OBJECT-ORIENTATED AND REFLECTIVE PROGRAMMING

THOMAS WILKINSON

COMPUTER SCIENCE MSCI FINAL YEAR PROJECT

DEPARTMENT OF COMPUTER SCIENCE

ROYAL HOLLOWAY, UNIVERSITY OF LONDON

TABLE OF CONTENTS

INTRODUCTION

PROJECT AIMS AND RATIONALE

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 programming languages being so commonplace. Reflectivity, while not quite as widely utilised, is also a fantastic paradigm to have a deeper understanding for, due to society relying more and more on dynamic systems that must react to the situation at hand. I will also acquire vital general experience in producing a project, using a version control system 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.

On a technical standpoint, the objective of this project is to produce an object-orientated and reflective 3D CAD language, which will allow for the creation and augmentation of shapes, utilising the JavaFx library to write the backend plugin. The language will use an attribute grammar interpreter, as well as eSOS rules and a syntax specification to format the front-end syntax and call the correct operations with the plugin.

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.

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.

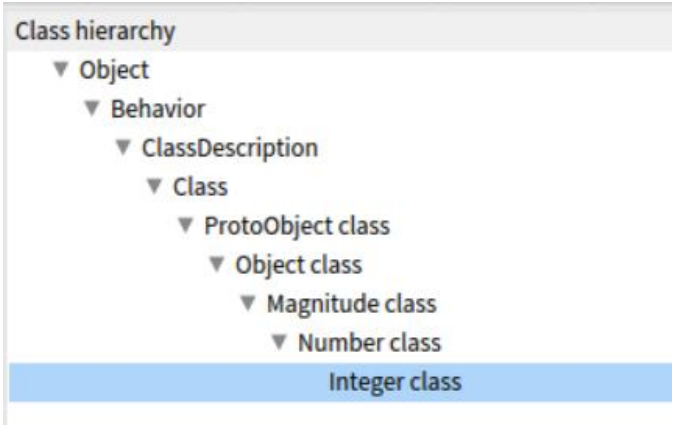
EARLY HISTORY

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 an '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

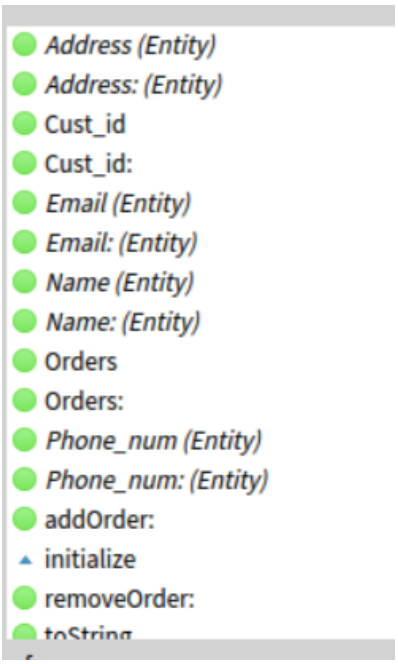
It is not enough to just have objects though; for events to happen, processes must take place. Within Smalltalk, this happens inside the objects themselves as they provide their own computation. Objects must also communicate with each other, as internal computation can only take a program so far. This 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]. This is well-shown by the fact that C# uses no access modifiers, inheritance is always public. 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

