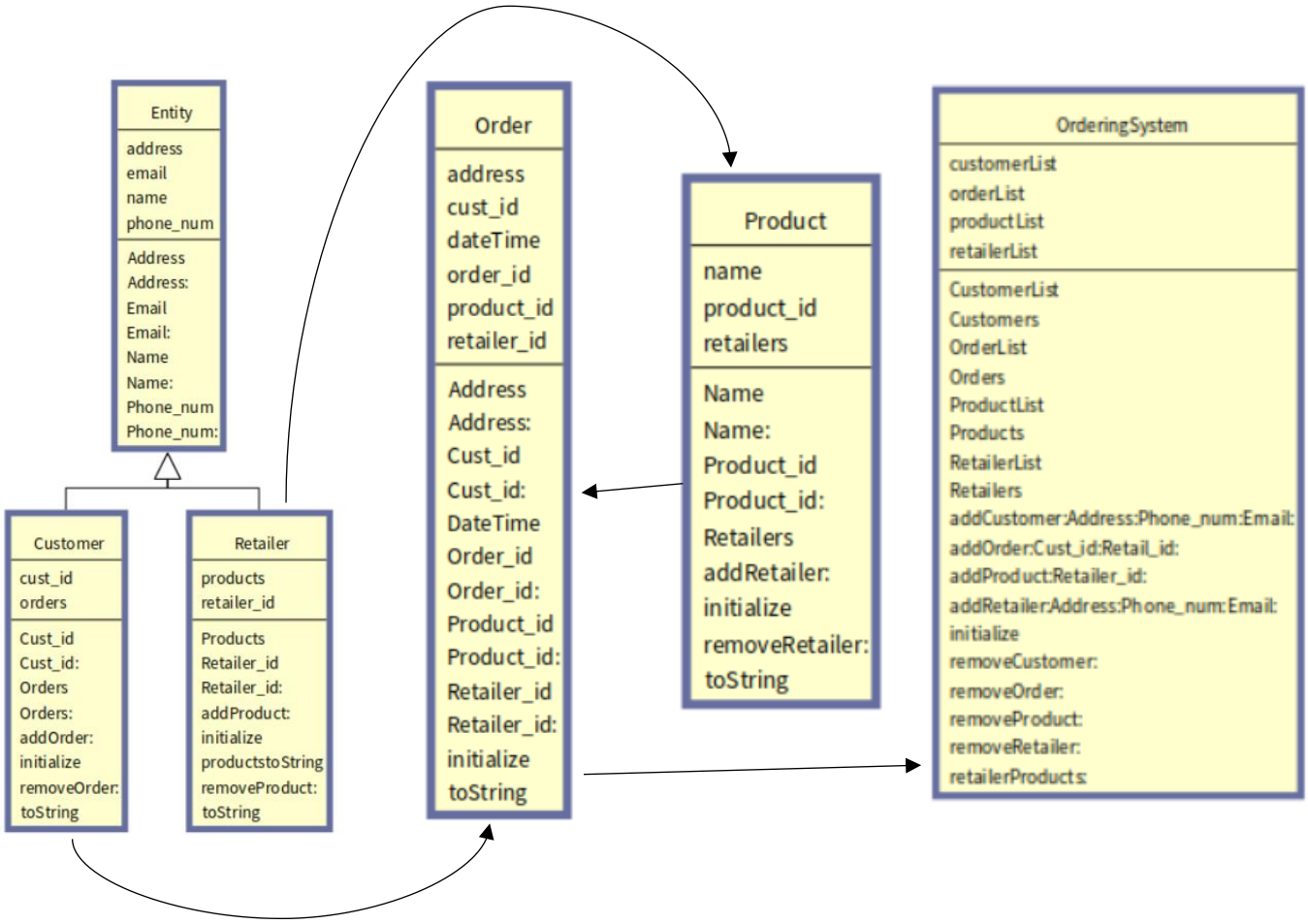
```
removeOrder: id
   | item |
   orderList do: [ :order | order Order_id = id ifTrue: [ item := order ] ].
   item ifNotNil: [ orderList remove: item ] ifNil: [ Transcript show: 'Order not found'; cr ]. "Checks if order exists and removes it"
   item ifNotNil: [ customerList do: [ :customer | customer Orders removeAllSuchThat: [ :order | order Order_id = id ] ] ]. "Removes orders from customer orders"
```

TDD AND UML

SUnit test cases were used in the production of this program, where important messages are asserted against the expected value to aid development by showing features were working as expected. In the cases of the use of OrderedCollections, the returned values were difficult to test against due to OrderedCollection's formatting; therefore the test cases were used to show the output using the debugger.

```
testremoveProduct
   | os |
   os := OrderingSystem new.
   os addRetailer: #(a) Address: #(b)  Phone_num: #(c) Email: #(d).
   os addProduct: #(Product) Retailer_id: '0'.
   os removeProduct: '0'.
   self assert: (os Products) equals: #('').
```

The test for removing a product. Note that arguments can be added as lists as Smalltalk's typing is quite loose due to types being objects



RELATION TO PROJECT

Through this program I began to understand the basics of Smalltalk and its model of objects being the only unit of structure, as well as the message passing methodology. Therefore, as Smalltalk is a strictly object-orientated language, I could conceptualise and model classes, objects and inheritance in a clear-cut way. I also came to understanding about the difference between class and instance variables, as I had a specific error as the Customer class was wrongly defined with its variables as class variables. These lessons were very useful in terms of my understanding of wider object-orientated programming, as well as helping me plan my final implementation of the 3D CAD language, which utilises object-orientated programming as shapes are instances of the specific shape classes. As the functionality for this language will be written in Java, knowledge from Smalltalk helped to further my understanding of, for example how the Sphere class is derived from the Shape3D class in JavaFx.

REFLECTIVE WINDOW

A short program which produces a window in Smalltalk that can be manipulated reflectively.

IMPLEMENTATION FEATURES

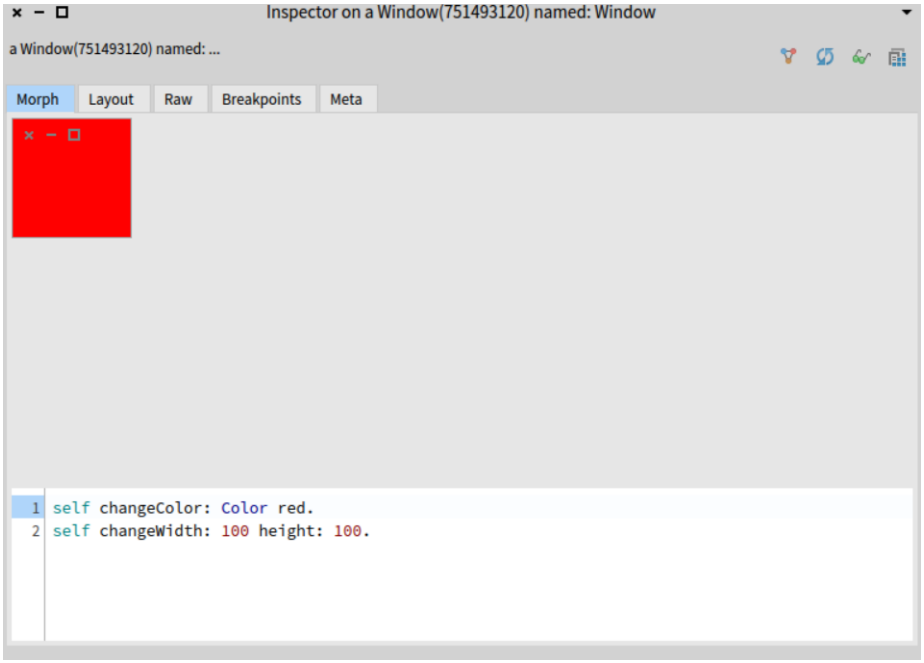
The program consists of one class, “Window”, that is a subclass of the Smalltalk class “SystemWindow”, which allows for the creation and augmentation of a window in the Pharo universe.

Window’s initialize message

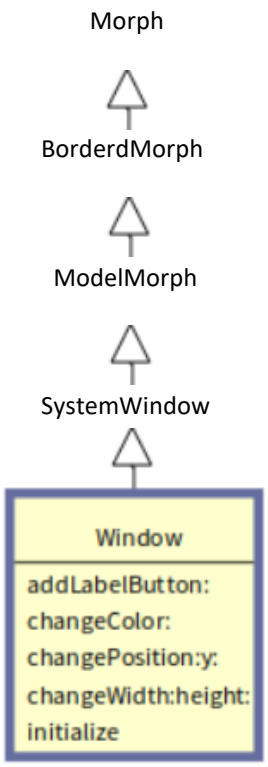
```
initialize
    super initialize.
    self setLabel: 'Window'.
    bounds := 0 @ 0 corner: 200 @ 200.
    self beResizeable.
    self openInWorld.
    self inspect.
```

Once the window has been opened, the inspector window can be used to run the messages for the manipulation of the window during runtime.

Inspector on Window object after the operations have been performed during runtime



UML



SystemWindow, ModelMorph, BorderdMorph and Morph are existing Smalltalk classes, which contain many methods and therefore will not fit on the page.

RELATION TO PROJECT

The reflective window class gave me a base point to go from in terms of understanding reflection, introducing me to concepts such as introspection and intercession while reading the Smalltalk documentation, as well as gaining practise of running code during runtime and the different way of thinking of running code dynamically instead of statically, such as the use of “self” to apply operations during runtime. This foundation gave me a base knowledge to take to applying reflection in Java.

ANIMAL ZOO

A system written in Java that allows for the reflective creation and manipulation of animals within a zoo.

IMPLEMENTATION FEATURES

The system contains three classes that specify the types of animal, those being “Monkey”, “Elephant” and “Lion”. Each of these classes implements the interface “GenericAnimal” and specifies its own animal type Enum.

```
public enum LionType {
    WHITE,
    ASIAN,
    AFRICAN
}
```

The “Zoo” class implements an ArrayList for each of the types of animal, representing an enclosure, and is used to store and fetch the animal objects from.

The method to find a monkey object

```
public Monkey findMonkey(String name) {
    for (int i = 0; i < monkeyEnclosure.size(); i++) {
        if (monkeyEnclosure.get(i).getName().equals(name)) {
            return monkeyEnclosure.get(i);
        }
    }
    return null;
}
```

The “ReflectiveZoo” class implements a command-line interface utilising string analysis to allow for performing operations at runtime. The “getFields” method takes an object of any type and finds its class, using this class object to get all declared fields for this class, and then all declared fields for the superclasses of this class, returning an array of field objects containing and declared and inherited fields. This method was taken from the Java Reflection in Action book by Nate and Ira Forman. Methods for setting a fields array to be accessible or nonaccessible, meaning they can be examined and changed at runtime, are provided.

The method to set the fields in a given fields array as accessible

```
public static void setAccessibleFields(Field[] fields) {
    for (int i = 0; i < fields.length; i++) {
        fields[i].setAccessible(flag: true);
    }
}
```

“getValues” takes an instance of a class as well as an array of that class’s fields and returns a list containing the raw data values stored within that object’s fields

```
public static List getValues(Field[] fields, Object obj) {
    List values = new LinkedList();
    for (int i = 0; i < fields.length; i++) {
        try {
        values.add(fields[i].get(obj)); // Gets raw value and adds to list
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
    return values;
```

```java
public static Monkey newMonkeyReflect(String name, int weight, Sex sex, String colour, MonkeyType type) {
    try {
        Monkey monkey = Monkey.class.getDeclaredConstructor().newInstance(); // Finds constructor for class and creates new instance
        monkey.getClass().getDeclaredMethod(name: "setName", ...parameterTypes: String.class).invoke(monkey, name);
        monkey.getClass().getDeclaredMethod(name: "setWeight", ...parameterTypes: int.class).invoke(monkey, weight);
        monkey.getClass().getDeclaredMethod(name: "setSex", ...parameterTypes: Sex.class).invoke(monkey, sex);
        monkey.getClass().getDeclaredMethod(name: "setColour", ...parameterTypes: String.class).invoke(monkey, colour);
        monkey.getClass().getDeclaredMethod(name: "setType", ...parameterTypes: MonkeyType.class).invoke(monkey, type);
        return monkey;
    } catch (NoSuchMethodException | InstantiationException | IllegalAccessException | InvocationTargetException e) {
        e.printStackTrace();
    }
    return null;
```

