Reviewing Object Oriented Programming within Empirical Modelling

1018739

Abstract

This paper shall begin by examining the challenges that are faced by EDEN and how object oriented principles could be potentially used to solve them. It shall then review the key concepts of object oriented programming and how this could be applied to modelling construals. Next it shall present an implementation that attempts to combine EDEN with Java's object orientation. Finally it shall review the outcome of the implementation, what it has achieved and where its problems lie.

1 Introduction

Objects within EDEN and the functionality they provide have been discussed by many people to date. There are a number of different notations and tools to try and encapsulate one or more of the features that objects provide. This paper shall discuss these features and how they could solve a number of existing challenges when modelling complex construals.

2 EDEN

The EDEN language provides a mixture of definitive scripting along with procedural code that can be used for modelling. While it is able to model a wide variety of complex construals there are still a number of challenges that are faced that provide a good basis to discuss the inclusion of objects within EDEN .

2.1 Objects in JS-EDEN

JS-EDEN is one of the tools that can be used to model construals using the EDEN language. It currently includes limited support for objects due its implementation in JavaScript (JS). It allows to define interactive objects using the example definition below and its properties may be accessed using the dot notation. While this does allow to reassign these values, it causes the definition to become an equals definition, i.e. the dependency is no longer updated as its dependencies update. More importantly however is that the user is not able to create an is dependency for a property after they have been initially created. For example, when defining a circle, you can reference position of x to depend on an observable, for example the position of the users mouse.

```
circle is Circle(mouseX, mouseY, 100);
circle.x is mouseX + 100;
```

The user may want to later change the position of x to be offset by some value or depend on a different formula as above. This use of dot notation is currently not supported and limits the advantages that objects in JS-EDEN provides. The workaround for the user is to either redefine the full definition for ”circle” or the modeller to change the initial definition of ”circle” to something like below.

```
circleX is mouseX;
circleY is mouseY;
circle is Circle(circleX, circleY, 100);
```

This now allows the user or modeller to define the coordinates of the circle independently of its definition but has removed all possible advantages of using objects which shall be discussed later such as encapsulation of variables.

Use of these objects can be considered an advanced feature of JS-EDEN as it requires knowledge of JS and combining it with EDEN code to allow the users to define their own objects they can use. This limits the accessibility of objects to beginner users which is something EM strives against.

2.2 Artificial Intelligence

Applying artificial intelligence to models is one of the bigger challenges that is currently faced by EDEN . There are a number of algorithms that can be applied to games such as OXO and pjawns, including minimax and it's extension Alpha-Beta pruning. These require being able to examine up to all possible states that the game could be in the future. Without objects, existing models rely solely on procedural code to represent the state in the array and pass it to different functions.

Objects in EDEN would allow to represent the board in terms of an array as currently but also add a number of dependencies that can be automatically updated as the game state changes. Examples include who's turn is next based on number of pieces placed, is there a winner and if any more moves are possible.

Conceptually when playing a game, the player is building up a tree of possible states that could happen depending on their next move. This is the basis for the minimax algorithm and objects would allow a simpler implementation. Having built up a tree of future moves, it is possible that they could be stored and used for visualisation of what would happen if the user were to place a piece where they currently are pointing. This type of visualisation would be able to model the link between what a human is thinking as they are playing and how a computer models the game.

2.3 Dynamic Objects

Current EDEN models rely on a fairly static structure for models such as games. The modeller must create a definition for every single possible move that can be made on the board. While this removes the need for procedural code to recalculate piece positions for a board, it leads to long list of similar definitions. This can be seen on a number of the existing models in the projects archive.

It makes it difficult for the user to be able to be able to change for example all pieces from a circle to a square. They would need to go through each definition in turn and update as required. The addition of objects would allow to create one structure that represents a piece and use this to place on a board. The user can then go into the object definition and make the change once in the appropriate places and the change would take affect over all instances of that object. This would allow for a much more interactive experience.

2.4 Learning About Models

Models in EDEN have a shared global list allowing defintions to depend on each other and procedures to alter them. These definitions are allowed to be added in any order and there is no notion of grouping definitions together. Naming conventions determined by the author of the model usually determine how well the user is able to understand what affect should happen when they change its value or definition. There are accompanying tools for EDEN that allows the user to be able to see how dependencies connect on what so the user is able to see patterns.

EM's focus is on experimentation and changing definitions to be able to understand how a model works. Without a clear understanding of what should happen, one can be relucant to make a change. It is currently difficult to undo any changes made without either going into the model code and finding the original definition or reloading the model completely, thus restarting the learning process.

3 Object Oriented Programming (OOP)

Object oriented programming uses the idea of objects to represent everything within a program. This section shall discuss the key parts of object oriented programming and how they could be potentially a useful addition to empirical modelling.

3.1 Objects

The use of objects allows to programmer to express their program in terms of a collection of variables and methods. An example would be a car, which has a number of variables that describe it such as number of seats, current speed / gear and it's colour. It also has a number of actions (or methods) that anyone can perform through procedural code, for example: accelerate, break and change gear.

Applying this same principle to EDEN would allow the modeller to think of their ideas in terms of smaller models which combine and interact together to create larger visual models. The variables would become a list of definitions that describe the object, for example the current speed could be described in terms of the current gear, pressure on accelerator and incline of road. The use of objects would allow the modeller to more easily model multiple cars in an environment by creating new instances of the car model, rather than having to repeat the definitions required for each car object. It also would allow the modeller to adapt the number of cars available more easily.