All 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;
```

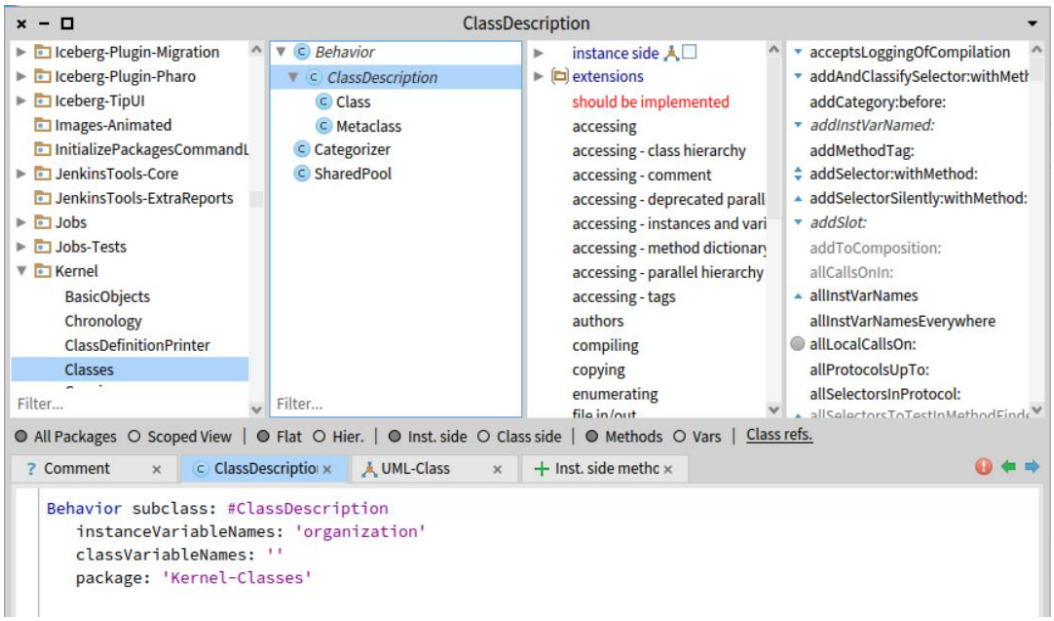
A method which obtains the values of all an object’s fields, both declared and inherited. This is paramount for serialization as it must access all information about an object.

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.

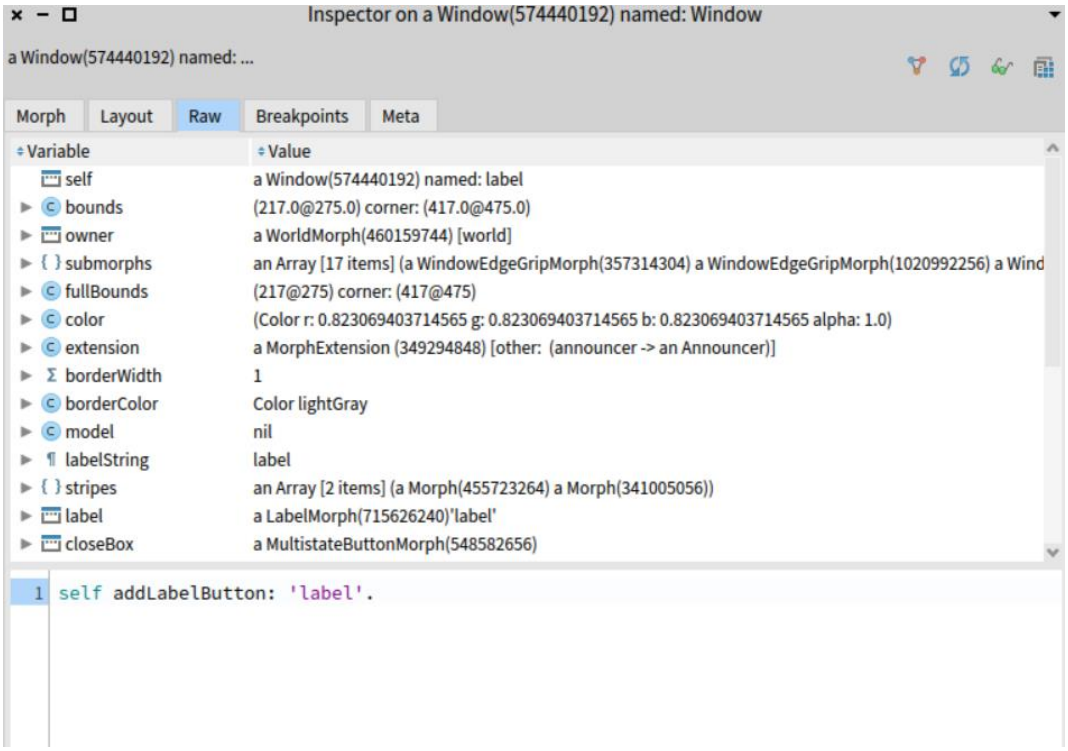
As metaclasses are a higher level than a normal class, an inevitable recursive query could be to the existence of a "metametaclass", or a class which defines the properties of a