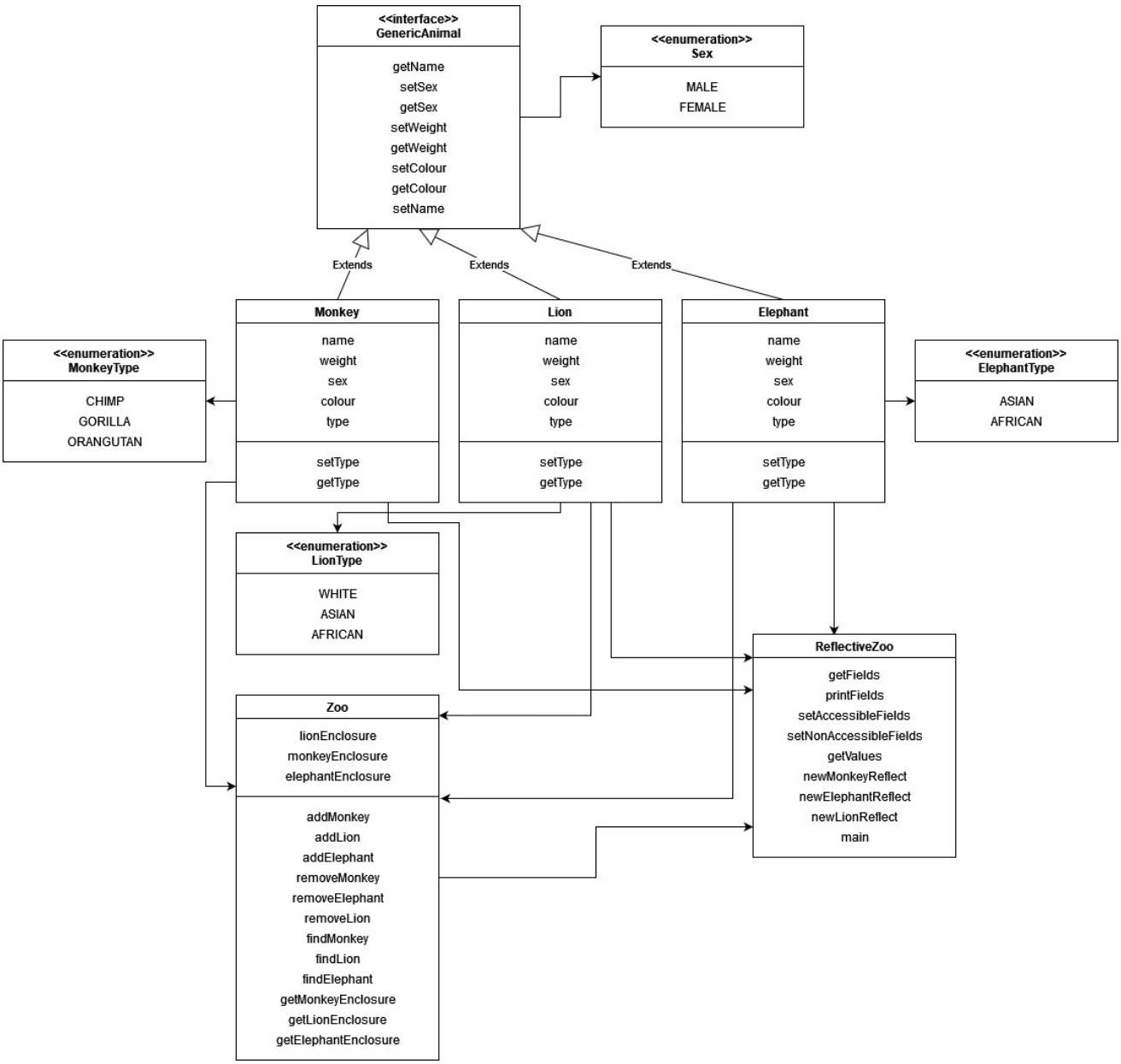
The “newMonkeyReflect” method, which reflectively creates a monkey object. A different implementation may have used the constructor to assign the values, but I chose to invoke the setter methods and to use an empty constructor to gain practise invoking a method at runtime.

One final operation was to access and set a field to a new value reflectively, which is performed directly within the command line interface in the main method. Another implementation of this operation could find the field directly instead of finding all the fields and then searching through the fields array.

Operation to set a monkey field during runtime

```java
if (animal.equals(anObject: "monkey")) {
    Monkey monkey = zoo.findMonkey(name);
    Field[] fields = getFields(monkey);
    setAccessibleFields(fields);
    Field field = fields[0]; // Gets first name field
    switch(field_val) {
        case "name":
            field.set(monkey, value);
            break;
        case "weight":
            field = fields[1]; // Gets second weight field
            field.set(monkey, Integer.parseInt(value));
            break;
        case "colour":
            field = fields[3]; // Gets fourth colour field
            field.set(monkey, value);
    setNonAccessibleFields(fields);
    break;
```

UML



RELATION TO PROJECT

This program was immensely useful in that fact that I was able to gain practise using the Java reflection API and the reflective operations, such as getting the fields of a class or declaring constructors and invoking methods, which is exactly what I will be using during the production of my end implementation, where these exact reflective methods will be used. Therefore, I was able to overcome issues early such as the mass exceptions needed or the addition of a type's class when invoking a method, which may having caused me problems when implementing reflection within the CAD language. I was also able to further my knowledge of reflection and the concepts which aided in the writing of the reflection report.

GARAGE

A small custom programming language which supports the ability to create subclasses and objects.

IMPLEMENTATION FEATURES

The attribute grammar supports cases to add a subclass of a given class or an object, or to print an object to a file; in this language the only class to be derived from is the class Vehicle.

```
| 'object' ID '=' ID '(' '"' ID '"' ',' '"' ID '"' ')'
  { iTerms.valueUserPlugin.user(new __string("object"), new __string(ID1.v), new __string(ID2.v), new __string(ID3.v), new __string(ID4.v)); }
```

Instead of parsing through the attribute grammar, eSOS rules and the external to internal syntax can be used. It features a term for sequencing statements, as all the operations must be sequenced into one line, as the eSOS rules require the formation of a “try” statement. Three cases for the operations of subclassing, the creation of objects and printing an object to a file are present.

```
class ::= subClass^^ | object^^ | printToFile^^
subClass ::= 'subClass'^ ID '='^ ID '('^ subExpr ','^ subExpr ')'^
object ::= 'object'^ ID '='^ ID '('^ '"'^ ID '"'^ ','^ '"'^ ID '"'^ ')'^
printToFile ::= ID '.printToFile'^
```

eSOS rules for the three methods, which pass the arguments alongside a string to be analysed are present.

printToFile eSOS rule

```
-printToFile
---
printToFile(_h),_sig -> __user("printToFile",_h), _sig
```

The backend plugin initialises two HashMaps, for storing Class and Object objects respectively, using the passed name as a handle. The “newClass” method creates a class object using the library “Javassist”, and sets the superclass to be the class passed, in this case always Vehicle. The Vehicle class provides fields for the name and colour, and features accessor methods as well as a constructor.

```
public class Vehicle {
    public String colour;
    public String name;

    public Vehicle(String name, String colour) {
        this.name = name;
        this.colour = colour;
    }

    public void setName(String name) { this.name = name; }
    public void setColour(String colour) { this.colour = colour; }

    public String getName() { return this.name; }
    public String getColour() { return this.colour; }
}
```

The class Vehicle, used as a superclass

The newClass method then adds two static integer fields named “wheels” and “doors”, which are the class variables. These variables are then found and set reflectively to the arguments passed. Finally, this class is then added to the HashMap “nameToClass”.

```
CtField wls = CtField.make("public static int wheels;", subClass);
subClass.addField(wls);
CtField drs = CtField.make("public static int doors;", subClass);
subClass.addField(drs);
```

Creation and adding of the class variables

The “newObject” method finds the class using the class name argument from the HashMap and then reflectively calls its declared constructor, passing the string values given. This object is then added to the HashMap “nameToObject”.

```
Object obj = cls.getDeclaredConstructor(...parameterTypes: String.class, String.class).newInstance(args[3].value().toString(),
args[4].value().toString());
```

The calling of the declared constructor

The “printToFile” method is featured as it was a simple way to get an output for testing purposes. A new file is created using the my filepath to my project, the respective fields and object are found and then the values from that object’ fields are written to this file. These fields are found using the reflective method “getField”.

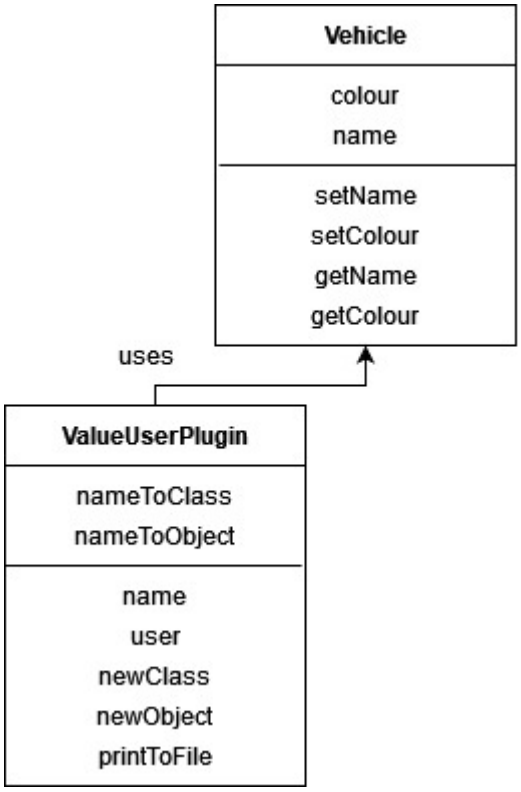
```
Class cls = obj.getClass();
Field doors = cls.getField(name: "doors");
Field wheels = cls.getField(name: "wheels");
Field name = cls.getField(name: "name");
Field colour = cls.getField(name: "colour");
```

The finding of the class’s fields reflectively

Finally, the language allows for the writing of programs using these operations:

```
subClass Car = Vehicle(4, 4)
subClass Van = Vehicle(4, 2)
subClass Lorry = Vehicle(8, 2)
object BlueFordFocus = Car("Ford_Focus", "blue")
BlueFordFocus.printToFile
object WhiteTransit = Van("Transit", "white")
WhiteTransit.printToFile
object RedLorry = Lorry("Red_Lorry", "red")
RedLorry.printToFile
```

UML



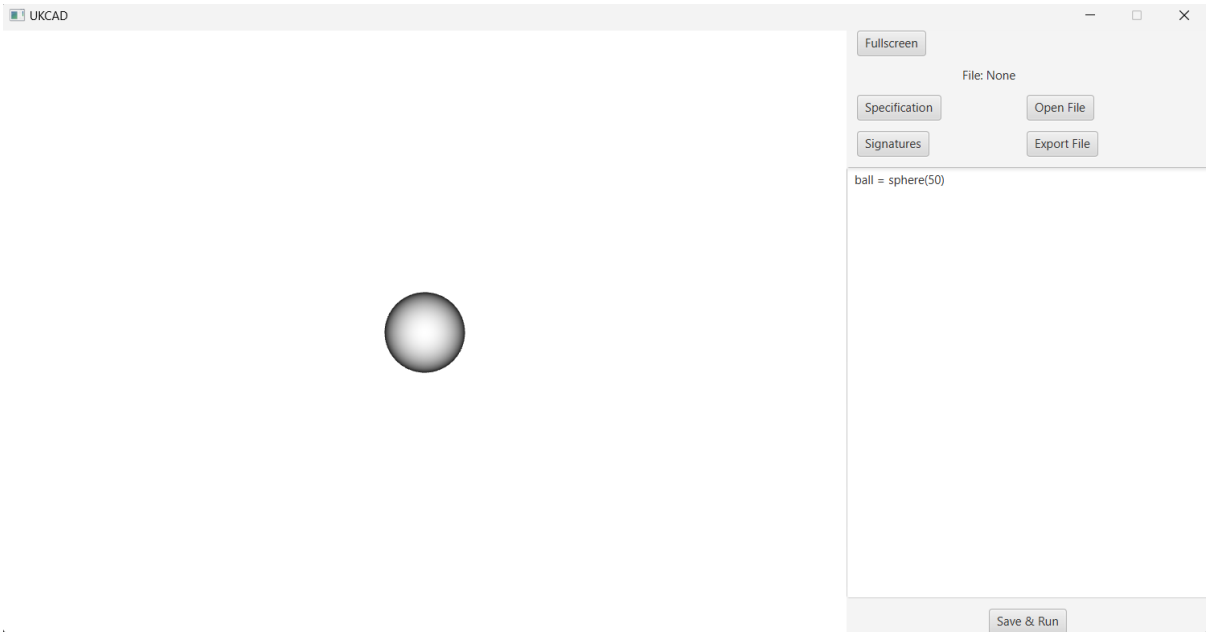
RELATION TO PROJECT

This small language implements a very limited form of classes, whereby a subclass can be created with the only difference being the value of some class variable. This is the exact feature that will be added to the CAD language, whereby a subclass of an already existing JavaFx shape class will be able to be created, with class variables indicating the bounds, or another property, of the shape. Therefore, these custom shapes will be able to have instances created augmenting them further. This implementation of classes will allow the CAD language to feature a form of object-orientated programming, beyond the expected utilisation of existing shape classes. Furthermore, the production of this language provided revision on how to use eSOS, ART and other tools which will be taken into the final implementation, as well as how to the use the library Javassist to create and manipulate class objects.

END LANGUAGE - UKCAD

OVERVIEW OF ARCHITECTURE AND FEATURES

UKCAD is a 3-dimensional Computer-Aided Design language, featuring its own syntax and grammar, as well as a basic IDE. It allows the user to create and customise 3D shapes from which they are able to model a desired object. The front end of this language is parsed using the ART library, which allows for the specification of a grammar and syntax and generates parser and lexer files based upon the rules given, which are compiled into class files upon running and used to convert the program written by the user into values to pass to the plugin. The back end of this language is a Java plugin file which sets up the basic IDE and draws and manipulates the shapes using the JavaFx library.



UKCAD upon startup with the UKCAD.bat file, running the default program.

UKCAD as a language allows for the creation of spheres, cuboids, cylinders, pyramids, triangular prisms and more. It also provides multiple “transform” methods, which are used to modify a shape in some way, such as the “colour” operation, which sets a certain shape to be a certain colour. Furthermore, UKCAD also features some additions which can be immensely useful when creating complex objects. Within UKCAD, each shape can be used to create a subclass, taking a variable number of arguments to add a certain “transform”, which will be applied upon the instantiation of this subclass automatically. No transform may also be applied to the subclass. Multiple transforms may be applied to the subclass through the “addTransform” method, which will all be applied upon object creation. When creating an object from a custom subclass, the arguments for the shapes dimensions must be given, the number of which depends upon which shape was used as a superclass for the subclass. The user can choose between the creation of one object, whereby the subclass is used just as a shape class would be with the object being given a name, or the user may choose to create several shapes, the number of which is specified. In this case, the user provides a name for the shapes by giving a value in the form “[name]”, which will be used as a base key for the objects, so that the first will be called “name1”, the second “name2” and so on. Each of these shapes can be individually modified by simply performing a transform using these names. UKCAD also provides two operations which are performed on this “set” of shapes, namely “translateAll” and “rotateAll”. These are the only transforms which cannot be applied to a subclass, and perform the respective transform, either translate or rotate, to each shape in the “set” incrementally, using start and step values.

UKCAD uses the library ART to specify syntax and grammar. ART also is used to provide parsing, term rewriting and the production of trees. On a technical level, the ART pipeline uses a general lexer file which tokenises the input string into lexicalisation, or sequences of tokens. This lexer is combined with a general parser file, which uses these lexicalisations to produce all the different combinations that will be accepted by the context free grammar and the term rewriter, using the rules written in ART's language. This pipeline features choice stages after the lexicalisation process. This allows the lexicaliser to divide the input into modules, which can be used to group rewrite rules and actions into phases. These phases can then be evaluated separately. Not all of the pipeline is used all the time, the direct attribute grammar will not use the term rewriting feature, while the external-to-internal syntax will. A full ART specification is produced by combining the individual ART files upon running and will use the pipeline to read inputs and run pipeline stages using the try command, while also sequencing rule phases through term rewriting in the case of the external-to-internal syntax.

There are two ways of running a program in UKCAD, using the external-to-internal syntax parser, or using the direct attribute grammar. Each method reads the program from the respective .str file written in UKCAD and then translates this program into a form that can be passed to the ValueUserPlugin java file so that it may perform the correct operations. The external-to-internal syntax grammar is located in the UKCAD_Syntax.art file and reads the program from the UKCAD_Syntax.str file. This grammar features statements for parsing from the external syntax, or the UKCAD syntax, to an internal syntax for use with the eSOS rules, and allows for creating shapes, creating subclasses, adding transforms to the subclasses, creating objects, and performing transforms on already created shapes. It features lexical items for strings, integers, and real numbers so that it may correctly identify the given value's data type. One very important statement is for the sequencing of statements, as the purpose is to rewrite the terms to one which can be read by eSOS rules, and therefore must be able to pass all the operations in the program as one statement to the eSOS_CAD.art file.

```
shape ::= sphere^^ | cuboid^^ | cylinder^^ | pyramid^^ | triprism^^ | cone^^ | tetra^^ | trapprism^^
sphere ::= ID '='^ 'sphere'^ '('^ subExpr ')'^
cuboid ::= ID '='^ 'cuboid'^ '('^ subExpr ','^ subExpr ','^ subExpr ')'^
cylinder ::= ID '='^ 'cylinder'^ '('^ subExpr ','^ subExpr ')'^
pyramid ::= ID '='^ 'pyramid'^ '('^ subExpr ','^ subExpr ','^ subExpr ')'^
triprism ::= ID '='^ 'triprism'^ '('^ subExpr ','^ subExpr ','^ subExpr ')'^
cone ::= ID '='^ 'cone'^ '('^ subExpr ','^ subExpr ')'^
tetra ::= ID '='^ 'tetra'^ '('^ subExpr ','^ subExpr ','^ subExpr ')'^
trapprism ::= ID '='^ 'trapprism'^ '('^ subExpr ','^ subExpr ','^ subExpr ','^ subExpr ')'^
```

The external-to-internal syntax statements for each shape, where not created using a subclass.

The eSOS rules feature rules for all the possible operations, including the sequencing of statements, and the language features a batch file, ExtToESOS.bat, which will combine the eSOS rules with a term text file, containing the sequenced term from the external-to-internal syntax. This makes use of ART's !try operation, which is be added to the start of the term. The generated ART file created by the plugin is called by the batch file to evaluate and pass the values to the plugin.

```
-sequenceDone
---
seq(__done, _C), _sig -> _C, _sig

-sequence
_C1, _sig -> _C1P, _sigP
---
seq(_C1, _C2), _sig -> seq(_C1P, _C2), _sigP
```

The eSOS rules for sequencing statements together.

The direct attribute grammar is present in the UKCAD_Interpreter.art file and reads the program written in the UKCAD_Interpreter.str file. Much like the external-to-internal syntax, the attribute grammar features the same rules for operations such as creating shapes or subclasses. However, it uses the iTerms low-level API to pass the values straight from the grammar to the plugin, without the use of eSOS rules. Each value is converted to the relevant "Value" depending on its data type, and the inner value is set to the value in the respective statement. The attribute grammar features the same lexical items and allows for the use of integer and real values, as well as negative numbers. The direct attribute grammar can be ran by calling the parsefx.bat batch file and passing UKCAD_Interpreter as an argument, or by calling the UKCAD.bat file which will run the plugin with a clean slate.

```
statement ::=
  ID '=' 'sphere' '(' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("sphere"), new __string(ID1.v), new __real64(subExpr1.v)); }
| ID '=' 'cuboid' '(' subExpr ',' subExpr ',' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("cuboid"), new __string(ID1.v), new __real64(subExpr1.v),
  new __real64(subExpr2.v), new __real64(subExpr3.v)); }
| ID '=' 'cylinder' '(' subExpr ',' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("cylinder"), new __string(ID1.v), new __real64(subExpr1.v), new __real64(subExpr2.v)); }
| ID '=' 'pyramid' '(' subExpr ',' subExpr ',' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("pyramid"), new __string(ID1.v), new __real64(subExpr1.v),
  new __real64(subExpr2.v), new __real64(subExpr3.v)); }
| ID '=' 'triprism' '(' subExpr ',' subExpr ',' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("triprism"), new __string(ID1.v), new __real64(subExpr1.v),
  new __real64(subExpr2.v), new __real64(subExpr3.v)); }
| ID '=' 'cone' '(' subExpr ',' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("cone"), new __string(ID1.v), new __real64(subExpr1.v), new __real64(subExpr2.v)); }
| ID '=' 'tetra' '(' subExpr ',' subExpr ',' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("tetra"), new __string(ID1.v), new __real64(subExpr1.v), new __real64(subExpr2.v),
  new __real64(subExpr3.v)); }
| ID '=' 'trapprism' '(' subExpr ',' subExpr ',' subExpr ',' subExpr ')'
  { iTerms.valueUserPlugin.user(new __string("trapprism"), new __string(ID1.v), new __real64(subExpr1.v), new __real64(subExpr2.v),
  new __real64(subExpr3.v), new __real64(subExpr4.v)); }
```

The statements in the direct attribute grammar for the direct creation of shapes without the use of subclasses.

The Java plugin file is ValueUserPlugin, and this is where the values from the programs written by the user are passed to, and where the corresponding operations are performed. ValueUserPlugin implements the interface ValueUserPluginInterface which specifies that it must contain and override the name and user methods, the second of which takes the list of Value objects, which contain the arguments passed from the program written by the user. The user method features a switch case statement, which evaluates the first argument in the list, and performs the corresponding method depending on the string value. The string value is unique for each operation and is passed alongside the arguments from either the direct attribute grammar or the external-to-internal syntax.

```
switch(args[0].value().toString()) {
  case "sphere":
  drawSphere(args);
  break;
  case "cuboid":
  drawCuboid(args);
  break;
  case "cylinder":
  drawCylinder(args);
  break;
```

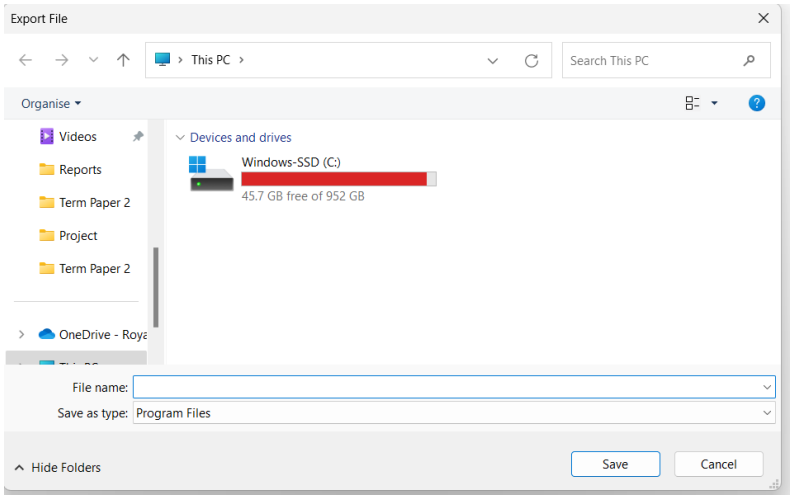
The beginning of the switch statement.

As the user method is always ran whenever a program is ran, the user method also features all of the plugin setup code. The setUpCamera method is called, which creates a PerspectiveCamera object, and three rotate transforms, one for each axis, are added which allow for the camera to be rotated in any direction. A pivot Translate object is also added so that the camera will rotate around the centre point.

```
private PerspectiveCamera setUpCamera(Rotate xRotate, Rotate yRotate, Rotate zRotate) {
  PerspectiveCamera camera = new PerspectiveCamera(true);
  scene.setCamera(camera);
  Translate pivot = new Translate();
  camera.getTransforms().addAll (
          pivot,
          xRotate,
          yRotate,
          zRotate,
          new Translate(0, 0, -700)
  );
  camera.setNearClip(1);
  camera.setFarClip(10000);
  group.getChildren().add(camera);
  return camera;
}
```

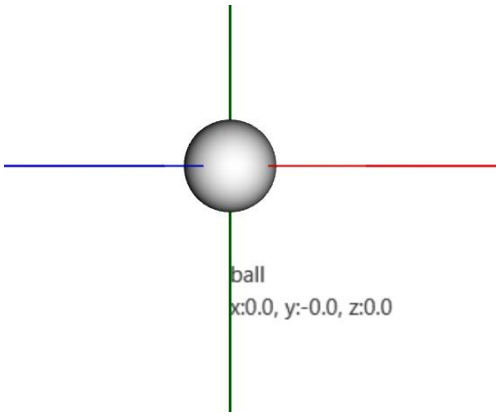
The setUpCamera method, called in the user method.

To allow for having 2D and 3D scenes side-by-side, a system of containers is used. A Group container called root holds both the 3D subscene and the editor, which is displayed within the scene called main. The buildEditor methods creates the editor subscene, creating the objects such as the Buttons and TextArea, specifying both the functionality of the editor as well as the layout, which uses a GridPane to set out the buttons and labels at the top of the editor. A VBox is used to contain that GridPane and the TextArea for the programs to be written in. The editor will display which program file is currently open, and features “Export File”, “Open File” and “Save & Run” buttons which allow for the user to save and open program files for ease of use. UKCAD also features buttons to allow the user to view the specification, signatures and controls of the language and 3D scene within the software itself, whereby the scene will switch to a non-editable TextArea, which displays the respective text file, alongside a back button. UKCAD uses a file called currentFile.txt to store the file path of the program .str file which is currently open across the program being “Save & Run”, so that the plugin can continuously save to that file without the user having to open the file before attempting to “Save & Run” it again. This program .str file is copied to the UKCAD_Interpreter.str file each time the file is opened or ran. Upon the “Save & Run” button being clicked, the currently opened stage will be closed and the reparsefx.bat file will be called, which will run the plugin class file again with the new copy of the UKCAD_Interpreter.str file passed as an argument to ART. The editor also features a “Fullscreen” button, which will enlarge the 3D scene to the size of the monitor.



The export file window, allowing for the user to export a program file.

The 3D scene features a camera which can be moved in x, y and z directions using the WASD and QE keys, can be zoomed in and out using the mouse’s scroll wheel and returned to the starting orientation using the H key. The x, y and z axes can be toggled using the F key, and each shape can be clicked on, which will display its name, as well as its current coordinates within the scene. Only one label displaying a shape’s information can be shown at any one time.