metaclass. However, these classes are not featured as they are not necessary, made possible by the fact that metaclasses can 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.

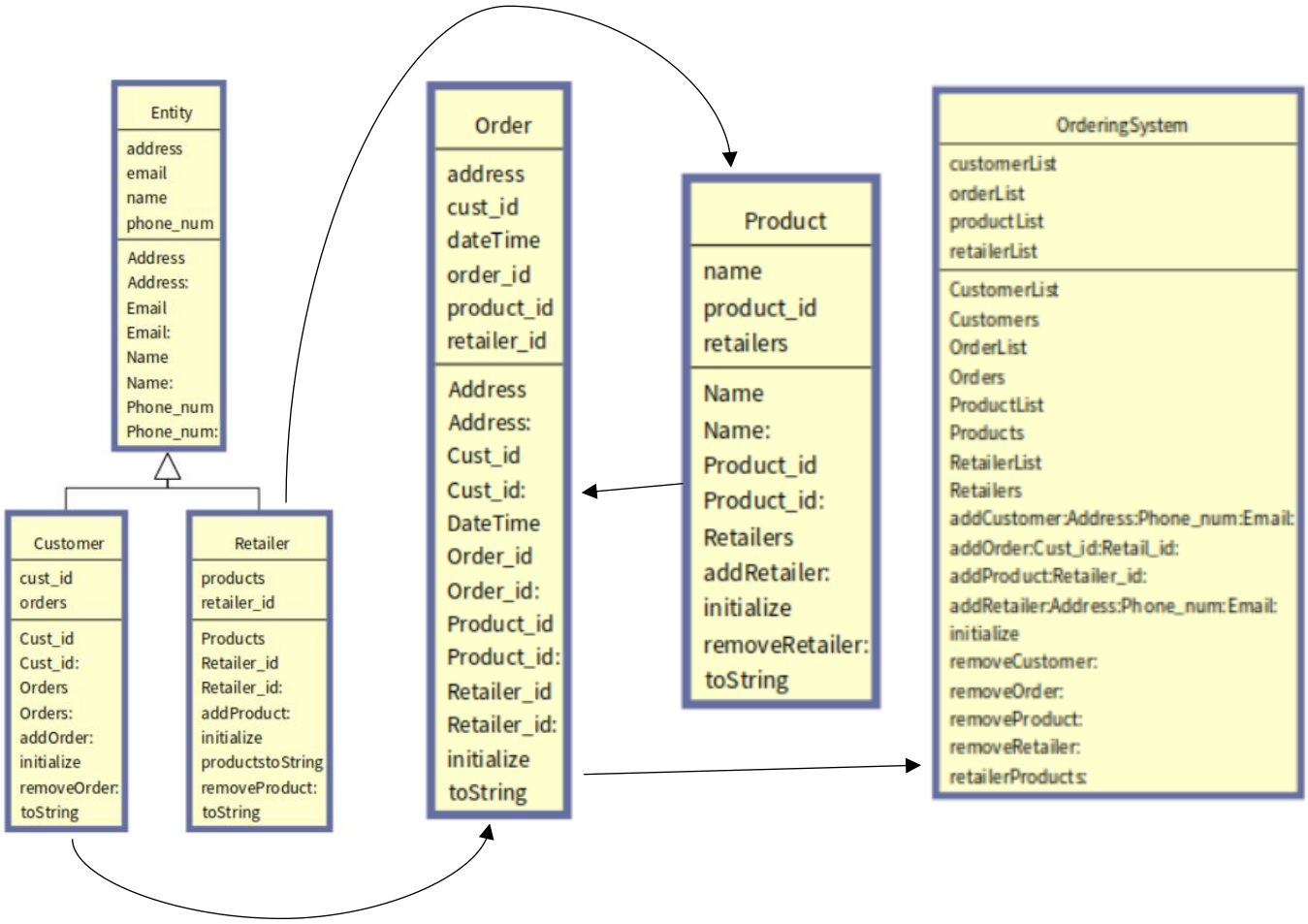
```
removeOrder: id
   | item |
   orderList do: [ :order | order Order_id = id ifTrue: [ item := order ] ].
   item ifNotNil: [ orderList remove: item ] ifNil: [ Transcript show: 'Order not found'; cr ]. "Checks if order exists and removes it"
   item ifNotNil: [ customerList do: [ :customer | customer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