The 3D scene after the F key has been pressed and the sphere shape has been clicked on.

As UKCAD has been built using the JavaFx library, the three default shapes Sphere, Box and Cylinder are present and can be created. However, there are further shapes such as Pyramid and Cone which are created using custom TriangleMesh objects and the MeshView class, where the points, texture coordinates and faces have been manually set up to take a certain number of arguments and create that mesh, which is then passed to the MeshView constructor, a subclass of Shape3D which allows for the custom shapes to be modified in the same way.

The subClass method is used to create custom classes which can be edited and instantiated. This uses the library Javassist, which is used for editing Java class bytecode through the creation of meta-objects and passing strings to insert code. subClass creates a CtClass object, which is a class before it has been loaded, and sets the superclass of this class to the passed existing class, such as Sphere. A CtConstructor object is created which is added to the custom class. Upon the case whereby the superclass will be a MeshView class, such as Pyramid, another custom class is created which is set as the superclass and named Pyramid, which itself will be a subclass of the class MeshView. A HashMap called classTransforms is used to store the respective class's bytecode to be inserted, as this code can only be inserted once and can be added to, so needs to be stored until instantiation. This bytecode will contain an instantiation of the ValueUserPlugin class, and then calling of the respective transform methods with "this" given as the shape to be transformed.

```
classTransforms.put(arg1, value: "ValueUserPlugin plugin = new ValueUserPlugin();");
if (args.length > 3) {
  String arg3 = toInnerString(args[3]);
  Value[] shortArgs = Arrays.copyOfRange(args, from: 4, args.length);
  StringBuilder insert = new StringBuilder("plugin.".concat(arg3.toLowerCase() + "(this, new Value[]{new __real64(0.0),new __real64(0.0),"));
  for (Value arg: shortArgs) {
    insert.append("new __string(\"" + toInnerString(arg) + "\")");
    if (arg != shortArgs[shortArgs.length-1]) {
      insert.append(str: ",");
    }
  }
  insert.append(str: "});");
  String insertString = insert.toString();
  classTransforms.put(arg1, "ValueUserPlugin plugin = new ValueUserPlugin();" + insertString);
}
ctclasses.put(arg1, subClass);
```

The section of subClass that prepares and stores the bytecode to be inserted.

The addTransform method is used to fetch and add to the respective class's bytecode in the HashMap, whereby it adds the new transform and then puts the new bytecode back into the HashMap.

The newObject method is used to finally instantiate the custom class. If this class does not already exist in the classes HashMap, then the CtClass object is fetched and loaded, with the respective bytecode inserted into the CtConstructor.

```
String arg1 = toInnerString(args[1]);
String arg2 = toInnerString(args[2]);
Class<?> cls;
if (classes.containsKey(arg2) != true) {
  CtClass ctcls = ctclasses.get(arg2);
  CtConstructor ctconstruct = ctcls.getConstructors()[0];
  ctconstruct.insertAfter(classTransforms.get(arg2));
  ClassPool pool = ClassPool.getDefault();
  cls = pool.toClass(ctcls, neighbor: ValueUserPlugin.class);
  classes.put(arg2, cls);
} else {
  cls = classes.get(arg2);
```

The section of newObject that loads the class.

After the class has been loaded or fetched, the constructor of the class is reflectively found and invoked, passing the correct arguments and creating the Shape3D object. Upon the case where a subclass of the MeshView class is being instantiated, instead the respective draw method is called which provides a mesh object, which is then passed to the constructor of the custom class to instantiate it.

```
case "javafx.scene.shape.Sphere":
  shape = (Shape3D) cls.getDeclaredConstructor(...parameterTypes: double.class).newInstance(toFloat(args[3]));
  break;
case "javafx.scene.shape.Box":
  shape = (Shape3D) cls.getDeclaredConstructor(...parameterTypes: double.class,double.class,double.class).newInstance(toFloat(args[3])
  ,toFloat(args[4]),toFloat(args[5]));
  break;
case "javafx.scene.shape.Cylinder":
  shape = (Shape3D) cls.getDeclaredConstructor(...parameterTypes: double.class,double.class).newInstance(toFloat(args[3])
  ,toFloat(args[4]));
  break;
```

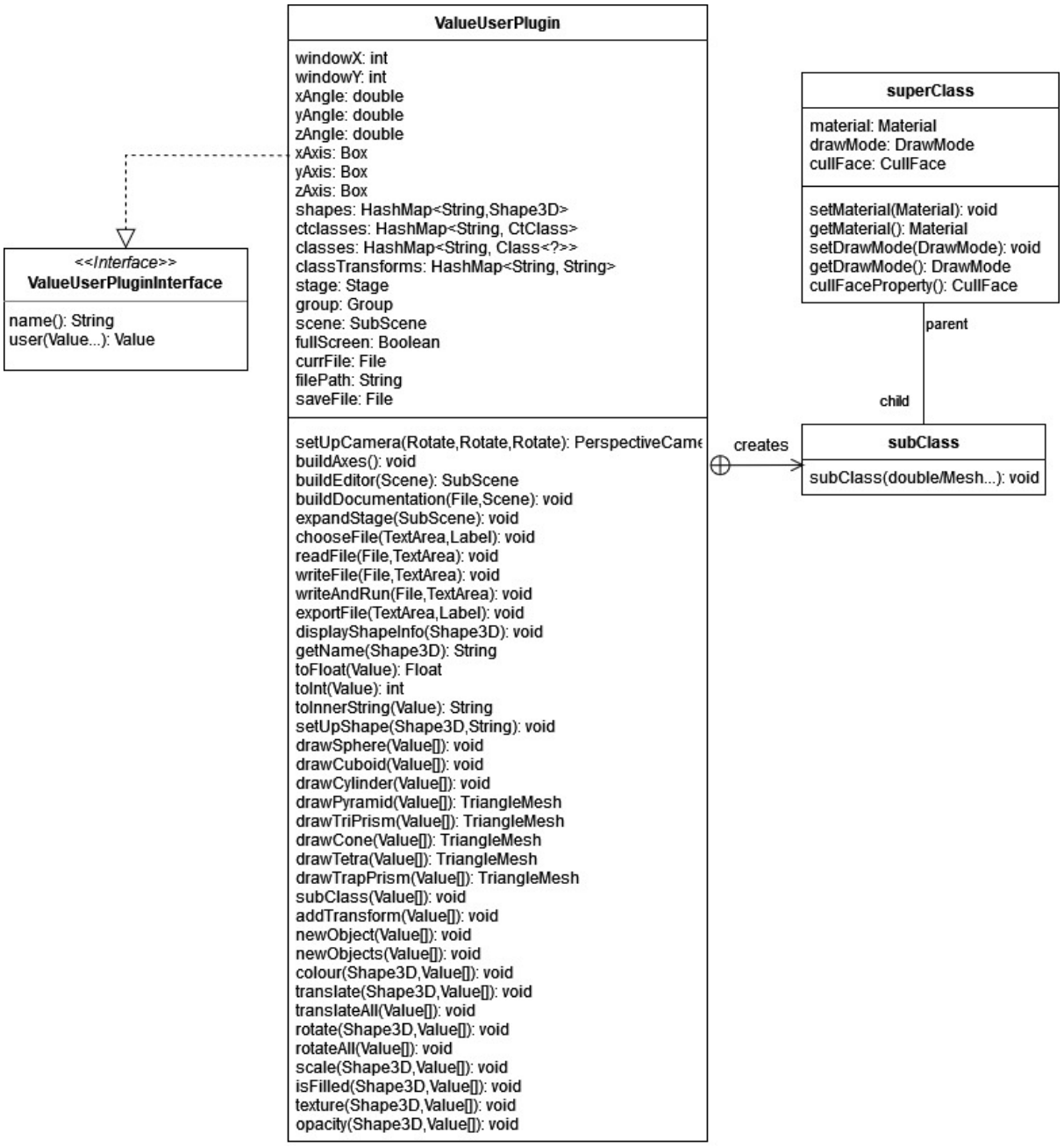
The section of newObject that finds and invokes the constructors of the custom class where either Sphere, Box or Cylinder are the superclass.

UKCAD also features the ability to create multiple objects at once, which is done through the newObjects method. This method simply calls the newObject method a specified number of times, with the ID of the shape being the given value with a numeric value added to it.

```
public void newObjects(Value[] args) throws ARTException {
  Value[] vals = Arrays.copyOfRange(args, from: 1, args.length);
  String arg2 = toInnerString(args[2]);
  for (int i = 1; i <= toInt(args[1]); i++) {
    vals[1] = new __string(arg2 + i);
    newObject(vals);
  }
}
```

Finally, the plugin contains all the transform methods, both which can be applied to one shape, such as colour, or a set of shapes created through newObjects, such as translateAll.

UML


ValueUserPlugin
windowX: int
windowY: int
xAngle: double
yAngle: double
zAngle: double
xAxis: Box
yAxis: Box
zAxis: Box
shapes: HashMap<String,Shape3D>
ctclasses: HashMap<String, CtClass>
classes: HashMap<String, Class<?>>
classTransforms: HashMap<String, String>
stage: Stage
group: Group
scene: SubScene
fullScreen: Boolean
currFile: File
filePath: String
saveFile: File
setUpCamera(Rotate,Rotate,Rotate): PerspectiveCame
buildAxes(): void
buildEditor(Scene): SubScene
buildDocumentation(File,Scene): void
expandStage(SubScene): void
chooseFile(TextArea,Label): void
readFile(File,TextArea): void
writeFile(File,TextArea): void
writeAndRun(File,TextArea): void
exportFile(TextArea,Label): void
displayShapeInfo(Shape3D): void
getName(Shape3D): String
toFloat(Value): Float
toInt(Value): int
toInnerString(Value): String
setUpShape(Shape3D,String): void
drawSphere(Value[]): void
drawCuboid(Value[]): void
drawCylinder(Value[]): void
drawPyramid(Value[]): TriangleMesh
drawTriPrism(Value[]): TriangleMesh
drawCone(Value[]): TriangleMesh
drawTetra(Value[]): TriangleMesh
drawTrapPrism(Value[]): TriangleMesh
subClass(Value[]): void
addTransform(Value[]): void
newObject(Value[]): void
newObjects(Value[]): void
colour(Shape3D,Value[]): void
translate(Shape3D,Value[]): void
translateAll(Value[]): void
rotate(Shape3D,Value[]): void
rotateAll(Value[]): void
scale(Shape3D,Value[]): void
isFilled(Shape3D,Value[]): void
texture(Shape3D,Value[]): void
opacity(Shape3D,Value[]): void
<<Interface>>
ValueUserPluginInterface
name(): String
user(Value...): Value
superClass
material: Material
drawMode: DrawMode
cullFace: CullFace
setMaterial(Material): void
getMaterial(): Material
setDrawMode(DrawMode): void
getDrawMode(): DrawMode
cullFaceProperty(): CullFace
parent
child
creates
subClass
subClass(double/Mesh...): void

TDD

UKCAD contains a Java file called “TestValueUserPlugin”, which contains the JUnit tests for ValueUserPlugin. These tests encompass testing new features before they were added, utilising the Test-Driven Development methodology by creating a test method and adding the assertations expected to hold when the functionality has been created, and then creating the method within the plugin. Not all new features are tested or are indeed testable; features such as exporting a file or displaying a shape’s name and coordinates upon clicking are not tested. However, all new operations with the UKCAD language itself are tested.

As all JavaFx operations are performed on the JavaFx application thread and cannot be performed on any other thread, and as it is required for UKCAD to be in a certain state so that state can be tested, each test features a piece of code to create a new JavaFx application thread to perform operations on. Lists of Value objects are created which are then passed to an instantiated object of the plugin within the thread, which runs the respective methods with the passed arguments. Each test features a “Thread.sleep()” statement, which will pause the thread for either two or three seconds. This is because it may take a short while for the required state to be reached as JavaFx and Javassist operations take time; therefore the assert statements may attempt to test conditions before they have had a chance to be reached.

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        new JFXPanel();
        You, 2 weeks ago | 1 author (You)
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                plugin = new ValueUserPlugin();
                try {
                    plugin.subClass(args);
                    plugin.addTransform(args2);
                } catch (ARTException e) {
                    e.printStackTrace();
                }
            }
        });
    }
});
thread.start();
Thread.sleep(millis:2000);
```

The code to create a new JavaFx application thread and perform the respective operations, featured in each test.

Each test then contains a number of assert statements, which will test whether a certain condition holds. This varies depending on the test; example assert statements are checking if a HashMap contains a key, checking if a superclass is a certain class or checking if an object’s superclass is the custom class.

```
assertTrue(plugin.ctclasses.containsKey(key:"redSphere"));
ClassPool pool = ClassPool.getDefault();
assertEquals(pool.get(Sphere.class.getName()), plugin.ctclasses.get(key:"redSphere").getSuperclass());
assertTrue(plugin.classTransforms.containsKey(key:"redSphere"));
assertFalse(plugin.classes.containsKey(key:"redSphere"));
```

The assertation statements from the testSubClass test.

DESIGN DECISIONS AND RATIONALE

SYNTAX AND GRAMMAR

UKCAD is a three-dimensional Computer-Aided Design (CAD) language. Therefore, all decisions made about the syntax and grammar of the language itself result from the question of how this will be used, or be useful to the user, when creating and modelling their desired object. Each structure and statement was designed to make maximum sense and provide maximum functionality in a CAD language.

This project is based on the ideas of reflective and object-orientated programming. Therefore, although a 3D CAD language provides an obvious form of object-orientated programming, as each shape is an object and has properties, I wanted to go further and add classes as a structure to the language. However, classes as they are found within other languages such as Java are not obviously built for a CAD language. Being able to define a new abstract class that does not represent any 3D shape does not have any use within a CAD language. Therefore, I designed an implementation of classes that made sense; that would both add classes to UKCAD as well as making them useful within the context. This was done partially through the "Garage" proof-of-concept program, where I experimented with creating classes and adding class variables, using the Javassist library to edit class bytecode. I came to the implementation of having each pre-existing shape such as "Sphere" as an existing class, where the user can then create subclasses of these shapes with a specified transform method, such as "Rotate" applied to that class upon instantiation. This made sense in the context of a CAD language as this would allow the user to create a class and apply a transform, then create objects from that class, as opposed to creating many shapes and applying the transforms individually.

UKCAD being a CAD language also influenced the choices around its control structures. UKCAD does not feature any loops such as while or for loops. This is a deliberate choice, as the use of these control structures was analysed in the context of a CAD language, and I found that the main use would be to create or apply transformations to many different shapes. As transformations could be applied to many shapes through the subclassing feature, instead of having loops I decided to implement another statement to allow for the creation of many objects from a class at once, as this provides a natural alternative to loops which uses my subclassing feature. Therefore, as many shapes could be created at once, I also added two transforms, "translateAll" and "rotateAll" which could be applied to all of these shapes at once, or on the "set" of shapes. These methods provide the user with a way to apply a translate or rotate transform to many shapes in one statement, using start and step values.

```
translateAll ::= '['^ ID ']'^ '.translateAll'^ '('^ subExpr ','^ subExpr ','^ subExpr ','^ subExpr ','^ subExpr ','^ subExpr ')'^
rotateAll ::= '['^ ID ']'^ '.rotateAll'^ '('^ subExpr ','^ subExpr ','^ ID ')'^
```

UKCAD also does not feature any if statements, or variables. This is because I found no real use for if statements or variables within a CAD language, as no internal logic needs to be performed, all that matters is the output of the 3D scene. Therefore, no checks or comparisons need to be made to create any programs within UKCAD, and therefore no variables or control structures are needed, instead focussing purely on the 3D shapes and manipulating them.

PLUGIN

The implementation of subclasses within the plugin went through a couple of forms. The initial implementation was very similar to the “Garage” proof-of-concept program, whereby a CtClass object, a pre-loaded class, would be created using Javassist, and a field named after the transform to be added would be created. The arguments for the transform, such as “Red” in the case of “Colour” would be stored in a hashmap under the key, the value of which was the class’s name. Upon instantiation, the class object proper would be created and stored in a classes hashmap, and the class variable field would be set to the respective values in the hashmap. Then, a loop would go through the fields and invoke the respective methods in the plugin reflectively on the object, depending on the name of the field.

This implementation had a few problems, the most important of which was that upon trying to add another transform to the same class, it would overwrite the hashmap entry for that class as they would have the same key value. Therefore, the key value was changed to a concatenation of the class name and the transform name. However, this did not solve the error completely as now the same issue would occur if the user tried to add two transforms of the same type, such as two different rotates. Another issue was that this implementation did not really add code to perform the transforms in the classes, instead adding fields and simply looping through the fields of the class, performing the transforms depending on the field names in the plugin. Another final, small issue was that I did not like the use of a hashmap to store the arguments for the fields until instantiation, as I felt that this implementation did not need to create classes at all to reach the desired output of auto-applied transforms. Therefore, I redesigned the classes feature.

The new implementation for subclasses aimed to instead edit the code of the class to perform the transform without any external help from the plugin. Therefore, I chose an implementation of adding a constructor to the CtClass object, and then inserting code into that constructor to create an instance of the plugin and to perform the respective transform. To this end, I changed each of the transform methods in the plugin to now take the shape to be transformed as well as the other arguments, as opposed to the method finding the shape itself. This was done as now the inserted code could pass “this” as the shape to be transformed, reaching my goal of the class performing the transform on itself upon instantiation. However, I was not able to fulfil my target of removing the hashmap, as this hashmap was needed to store the code to be inserted for each class. This is because when inserting code using Javassist, the inserted code cannot depend upon any previously inserted code, such as the instantiation of the plugin to perform the transform. Therefore, all inserted code needs to be inserted as one block, and since multiple transforms can be added to each class, the block of code must be added to and inserted upon instantiation instead.

```
case "Cone":
  Value[] twoShapeArgs = {new __string(""), args[1], args[3], args[4]};
  mesh = (TriangleMesh) ValueUserPlugin.class.getMethod("draw".concat(cls.getSuperclass().getName()),
   ...parameterTypes:Value[].class).invoke(this, (Object) twoShapeArgs);
  shape = (Shape3D) cls.getDeclaredConstructor(...parameterTypes:Mesh.class).newInstance(mesh);
  break;
```

Code that creates an object reflectively from a subclass of the class “Cone”.

Another decision that was made was to do with the rerunning of a program file. The original design featured an IDE that would constantly check and run the program without the user having to press a “run” button. This design was moved away from to having a more standard “Save & Run” button as this would require differentiating between when a program file has a fully formed statement or not, as well as reparsing the new program file without closing and reopening the window, which I considered outside of the scope, and not the primary focus of this project.

On the side of graphical decisions, the axes were chosen to be togglable as the only aim of the language is to produce a combination of shapes to model a certain object. Therefore, although axes are useful when building the object, upon trying to view it they become unnecessary. The addition of the fullscreen feature also adds to this functionality. The creation of the editor also vastly improves the user experience, as I have experienced through testing the language, as the user is able to edit and rerun programs from within the window itself, as opposed to editing the raw file. The specification, signatures and controls for the language have been added to the window via buttons above the code area, as this allows the user to easily view and consult the syntax or grammar or controls for the language without having to open an external manual. The editor features a label which displays the currently open file, so that the user knows which file is being saved to when the program is rerun, and UKCAD will save this file path to a text file, so that it may preserve this path across the program being rerun, so that the user does not have to re-open the file using the "Open File" button. Finally, the user is able to click on a shape created by a program and display a label that shows the shape's name and current coordinates. This feature is exceptionally useful when creating many objects from a subclass at once, as it can be easy to lose track of which shape is which, and therefore being able to click to view a label solves this problem.

POSSIBLE FUTURE ENHANCEMENTS

UKCAD could be expanded vastly given more time and a greater scope to the project. New shapes using TriangleMeshs could be added, and new transforms, such as an intersection transform whereby two shapes would intersect and the non-intersecting piece would become a new shape, allowing for endless creation of new shapes. However, this transform would require the editing and cutting of TriangleMeshs, which is very difficult and not my priority for this project. The original idea for the IDE, whereby the IDE would wait for a fully-formed statement to be typed and then immediately run it, so that the user may, for example, edit the dimensions of a shape dynamically, could be also added. The camera could be changed so that rather than rotating and zooming around a fixed pivot, could instead zoom into the mouse's location and move the pivot, allowing for more 3D camera freedom for the user.

INSTALLATION INSTRUCTIONS AND REQUIREMENTS

UKCAD requires the libraries JavaFx and Javassist, as well as ART although art.jar is included within UKCAD itself. The versions of these libraries that I used to develop the language, and therefore the versions that are guaranteed to work are JavaFx 17.0.2 and Javassist 3.29.2-GA; the latter can be found here: https://github.com/jboss-javassist/javassist.

Each batch file within UKCAD contains system variables which should point to the respective folder or file, as well as comments explaining where or what they should point to. To run UKCAD using the direct attribute grammar, the "reparsefx.bat" file should be edited to point to the correct locations. Then, the "UKCAD.bat" file can be ran which should use the already created class, lexer and parser files to run UKCAD, using the command "cmd.exe /c UKCAD.bat" in the Windows powershell, or just by clicking on it.

To run the external-to-internal grammar, the "parse.bat" file should be edited to point to the folder containing "art.jar". The file "UKCAD_Syntax.str" should be edited to contain the desired program, and then the command "cmd.exe /c parse.bat UKCAD_Syntax +showAll" should be ran, which parses the program file. Then the "ExtToESOS.bat" file should be edited to point to the "temp" folder, and ran.

"buildPlugin.bat" and "parsefx.bat" only need to be edited and used to recreate the plugin class file or lexer and parser files and can be ran using the same method as the other batch files, with parsefx.bat taking "UKCAD_Interpreter" as an argument. The specification, signatures and controls for the language can be viewed within UKCAD.

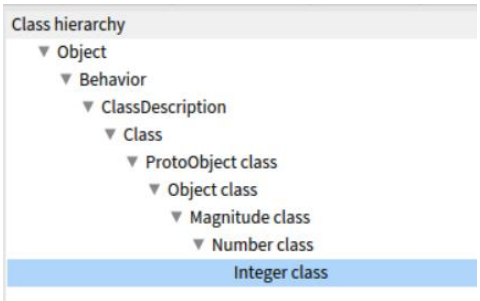
CRITICAL ANALYSIS OF THE PROJECT

This project is focused on the paradigms of object-orientated and reflective programming, both as stand-alone ideas as well as how they can be used in tandem with one another. However, the key aim of this project is that I gain valuable knowledge in how these paradigms are used to both design languages, and how they are applied using those high-level programming languages. This project also aimed to give me vital experience in working individually, planning and achieving goals and designing and putting into place a system which links in with the studied paradigms. This section of the report will analyse whether I managed to achieve these goals, and to reflect on the process of the project, discussing both the successes and difficulties I had.

PROJECT ACHIEVEMENTS

From a theoretical standpoint, one of the project's greatest successes is that I have widely expanded my knowledge of not just object-orientated and reflective programming, but also of programming paradigms and programming in general. I have had a chance to perform in-depth research about the history and implementation of both paradigms, as well as the opportunity to implement these ideas within the context of my own system, thinking about how they can be used to benefit my code.

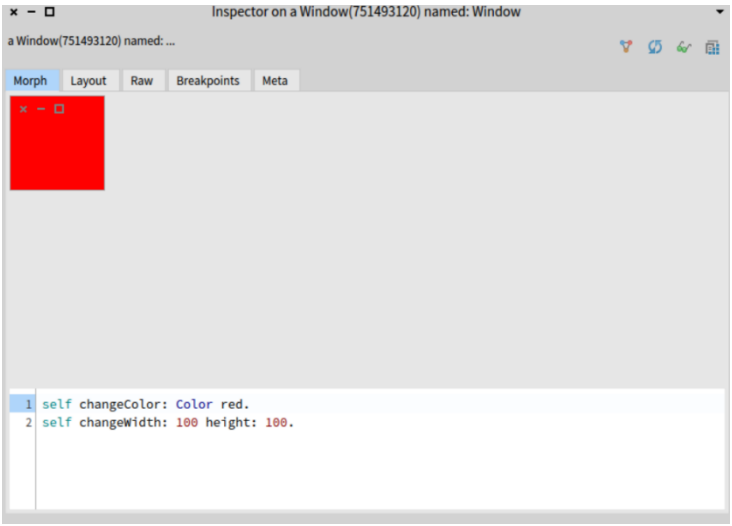
Before this project, I knew of object-orientated programming as I had used structures such as classes in Python, Java and other languages, as well as studying the concepts to a certain degree. However, my understanding was relatively shallow. Therefore, the choice to begin to learn Smalltalk was a great starting point. Smalltalk is quite unique, partially due to its age but also partially due to its design decision to operate around the fundamental idea of everything being an object that can be manipulated by the user, even the meta items. This was fantastic for my understanding as it meant I had to really analyse and consult the documentation before I even thought about starting to code, even though I had a couple of proof-of-concept programs I had planned, as the syntax and grammar alone were unlike anything I had ever seen in a programming language. A great example of this was the differentiation between class variables and instance variables. Before, I had a very limited view of what the difference was in theory; it was not until I had wrongly assigned a handle variable as an instance variable meaning that multiple objects had the same handle did I really learn the difference, as I had to when debugging this issue. The production of the first program, the "Ordering System" in Smalltalk, gave me a clear view of concepts such as inheritance, subclassing and superclassing and message passing, as Pharo provided in many cases graphical representations of these concepts.



Pharo's graphical representation of a class hierarchy.

Reflection as a paradigm was an idea that I had not even heard of before researching for a project topic. Therefore, this means that my understanding of reflection currently is completely a result of this project. Reflection was a concept that took me a while longer to understand in comparison to object-orientation, partially due to reflection covering a lot of different concepts such as intercession and introspection. Therefore, my first action being to learn Smalltalk was also a fantastic choice as Smalltalk is a naturally reflective language as a result of all meta-objects being able to be manipulated by the user, as opposed to other languages such as Java that place certain protections on those meta-objects. Pharo also provided a graphical implementation of reflection, whereby a user can perform operations upon an already running program. This lead to my first program that introduced me to the paradigm, where I created a window object and ran it, and then used the inspectors within Pharo to perform introspection by viewing the internal meta data and intercession by calling methods on that window during runtime and watching the window change.

Pharo inspector on a Window object.



These small programs lead me to create the “Animal Zoo” system, where I used the Java reflection API to create a system simulating a zoo, and used reflective operations to add, remove and manipulate animal objects during runtime. Without the previous knowledge I had gained from the research I had done to understand reflection as it is used in Smalltalk, this program would have been much harder to produce, as many lessons such as concept of meta-objects translated directly over to Java. However, this system was also immensely useful for my final implementation, as well as my understanding of the paradigm as I took onboard how a language that is not fundamentally designed to be reflective handles the dangers of reflection, as it is a greatly powerful paradigm. I began to understand the array of exceptions that need to be handled to perform reflection in Java and other concepts such as setting fields as accessible and non-accessible to prevent creating backdoors into the system.

The final proof-of-concept program I produced was the “Garage” mini-language, which served a triple-purpose; to firstly remind me of how the ART library, plugin and the specification of syntax and grammar of a language works for my final implementation, to use both object-orientated and reflective paradigms whereby classes, subclasses, superclasses and inheritance would feature alongside the use of introspection to find meta-objects such as constructors, and to finally use the library Javassist for the first time to edit and insert bytecode into Java classes for the final implementation. On all three accounts, this language was a huge success as the techniques and in certain cases the exact code was used in my final implementation, and without this mini-language I would not have been as clear-of-thought as to the implementation of UKCAD and also I would have wasted a fair amount of time fixing issues that I had found solutions to through this mini-language. This mini-language was not a part of my original project plan, and in fact was conceptualised during a project meeting.

Use of Javassist to create and add class variables.