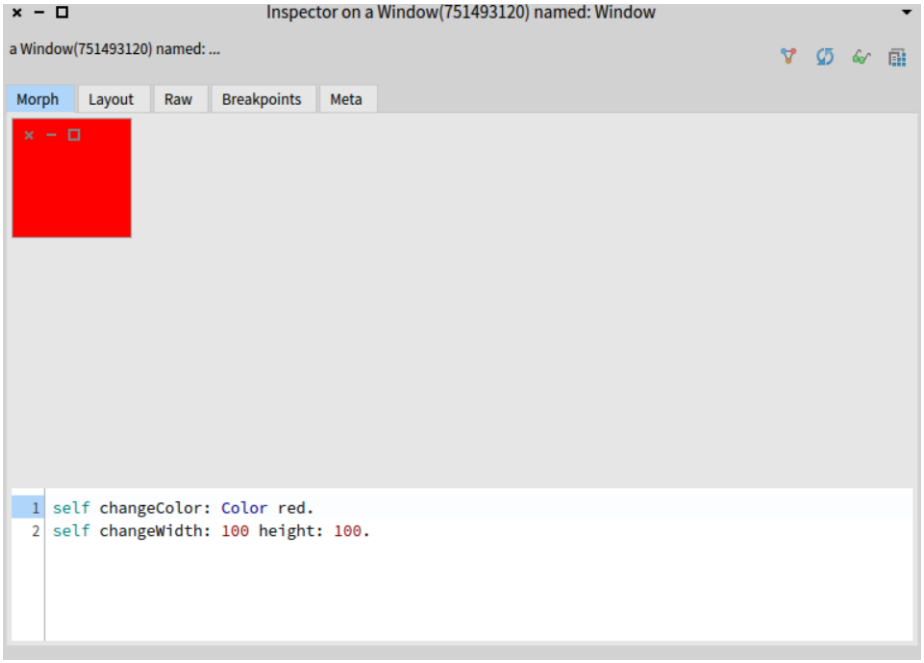
The program consists of one class, “Window”, that is a subclass of the Smalltalk class “SystemWindow”, which allows for the creation and augmentation of a window in the Pharo universe.