```
CtField wls = CtField.make("public static int wheels;", subClass);
subClass.addField(wls);
CtField drs = CtField.make("public static int doors;", subClass);
subClass.addField(drs);
```

During the production of these programs, I also produced two research papers on reflection and object-orientation. These papers gave me the opportunity to research how these paradigms were implemented within Smalltalk and Java, as well as in other languages that I do not know such as C++. These papers were fundamental in establishing my knowledge of these paradigms which directly translated over to producing the programs, while knowledge of terminology and metaobjects translated over from my programs to the research papers and gave me foundation level information to expand upon in my research. Therefore, the research papers also hugely benefited the project in cementing my understanding of the paradigms.

The final implementation of UKCAD was chosen as it gave me an opportunity to work upon an artefact that I had already laid the groundwork for in the previous year, as well as to implement both of the paradigms of the project. Each proof-of-concept program was designed to provide both understanding and solutions to problems that I would encounter during the expanding of this language, and I correctly evaluated this as each was immensely useful. In terms of technical aims, I managed to achieve all except one of my specified aims for UKCAD, whereby I used the Javassist library to add the functionality to create and instantiate classes with edited constructors to perform a transform on a subclass of an existing Shape3D class, using both the reflection and object-orientation paradigms as objects were reflectively instantiated. I provided a basic IDE with features such as exporting files and running the updated program file, provided attributes such as zooming and viewable labels to the 3D scene, and added new shapes and transforms using TriangleMesh objects. On a more general standpoint, I also achieved the aims of using a version control system to manage my own repository, used software engineering techniques such as TDD and the production of UML diagrams, and worked independently to my own specification making informed design decisions.

PROJECT DIFFICULTIES

Overall, I consider the project to be largely successful in terms of expanding my knowledge of the two key paradigms and producing the technical artefacts that I planned to produce. However, there are some difficulties I faced during the course of this project.

The first real difficulty I faced was the understanding of Smalltalk. As discussed, Smalltalk is unlike any other language I have seen and although I did begin to understand concepts such as message passing as I experimented and attempted to debug my programs, the initial experience was very confusing. Smalltalk, and also Pharo, feature an architecture whereby every package containing every class is viewable and this became frustrating as I found navigating the system to be very difficult at the start. However, as I continued I found tools such as the viewable class hierarchy that began to piece the architecture correctly for me and I eventually came to a point where I felt comfortable beginning to produce my own programs. While an initial difficulty, this eventually became an achievement of the project.

Another real difficulty I found was the use of the external-to-internal syntax to parse UKCAD. During the development, I mainly used the direct attribute grammar and designed the editor and language around that implementation. However, this was also partially facilitated by the external-to-internal syntax seemingly producing errors where, for example, a string would be passed as a binary value rather than the string itself. As I began to properly debug and analyse the external-to-internal syntax, whereby I would disconnect the plugin file and let the term rewriting process reach its conclusion so that I could examine the passed values and their underlying classes, I began to formulate solutions to the problems I had which in almost all cases fixed the issues I had, such as the string binary issue.

A key difficulty I had was the inserting of bytecode using Javasisst. Although the insertion of bytecode was not an issue, the issue was that bytecode could not rely on previous bytecode inserted as I found through extensive testing, and that a loaded class could not be edited. Therefore, I needed to redesign the implementation of insertion of bytecode whereby the bytecode would be added to and then inserted at the last possible moment, when the class is loaded and instantiated.

The final real difficulty I had was to do with the specification of TriangleMesh objects. To create new custom shapes in UKCAD, I needed to design and map out these shapes with their bounds, texture coordinates and faces. The bounds and texture coordinates were straightforward, however the faces were not. The direction of a definition of a face using bounds matters, as a clockwise defined face will only show up if it is at the back of a shape when being drawn, which I found out through testing. However, once I knew of this I was able to, generally, be able to create mesh objects that were correctly defined.

PROFESSIONAL ISSUES

As reflection is one of the two key paradigms of this project, it features heavily both in the theory and programming aspects of the project. However, reflection is interesting as it is a very powerful paradigm, allowing the user to perform actions that normally would not be possible, such as searching and using meta objects like a constructor object. Therefore, the use of reflection in a professional setting needs to utilise a cautious approach and be planned well in advance to ensure that the paradigm does not leave the fabricated system with huge security violations for potential attackers. During this project I attempted to emulate some of these protections that are used in the industry to mitigate the security effects of reflection.

As reflection is intrinsic to Smalltalk, and as Smalltalk is not used in any professional setting for software engineering, security concerns do not really apply to Smalltalk in this sense. However, reflection in Java is very much a different case, as Java is a high-level, powerful programming language that is used widely in industry, and is also not a language that inherently champions reflection, only providing an API and limited tools. Java as a language is designed around the idea of security and not allowing users access to meta objects that can cause significant damage in the wrong hands. Therefore, the reflective power that the API provides inherently goes against some of these attributes, such as staples of Java being static typing and encapsulation, both of which can be defeated by reflection by its inherent nature.

Reflection is built around two key ideas, being intercession and introspection, and Java offers introspection, being the ability to examine meta objects, much more than intercession, the ability to change meta objects. During the production of UKCAD as well as the “Animal Zoo” and “Garage” proof-of-concept programs, security when using reflection was a real concern. All of these systems feature or featured use of the “setAccessible” method which allows the user to change a Field meta object to be able to change the internal value for a given instance, or no instance in the case of a static field. This method indicates whether the reflected object can suppress checks for Java language access control, and therefore could induce security problems within a system that is being produced in the industry that has specific security requirements, such as writing code for the Java core for Oracle. In fact, the use of setAccessible always resulting in a security exception, which must be handled. Therefore, whenever using the setAccessible method, I always made sure to a set the field that was set accessible to be set non-accessible at the end of the operations of the program, in the same way as a scanner object is closed to prevent data leakage. It should be noted that UKCAD does not feature the use of setAccessible anymore, as this was a part of the previous iteration of subclassing using Field objects, which needed to be set accessible. UKCAD now uses the editing of constructor objects instead.

The use of reflection in Java also requires the use of exception handling. Every reflective statement such as declaring a new instance of a class object will throws some kind of exception, as well as usually a "SecurityException". Within UKCAD and other proof-of-concept programs, these exceptions are handled via using a try/catch statement that attempts to catch many different types of exceptions, and prints the stack trace of the exception object, as this implementation is suitable for debugging my project. However, in industry a new exception class known as "ReflectiveOperationException" [34] is beginning to be used as it is able to handle exceptions of types that are frequently thrown by reflective operations, such as "NoSuchFieldException". This makes it much easier and more readable to catch many different exceptions are reflection in Java requires but also provides the benefits of individual exception handling as it will handle each type of exception differently [35].

Although the use of reflection within UKCAD and the proof-of-concept programs do not induce many serious security problems, reflection as a paradigm has such power that in certain uses, huge security issues can arise. An example of this is the ability of an attacker to create unexpected control flow paths through a system facilitated by the use of reflection within the application [36]. This weakness can cause a limited form of code injection to be possible, and can be caused by the creation of a so-called "command dispatcher" within the program [37]. The idea is that if an attacker is able to insert values that the application uses to find a class or method, the attacker could create a path to classes that were not intended to be accessible by the developer [38]. This may allow the bypassing of checks or authentication, and if the attacker if able to upload files through this dispatcher, then reflection can be used to add malicious behaviour, such as a backdoor, to the application [39].

An example of a command dispatcher using reflection that could be exploited.

```
String ctl = request.getParameter("ctl");
Class cmdClass = Class.forName(ctl + "Command");
Worker ao = (Worker) cmdClass.newInstance();
ao.doAction(request);
```

This dispatcher above would allow the attacker to instantiate any object that uses the Worker interface [40]. The programmer must remember to modify the dispatcher's access control code whenever a new class is created that implements the Worker interface, as otherwise some Worker classes will not have any access control. However, the default constructor of any object could be called, not just the classes that implement the Worker interface, as although a "ClassCastException" will be thrown, if the constructor itself performs any malicious operations the intended damage will have been done [41]. This scenario is very small scale, however if implemented in a larger, more security-conscious application by an un-informed programmer, this could cause a huge potential security violation.

VIDEO LINKS

https://youtu.be/kWhbAyBdvd0 - Proof-Of-Concept Programs Demo

https://youtu.be/bVL4ygpF2qo - UKCAD Demo

EXAMPLE UKCAD PROGRAM

```
subClass frontWall = Cuboid(Texture, "resources/bricks.jpg")
2 objects: [fwall] = frontWall(125,100,10)
[fwall].translateAll(-87.5,175,50,0,0,0)
object fwall3 = frontWall(300,100,10)
fwall3.translate(0,150,0)
2 objects: [fwalltop] = frontWall(50,50,10)
[fwalltop].translateAll(-125,250,225,0,0,0)
subClass window = Cuboid(Opacity, 0.1)
2 objects: [frontwindow] = window(50,50,5)
[frontwindow].translateAll(-75,150,225,0,0,0)
object fwall4 = frontWall(100,50,10)
fwall4.translate(0,225,0)
object fwall5 = frontWall(300,50,10)
fwall5.translate(0,275,0)
door = cuboid(50,100,5)
door.translate(0,50,0)
door.texture("resources/door.jpg")
handle = sphere(5)
handle.colour(Yellow)
handle.translate(22,50,-5)
subClass otherWall = Cuboid(Texture, "resources/bricks.jpg")
3 objects: [otherWalls] = otherWall(300,300,10)
[otherWalls].rotateAll(90,90,Y)
otherWalls1.translate(150,150,150)
otherWalls2.translate(0,150,300)
otherWalls3.translate(-150,150,150)
floor = cuboid(300,10,300)
floor.translate(0,0,150)
floor.colour(Grey)
ceiling = triprism(300,100,300)
ceiling.colour(White)
ceiling.translate(0,350,150)
subClass roof = Cuboid(Texture, "resources/tiles.jpg")
2 objects: [roofs] = roof(250,15,330)
[roofs].rotateAll(33,114, Z)
roofs1.translate(90,340,150)
roofs2.translate(-90,340,150)
```

PROJECT DIARY

07/10/2022

I downloaded and installed Pharo, and used a tutorial to have a look around the Pharo IDE and to start to get familiar with how it works. I created a Hello World class with a Hello World method, as well as checking out and checking the file back in. I also used the transcript.

10/10/2022

I used the Pharo By Example book to produce a program, a game called Lights Out, which helped me to begin to familiarise myself with how Pharo and Smalltalk work. During the creation of this game I created classes and methods, as well as handled mouse events and produced a graphical window.

12/10/2022

Planned out my first proof-of-concept program, the customer ordering system. Created the Entity and Customer objects, and added getters and setters for all fields, as well as testing they work in the Pharo playground.

13/10/2022

I added the Product class with the fields Name, Product_id and the OrderedCollection Retailers that it is sold from. I changed the orders array to an OrderedCollection as this is a dynamically-sized array compared to the static-sized array that I was previously using. I also added the ability for ids to automatically increment and be assigned once the object is created, so that the user does not have to manually enter ids for each object. I also added SUnit test classes for each method I have written so far and I will test each feature as it is added.

14/10/2022

I added the Order class with fields Address, Order_id, Product_id, Retailer_id and DateTime. I also added getters and setters for each field. DateTime was initialised to take the current date and time to add to an order. I also added the OrderingSystem class which will provide methods to add and remove Customers, Retailers, Orders and Products. OrderingSystem contains arrays for each object so I can call back to previously created objects using the id. I also added tests for the Order class and tests for the methods created so far in the OrderingSystem class. Finally, I added a toString method to the Customer and OrderingSystem classes.

18/10/2022

I added toString methods to the Retailer, Product and Order classes. I also expanded the OrderingSystem class to allow for the adding of Retailers, Products and Orders. The addProduct method adds the id of the product automatically to the field Products for the corresponding Retailer and the addOrder method adds the order_id to the corresponding Customer who placed the order. Finally, I added methods to remove Retailer, Product and Order objects.

20/10/2022

While testing the OrderingSystem class, I found a bug where the instances of the Entity class would save into the respective OrderedSelection via overwriting the previous entry, so that the number of objects was correct but it would output the last instance created that number of times. I also found a bug where the remove methods would not correctly remove the selected item.

21/10/2022

I fixed the repeated entry bug via changing the variables of the Customer, Retailer, Entity, Product, Order and OrderingSystem to instance-based rather than class-based, as the values entered would overwrite the previous instance of that class while they were class-based. I also fixed the remove bug via creating an item variable so that it was not performing operations on the list while iterating through it, and by using = rather than == for checking equality. I also changed the id generation process for every class by using the size of the respective OrderedSelection as the id, as this will increment each time an object is added, rather than each time a new instance is created so that ids will always start from 0.

22/10/2022

I changed the addOrder method and the Order class so that it now takes the id of the Customer, Retailer and Product, and then checks if the retailer, product and customer exist, and if the retailer sells that product; adding transcript prints for error cases. I also fixed the DateTime to be in a form that can be outputted correctly. The address of the order now is fetched and set automatically from the respective customer, if it exists.

25/10/2022

I updated the methods for removing customers, orders, products and retailers to also remove the references in arrays held by instances of those classes. For example, when an order object is removed, it will also remove the reference to it in the relevant customer's orders array. I also added labels into the various toString methods so that it is more obvious what each piece of information that is returned is.

26/10/2022

I started work on the reflective window program, making a new class called Window that is a subclass of SystemWindow. I set it to create a window and inspect on initialisation. I had a play around with changing values such as the bounds during runtime using inspectors. I then added methods to change the bounds while also changing the window size, to change the position as well as changing the colour as the inbuilt change colour method did not work.

27/10/2022

I added a method which creates a button with an argument, and then when this button is clicked, it will change the window's label to the given argument during runtime. My original aim was to have this button be created within the running window, however although the addMorph message adds the button to the window, as shown via the use of the inspector, it will not display the button. I think this is to do with the message being run at runtime. I therefore opened the button in a seperate window and it works.

30/10/2022

I tested the reflective window class, using both the methods I have written and the inherited methods at runtime. I also retroactively added return statements into the ordering system class when a customer, order, product or retailer is added, so that these can be called back giving the user a choice of saving each object into a variable to use later or to use the output methods to find the id of each object.

01/11/2022

I began work on the Java proof-of-concept program, writing the animal classes and a zoo class which contains lists or 'enclosures' for each type of animal. Each animal implements a generic interface which specifies basic getters and setters for shared fields. I will add the reflective elements of the program, allowing for the examining and changing of values, as well as adding new objects during runtime.

03/11/2022

I edited the constructors of the lion, monkey and elephant objects to use setters properly. I also began work on the ReflectiveZoo class by reading through the textbook and implmenting the methods getFields and printFields which will allow me to examine the fields and their values of an object. I also set the monkey object to be accessible, which will allow me to edit its values at runtime by disabling all runtime access checks on uses of the metaobject.

06/11/2022

I chose to split up my report on OOP and reflectivity into two reports, one on both topics. I planned these reports using my own ideas and advice from my supervisor.

08/11/2022

I began work on the first report on OOP, detailing the paradigm and how Smalltalk implements it.

09/11/2022

I continued work on the OOP report, finishing the draft of the first Smalltalk section. I added methods to the reflectiveZoo class which remove accessibility from selected fields and get the values of a given object reflectively. I also began the implementation of a command line interface so that I can perform reflective actions during runtime. Finally, I edited the methods to fetch animals from the respective enclosures correctly.

11/11/2022

I continued work on the OOP report, detailing C++ and how it implements OOP from the C model.

12/11/2022

Continued work on the OOP report, talking about how C# implemented the OOP paradigm and compared this implementation to C++.

13/11/2022

I finished the first draft of the OOP report, finishing with a discussion of generics in Java vs C++, and a conclusion.

16/11/2022

I added methods to the ReflectiveZoo class which add an animal reflectively by locating their constructor and setting the values. I also changed the main function to allow viewing of fields reflectively, by printing them on the screen, and to reflectively change certain fields during runtime. This will be very useful for my final implementation as these are exactly the operations I will need to carry out when editing objects during runtime in the 3D CAD language. I also 'prettied' up my OOP report. Finally, I switched out my if statements for case statements in the command line interface.

18/11/2022

I finished the second draft of the OOP report, adding how I intend on using OOP in the final implementation of the CAD language. I also began work on the reflective report. I changed the ReflectiveZoo class to reflectively call the setter methods when adding an animal. I intentionally used a different way of calling a method reflectively, that is directly, compared to when the fields of an animal are got and then set to a new value reflectively.

19/11/2022

I continued the reflective report, talking about core concepts of reflection. I also began on an implementation of classes using eSOS. During this implementation, I considered my final implementation and I decided to implement a limited form of classes, where each defined class will be a subclass of a shape3D class, holding specific values to make a specific shape. Instances of this defined class will be able to be produced which can be individualised further. However, these subclasses will not be able to have user-defined methods added as all methods needed will be provided by the shape3D superclass.

20/11/2022

I furthered my reflection report, discussing the implementation of the paradigm. I also continued on implementing classes in my plugin, deciding on an implmentation where the subclass will be intialised, taking three parameters which will then update the class variables, so that the class has some fields that are set when instances are created. This will follow the implementation of the CAD software, which will create shape subclasses with class variables of the size of the shape for example.

21/11/2022

I expanded my reflection report, talking about inspectors within Smalltalk. I continued the classes implementation, adding functionality to create an object and to print an object's values to a text file, to check outputs. While testing, I found an error where the classes' class variables are updated each time a subclass is created, rather than the subclasses' class variables.

23/11/2022

I continued with the Classes implementation, and began utilising the library Javassist to create classes and set the Vehicle class as a superclass. I managed to create and assign classes correctly. However, my implementation requires the inheritance of the methods of the superclass, and I intended on having certain class variables that would translate to class variables for the shapes in the final implementation, such as bounds of a shape. After extensive testing and numerous different implementations, I realised that static methods, required to be static to alter the static class variables, cannot be inherited. I then came to the conclusion that my design is impossible to implement due to polymorphic behaviour not being achievable on static methods. Therefore, I redesigned my final implementation to simply create objects of the shape classes, as this is a more natural design, and classes are not needed within my CAD language, as all objects will be created from already existing JavaFx classes, namely Sphere, Cylinder, Cuboid or Mesh. However, this time was not wasted as I delved into classes and the theory, learning a lot, and the attempted implementation caused me to think deeply about my CAD language and the structures needed.

25/11/2022

I finished the first draft of my reflection report, discussing the three different types of reflective methods within Java.

26/11/2022

Began work on the interim report, creating the base structure and plan, as well as importing the two reports written. Also wrote the description of the Ordering System program, detailing its features, TDD and UML and its relation to the greater project.

27/11/2022

Continued work on the interim report, detailing the Reflective Window and Animal Zoo programs, their features and UML diagrams as well as their relation to the greater project. Also added the introduction.

03/12/2022

After the final meeting with my supervisor, I decided to revisit the implementation of classes as this would be a good structure to implement to utilise the OOP paradigm within my CAD language. I changed the implementation to result in the creation of these class variables when the subclass is instantiated, rather than inheriting them from the superclass as a problem arose when changing them, as this would change every subclasses' class variables due to each subclass using the same variables from the superclass. To this end I also changed the Vehicle class to no longer contain these class variables, as they can be added when the subclass is created. Therefore a small language which allows the creation of limited subclasses, where the only difference between these subclasses is the value of these class variables, was created. So far the plugin and the attribute grammar interpreter work and therefore eSOS and the syntax handler must be finished.

04/12/2022

Created presentation detailing the project and the concepts of object-orientated and reflective programming.

06/12/2022

Finished the implementation of classes, adding eSOS rules and cases to the external to internal syntax.

07/12/2022

Finished draft of the interim report.

16/01/2023

Set up the file structure of UKCAD, copied files across from 3rd year project and set up batch files to correctly link to the art files. I tried experimenting with the creation of a maven project, but due to the nature of this project, being ran by batch files and using ART and eSOS tools, the maven project had huge issues with the existence of ART tools such as 'ARTException'. Therefore, I have chosen to use my own file structure and to inline the libraries I will be using in the batch files. For JUnit testing, I will create another batch file which will inline the library and run all the test files.

18/01/2023

I changed the language to use the variable name as the handle for each shape and passed this variable name straight using either the attribute grammar or the external to interal syntax, removing the hashmaps for shapes in each. This is because the former implementation of using a numeric value as a handle and having a different hashmap in both the plugin and the syntax was needlessly complex and caused some issues with the internal to external syntax. The variable is by definition unique and therefore will work as a handle. I also began on the implementation of negative numbers in the external to internal syntax.

19/01/2023

I added functionality to toggle the axes using the key 'F'. I also continued to attempt to write an eSOS rule for negative numbers using the subtract rule, but it is not working yet.

23/01/2023

I added the ability to create pyramids using TriangleMesh and MeshView classes. This object takes three arguments, height, width and depth and can be transformed like the other three shapes. Rules for the external to internal syntax and the attribute grammar have been added.

24/01/2023

I began to implement classes using the library Javassist in much the same way as the in Garage language. The design of these classes will be limited subclasses of the classes Sphere, Box, Cylinder and MeshView, and will contain class variables allowing for instantiation of shapes with these properties.

26/01/2023

I continued the implementation of classes, deciding upon an approach of a subClass specifying a transform (method) and values to go along with the transform, and saving this as a static field or class variable, which can then be read when an object of this subClass is instantiated and this transformation applied to that object. This will provide a function for subclasses to perform, i.e to perform a transformation upon many shapes without having to individually perform it on each shape, utilising the reflective code I have written. I also plan on adding a method to allow for more than one transformation to be added to each subClass.

27/01/2023

Classes have been implemented allowing for the production of instances with a specified transformation applied. External to internal syntax and eSOS rules have been written allowing for either method of running a program using subclasses. Each transformation has been tried and successfully works. This transformation is ran by reflectively invoking the specified transformation method in the plugin, using array comprehension to build an array of Values from the args recieved for the object and the Value[] field added to the subclass to be passed to the relevant transform method. At the moment, the TriangleMesh shapes are not featured as their method of instantiation is more complicated.

28/01/2023

I have began the addTransform method, allowing for multiple transforms to be applied upon object instantiation through the creation of another Value[] field, one for each transform and containing the arguments to be applied. I ran into an issue whereby due to the class being defined in the subClass method, trying to edit and load this class again results in Java attempting to load the class with the same name, which is not possible. A solution I will try is to instead to save the class as a CtClass in the hashmap, rather than loading it and then saving it. This will allow me to edit this class to add another transform as it has not been loaded, and then to load upon object creation.