Window’s initialize message

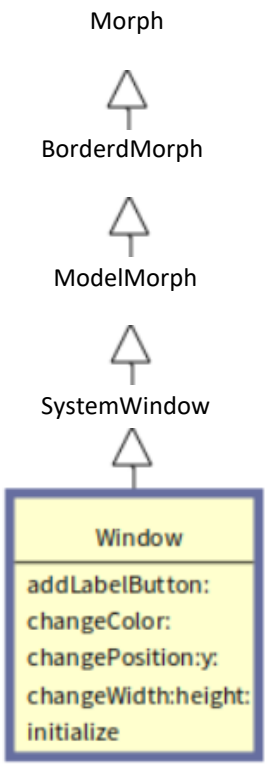
```
initialize
    super initialize.
    self setLabel: 'Window'.
    bounds := 0 @ 0 corner: 200 @ 200.
    self beResizeable.
    self openInWorld.
    self inspect.
```

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;
```

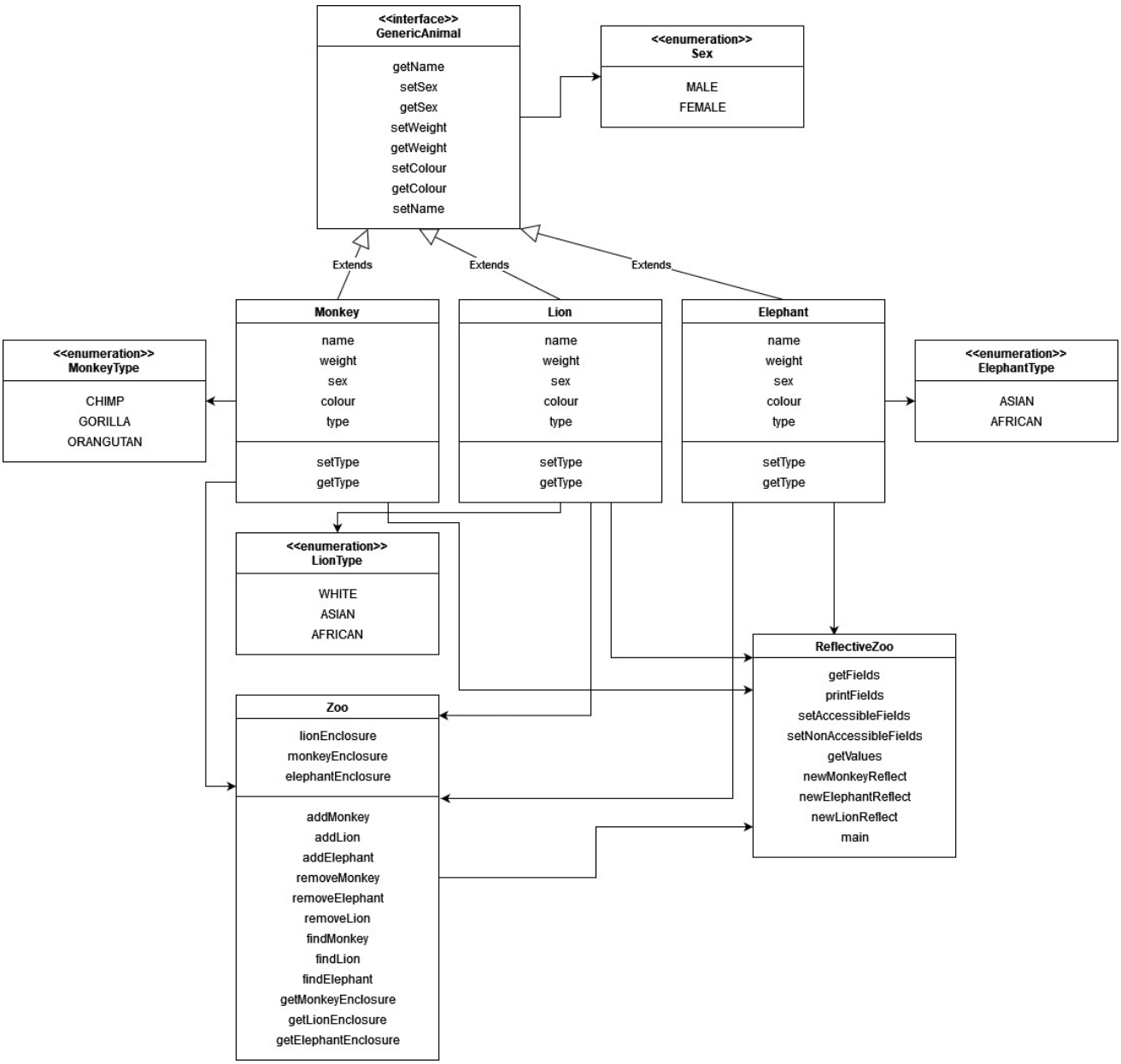
The “newMonkeyReflect” method, which reflectively creates a monkey object. A different implementation may have used the constructor to assign the values, but I chose to invoke the setter methods and to use an empty constructor to gain practise invoking a method at runtime.

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”.

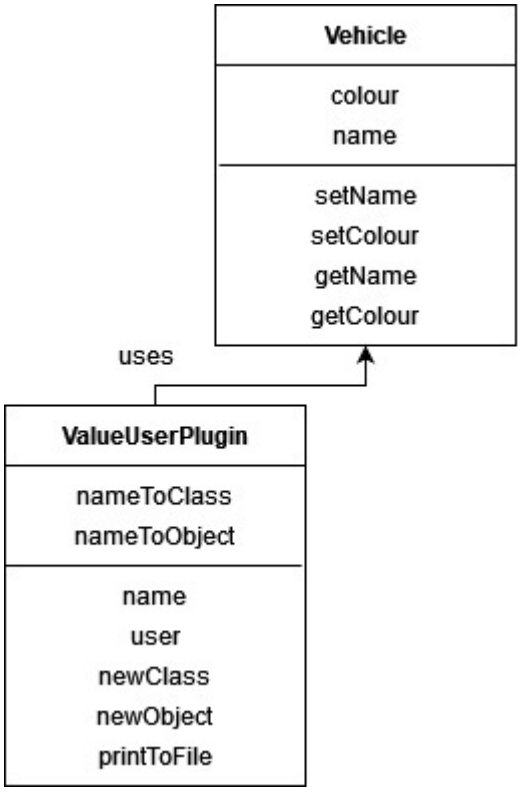
```
Class cls = obj.getClass();
Field doors = cls.getField(name: "doors");
Field wheels = cls.getField(name: "wheels");
Field name = cls.getField(name: "name");
Field colour = cls.getField(name: "colour");
```

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



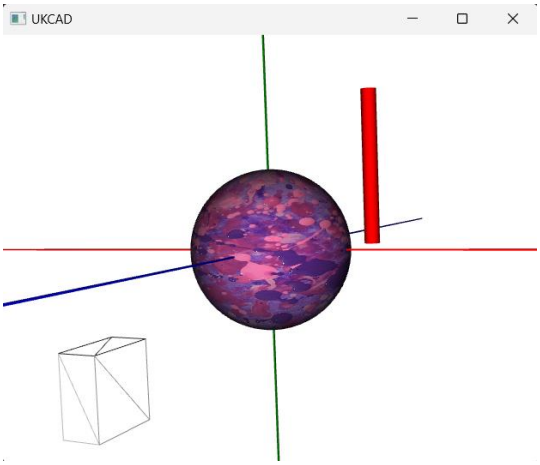
RELATION TO PROJECT

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

CURRENT ARCHITECTURE AND FEATURES

The current version of UKCAD was created during the third-year Software Language Engineering module, and features eSOS rules, an external to internal syntax, an attribute grammar and a backend plugin. It provides functionality to create one of JavaFx's Shape3D shapes, being "Sphere", "Cuboid" or "Cylinder", which can be sized using the passed arguments, and then coloured, translated, rotated, scaled, filled or textured using a texture file. The plugin also creates the 3D space and axes and features a camera which can move in three dimensions using the keyboard. The language also allows for certain control structures and logical operations where appropriate to a CAD language, such as if statements, variables, mathematical operations and while loops.



An example output

PLANNED FEATURES