29/01/2023

I finished the addTransform method, allowing for multiple transformations to be applied per class. This was achieved via creating two new hashmaps, whereby each class would be left as a CtClass and added to a hashmap, which can then be retrieved and a new field (transform) can be added. This bypasses the error of only being able to load the class once, as a class is now loaded upon the first instantiation of an object. I ran into another error where the field cannot be set as the needed Value[] through Javasssist; I therefore solved this by creating another hashmap which stores the Value[] for the field, and then upon loading the class, fetches the Value[] from the hashmap and reflectively sets the field(s) object(s).

30/01/2023

I noticed a bug with the rotate method, which was written last year, whereby it would incorrectly calculate the centre point of the shape using the bounds. I changed this to instead use the translate X,Y and Z values as the pivot point. I also added an argument to allow the specification of which axis to be rotated around. Finally, I added functionality to use the pyramid when creating a subclass, however due to TriangleMesh being unlike the defined shapes which have a set constructor as a mesh requires specific points and faces, so the newObject function simply sets the values to be the same as when a pyramid is generated normally. Later on I will implement a method to create more generic TriangleMeshs, and the subClass function will be changed accordingly.

31/01/2023

I began the implementation of an editor into UKCAD, allowing for viewing and editing of a program. My original design was to have an editor alongside the scene, but this was difficult to implement due to JavaFx only allowing the showing of one scene per stage at any time. Therefore, I decided on an implementation of a key (X) being used to swap to the editor scene, which displays the code written in UKCAD_Interpreter.str and features two buttons; one to switch back to the scene and one to save and run the changed program. At the moment, the save and run button saves the changed program to the file and runs a new process in a separate window through running the batch file. This will be changed to run the new program within the same window.

01/02/2023

I added a button to allow for the importing of a program file to the editor, which can then be saved to the main program file (UKCAD_Interpreter.str) and ran. I also added a button to export the code in the TextArea object to either a new or existing file. I plan on adding the ability to save the changed code to the original file as well as UKCAD_Interpreter.str when ran.

04/02/2023

I added the ability to save the changed code to the original file via saving the file as a variable and then upon the save & run button being clicked, it will attempt to write the text in the codeArea to the file. Upon calling the write method, it will check if the file exists or not. I also added a label to display which file has been opened which will update upon selecting a file. Finally, I changed the editor to instead be present alongside the 3D scene rather than the X key having to be pressed to swap over to the editor. This was done through the use of subscenes, whereby the 3D scene and the editor are subscenes and part of a root group, which is added to a main scene. This means that the user no long has to switch between them and has the editor at the side, following my original design.

05/02/2023

I added a new batch file (reparsefx.bat) that only contains the command to run the classfile of the plugin with the input program file. This reduces the time taken upon running a program as now the parser and lexer do not have to be generated each time a program is reran within the editor. I also added functionality to save the file path of the opened file within a .txt file, which can then be read and the currFile variable set so that upon running the new program with the editor, it will save where this file is so that it does not need to be opened again to save the file upon running.

07/02/2023

I went through much of my code, finding inefficiencies and changing them, such as the repeated use of string addition, which I have replaced with the concat method. I also added the ability to display a label, showing the name and the coordinates of a shape. At the moment, the implementation is through a key press, which displays the labels for all shapes. This will be changed to when the specific shape is clicked, meaning only one label can be shown at any one time.

08/02/2023

I edited the label for shapes to now display upon the clicking of a shape, where only one label can be shown at any one time. This label will rotate to the angle of the camera and can be rotated again by clicking on the shape at a different angle. These labels will help the user to see which objects are which and where they are located.

09/02/2023

I added an expand window button, which upon being clicked will hide the editor subscene and set the 3D scene to be the dimensions of the display. The stage will then be fullscreened. The escape key can be used to exit fullscreen, which will reverse the actions taken. This feature was implemented for the user, as the language's purpose is to model shapes, and therefore to be able to see the shapes in fullscreen is a useful option.

10/02/2023

I changed the labels for the shapes to now hide the label if the same shape is clicked again, after it has been clicked to display its label. This adds the ability for the user to view the label and then hide it. I also added the ability to be able to zoom in and out of the 3D scene, using the mousewheel. This was implemented through translating the camera along the z axis in the direction that has been scrolled along the y axis, i.e forward or backward. Finally, I added a new transform: opacity.

11/02/2023

I added the ability for float numbers to be used within a program. I also added a new operation, allowing for the creation of many shapes at once. This is performed using the subClasses, whereby a subClass is created, and then a variable number of objects can be created, which are assigned using the given ID concatenated with a numeric value. This newObjects method calls the newObject method x number of times, passing its arguments with the object ID value changed to be the ID + numeric value. To this end, I also created rules for a subClass with no arguments, i.e no transforms to be applied, therefore allowing for the creation of many basic shapes. Finally, I noticed a bug where if two subClasses had the same transform to be applied, the second would overwrite the first. This was caused by the fieldVals hashmap using the name of the transform as a key, which would be overwritten to the last subClass with that transform. I therefore changed the hashmap to now use the subClass name concatenated with the transform name as a key.

13/02/2023

I added a new transform, allowing for sets of objects, denoted with [name] and created using the newObjects method, to be translated using a starting point and step value for each coordinate. Within the plugin, it will search the hashmap for all shapes with name "name" + i, where i is a numeric value, and apply the relevant translation. This will allow the user to create a line of objects easily, without having to translate each shape. I also began to implement a new shape, a triangular prism, using triangleMesh. At the moment it is missing some triangles, but I have drawn out each face and point on paper and will implement the shape correctly.

14/02/2023

I added a new shape: a triangular prism. This shape is specified by x,y,z arguments and is created using the triangleMesh class. I had a problem with my initial faces as I did not realise that the order of points for a face is significant, determining which way it is facing and therefore resulted in a shape with gaps. However I correctly drew out my faces and implemented the shape. I also changed the subClass method to accommodate the new prism, where if a shape using triangleMesh is used to create a subClass, a new class is created, at the moment either Pyramid or TriangularPrism, which has the superclass MeshView and itself is used to be a superclass to the created class. This implementation was used as upon the creation of an object, the MeshView constructor cannot be simply called like Sphere, Box and Cylinder as it must take a list of points and faces. Therefore, the method for creating the respective shape will be called reflectively, passing the correct arguments to create the shape desired. This implementation allows me to easily implement more triangleMesh shapes and use them as subClasses.

15/02/2023

I added texture coordinates to both pyramid and triprism shapes; now they will be able to display a texture correctly.

17/02/2023

I added two new shapes: cone and tetrahedron. I found a library to create complex shapes in JavaFx and I used some of the source code, changing it to fit my plugin and adding the correct texture coordinates. I also added cases for cone and tetrahedron shapes being used as subClasses. Finally, I changed the attribute grammar to now take float values as well as integer values for all rules.

18/02/2023

I changed the tetrahedron shape to now take x,y,z as arguments. I did this as the implementation that I took from the library produced a tetrahedron that was not facing the correct way, so rather than rotate it I traced out points and faces and implemented my own. I also added two buttons to view the specification and signatures from the editor, as this will help the user to use the language while producing a program as they can lookup how to perform a certain operation. While doing this, I changed the editor to now use a GridPane wrapped in a VBox so that I can position nodes more precisely.

21/02/2023

I added a new shape: a trapezium prism. This shape takes x,y,z arguments. At the moment there is a graphical bug to with two faces. I believe this is a result of an incorrectly assigned face, which will be fixed. I also continued my final report, by expanding the introduction of my interim report.

22/02/2023

I fixed the issue with TrapPrism, the issue was two points were incorrectly assigned to -bx when they should have been bx. I also changed TrapPrism to now take the arguments bx, tx, y, z; where bx and tx are the bottom and top x values on the trapezium.

23/02/2023

I changed the batch files so that now class files of both the plugin and the generated parser and lexer are compiled into the "classes" folder and ran correctly. This cleans up the main directory. I also added a section in my final report on analysing existing 3D CAD software and languages, and began to write about the UKCAD language.

27/02/2023

I implemented a different method of applying transforms to classes. Before, a subClass would be defined and then the transform held in the fieldVals hashmap, when upon instantiation, the field would be set to the corresponding value in the hashmap. This implementation consisted of using a field to store the Value array which would be read upon creation. This made little sense as it required the use of an addition hashmap and did not really use the class to instantiate the object in the case of triangle mesh shapes. Also, the fieldVals hashmap used a concatenation of the class name and transform name to store the values, meaning more than one transformation of the same type could not be applied to the class. This is now changed to run the corresponding method for transformation in the custom subClass's constructor. Each subClass is now a part of the package main, which allows it to take the method from the plugin. Each transform method has now been changed to take the shape as an argument, so that the subClasses can call 'this' as the shape argument. This solves many issues with the program, such as now the transformation is applied once in the constructor rather than having to search the fields each time an object is created. Each triangle mesh shape now returns the mesh object, which is used for the subClass which takes a mesh object, due to the subClass inheriting from the MeshView class.

01/03/2023

I found and fixed errors to do with the new method of adding a transform to a subClass's constructor. This was mainly due to inserting code using Javassist, as code needs to be inserted as one block to be able to reference objects created within that block. Inserting code in multiple stages and attempting to reference an object previously created will not work. Therefore, I chose to use a hashmap to store the code to be inserted and append to it each time a transform is added. Then, upon first-time object instantiation, the relevant code will be inserted into the constructor, which will call the methods needed to provide the transform.

08/03/2023

I added a new transform: rotateAll, which can be used upon sets of objects much like translateAll. This method takes a starting angle, a step angle and an axis to rotate around. Then it will rotate each object adding the step angle to the start angle i-1 times, where i is the index of the shape. This allows users to, for example, create a hollow box out of four cuboids via rotating each cuboid 90 degrees further, creating a hollow box shape.

09/03/2023

I continued to write about UKCAD in the final report, detailing the attribute grammar and external-to-internal syntax.

13/03/2023

I added a new batch file: UKCAD.bat that will run the direct attribute grammar parser, so the user can just click to run UKCAD without needing to use any commands. I also tidied up the batch files so that now they use environment variables rather than the raw path to the files, so that UKCAD can be ported over to another system more easily; only the environment variables need to be changed. Finally, I fixed an error so that the external-to-internal syntax now looks for the class files in the classes folder.

14/03/2023

I discovered an error whereby the external-to-internal syntax would not pass a string value correctly. Instead of setting the inner value of the Value object to the string, it would set the object name to the string. Therefore, I added a method in the plugin to look for if the type starts with an underscore character, and then either return the inner value or the object name.

16/03/2023

I continued writing my final report, finishing the section on the overview of the architecture and features of UKCAD. I also moved the section of code that sets up the camera into its own method, and I noticed a bug whereby if a MeshView class was used to create a subClass twice, the custom superclass will be created twice, which cannot be performed. Therefore, I added an if statement to check if this superclass already exists, and to not create it if it does. I added a new button to the editor, which will allow for the user to view the controls for the 3D scene. Finally, I added the lexical item REAL to the external-to-internal syntax, allowing for the use of decimal values.

19/03/2023

I changed the UKCAD.bat file to now reparsefx.bat instead of parsefx.bat, as parsefx.bat creates the parser and lexer files, which if the grammar or syntax for the language have not been changed, is not needed and is therefore slowing UKCAD down. Therefore, reparsefx.bat, the file used to rerun a program, simply runs the new .str file which the plugin class file. I also added the sections in the final report on the design decisions of UKCAD.

20/03/2023

I added the sections on professional issues and critical analysis of the project to the project report.

21/03/2023

I added JavaDoc comments to the plugin and test classes, added the project structure to ReadMe.txt and updated the signatures, controls and specification.

22/03/2023

I found and fixed two errors with the external-to-internal syntax, being the passing of the url for the texture transform, and the passing of negative numbers. This was done via using the &STRING_DQ structure and adding eSOS rules.

23/03/2023

I recorded and uploaded the demo video of UKCAD. I also finished the final draft of the report.

BIBLIOGRAPHY

[1]: Holmevik, J. R. (1994). Compiling Simula: A historical study of technological genesis. *IEEE Annals of the History of Computing*, *16*(4), 25-37.

[2,3,4,5,7,8,9,10]: Rentsch, T. (1982). Object oriented programming. *ACM Sigplan Notices*, *17*(9), 51-57.

[11]: *Sigplan Notices*, *17*(9), 51-57.Ingalls, D (1978). "The Smalltalk-76 Programming System: Design and Implementation," 5th Annual ACM Symposium on Principles of Programming Languages.

[6]: Ducasse, S. (1999). Evaluating message passing control techniques in Smalltalk. *Journal of Object Oriented Programming*, *12*, 39-50.

[12]: Dubois-Pelerin, Y., & Zimmermann, T. (1993). Object-oriented finite element programming: III. An efficient implementation in C++. *Computer Methods in Applied Mechanics and Engineering*, *108*(1-2), 165-183.

[13,14]: Stroustrup, B. (1995, October). Why C++ is not just an object-oriented programming language. In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)* (pp. 1-13).

[15]: Brunner, T., Pataki, N., & Porkoláb, Z. (2016). Backward compatibility violations and their detection in C++ legacy code using static analysis. *Acta Electrotechnica et Informatica*, *16*(2), 12-19.

[16,17]: Clark, D., & Sanders, J. (2011). *Beginning C# object-oriented programming* (Vol. 1). np: Apress.

[18,19,20,21]: Ghosh, D. (2004). Generics in Java and C++ a comparative model. *ACM SIGPLAN Notices*, *39*(5), 40-47.

[22]: Smith, B. C. (1982). *Procedural reflection in programming languages* (Doctoral dissertation, Massachusetts Institute of Technology).

[23,24,25,26,27,28]: Forman, I. R., & Forman, N. (2004). *Java reflection in action (in action series)*. Manning Publications Co..

[29,30,31,32,33]: Li, Y., Tan, T., & Xue, J. (2019). Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *28*(2), 1-50.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.

Adrian Johnstone. (2021). Software Language Engineering.

Ducasse, S., Rakic, G., Kaplar, S., & Ducasse, Q. (2022). *Pharo 9 by example*. BoD-Books on Demand.

[34,35]: Dustin Marx. (2011). JDK 7: Reflection Exception Handling with ReflectiveOperationException and Multi-Catch. https://www.infoworld.com/article/2074084/jdk-7--reflection-exception-handling-with-reflectiveoperationexception-and-multi-catch.html

[36,37,38,39,40,41]: OWASP. (2016). Unsafe Use of Reflection. https://owasp.org/www-community/vulnerabilities/Unsafe_use_of_Reflection