The existing UKCAD language will be heavily extended in the final implementation. Custom objects of any shape will be able to be added using the JavaFx "mesh" class, as well as subclasses, which will allow the user to define a shape's bounds and other properties as a subclass using the class variables, and then use this class as a "factory" for further augmented instances of this shape. These subclasses will use the JavaFx classes to inherit from. Each operation will be redesigned so that it can be performed reflectively, allowing for the addition of an editor alongside the window, so that the user can dynamically change the program during runtime and the scene will change as soon as the new line is written. This editor may feature some sort of IDE-like features, such as a terminal for outputs, alongside operations to print to this terminal. New operations will be added, using the JavaFx methods, allowing for further augmentation of shapes.

RELATION TO PROJECT CONCEPTS

A 3D CAD language provides a natural and obvious way of showcasing object-orientated programming, as the objects are 3D shapes and the properties make a visual difference. Therefore, although object-orientated programming is already present, subclasses of existing shape classes in a limited form will be added; this design was chosen as it seemed the most useful implementation of classes within a CAD language, as a class that does not represent a shape in a CAD language has no use. Reflection will be ensured by the changing of operations to be reflective, as a CAD language is much more intuitive if dynamic; if the user has to rerun the program to keep changing one value it can become frustrating. This led to the editor extension, whereby if operations are reflective then an alongside editor will be very helpful to the user.

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.

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

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.