3.2 Typing

As discussed in Beynon (2012), it can be difficult to understand the concept of symbols representation and how they will be used by an operator. Languages such as JS and EDEN do not use explicit type which can lead to unexpected results. For example, a model may have a definition of x + x. The type of x could potentially be unknown to the user if they are experimenting with an existing model, as such x may either be a number or a string. This could lead to an unexpected result of 22 if the user though x was a number when in fact it is a string.

This lack of communication of typing in assignment definitions can potentially reduce the ability to be able to successfully experiment with the model. Explicit typing in EDEN could help users to understand the mind of the modeller and what he expects values to be able to be assigned to in terms of their type.

The main disadvantage of typing in these languages is that it is very restrictive to users who have no prior experience of programming or typing in general. EM is about being able to express ideas without needing to worry about creating a specification. It could dramatically reduce the freedoms that currently exist with the lack of explicit typing.

3.3 Inheritance

Inheritance is another important part of OOP, allowing to define multiple objects that have a shared parent. For example, a person will have a number of attributes such as age, height and actions they can perform which can be put into an object. A student and a staff member can inherit this person object as they both will share these same attributes but also define their own such as list of assignment for the student and classes taught for the staff.

This notion of inheritance would allow to build up relationships between groups of definitions. Using the example above, the modeller could define a person and then a student / staff group which would both share the definitions of the person. This would reduce the need for repetition in definitions and could allow users to be able to better understand what each part of the model represents and the conceptual links that can be made between them.

3.4 Encapsulation

Having examined a number of the models in the project archive, there is a lot of repetition of definitions, for example the games that need to have a definition for each possible piece. Objects allow to group up definitions that model the same object providing a simpler way for users to be able to adapt a model. For example, if the user wishes to change to shape of all the pieces from a circle to a square, the user would currently have to modify every single definition for each piece on the board.

Variables in OOP can have their visibility set to either public or private. Applying this same principle to EDEN would allow to have private definitions that can be called only by actions within that group and can not be modified by external means. Public definitions would then be exposed that can be used in other definitions to connect groups together.

4 Proposal

Having discussed some of the challenges that faces EDEN and how objects could provide an alternative to solving some of these, this section shall present an implementation that attempts to combine definitive programming within an OOP language.

Dupont (2004) discusses how current approaches to bring empirical modelling to object orientation rely on a separation machine to model dependencies. One approach to consider in the future that it mentioned is to change the Java virtual machine to accept is dependencies. This paper discusses the use of a translation tool instead that accepts is dependencies and translates them to standard code.

4.1 JaDEN

The JaDEN syntax explores the ability to adding is dependencies to Java. It uses a translator to convert the additional syntax that can be compiled by a standard compiler. Currently, definitions can be added to class variables within a Java program. Below is an example dependency that can be added to a model.

```
public Text hello is new Text(mouseX,
                   mouseY, "Hello World!");
```

It allows to define objects that combine both the computation and UI within a single group that can be used multiple times within a larger model. The OXO model presented later discusses the use of the translator and what advantages it provides.

4.2 LiveEdit

Adding dependencies alone to Java does not incorporate the true philosophy of empirical modelling which is to allow users to be able to take a model and interact with its definition and see effects (if any) live. The second part of this project is the development of LiveEdit which is a modelling environment which shares similarities to current tools such as JS-EDEN . LiveEdit currently supports some basic interaction using the Java syntax, enabling the user to be able to take a model translated from JaDEN and edit / add definitions of objects.

It's design is aimed at allowing users to create definitions using the Java syntax without needing to worry about advanced features such as ensuring correct typing. It reduces the entry barrier that Java can provide while allowing users to interact with complex models that use a number of objects to model.

The current environment proposed combines visualisation and interaction with the current model on the left portion of the screen. It lists all the variables that the model contains, showing the name, type, current value and the definition if one is defined. This allows the users to easily see the current state of the model and they can change either the current value of a variable or the definition to examine the affect it has. This will update the value of all variables that depend on the updated variable and show on the model on the left.

The next set of information provided is the methods that the model contains, currently these are uneditable, but those that require no arguments can be called at the click of a button. Finally the list of classes that are used by the model are presented and can be explored.

There are two ways that model dependencies can be altered using the environment. The first is to edit a specific instance of a class, for example one agent. The user could change how the position of the agent is calculated while leaving the other instances the same. The other way is to edit all instances by editing the class definition instead. This would allow for example the user to be able to change the size of all agents where other variables have dependencies on size.

4.2.1 OXO Model

Games like OXO provide for interesting discussion to compare it's possible implementations in EDEN and in the newly proposed JaDEN language. Algorithms such as minimax examine the whole state space of possible moves that could be taken by each player in the future. The next move is then chosen by maximising the utility for the current player.

The advantage of JaDEN is the ability to create a Board class which has dependencies for UI, the current / next players and if the current state of the board represents a win. By combining this with the procedural aspect of Java, the model can search through all the different possibilities to find the next best move.

The state of the board is still represented as a 2D array, but additional definitions that depend on this are added to reduce the requirement for procedural code. Whenever the state is change, i.e. a piece is added or removed, other information such as if the board represents a win or draw can be automatically updated via dependencies. In standard Java programs, users would need to explicitly update these values when they change the state. Below demonstrates how these definitions can be implemented:

```
private boolean lineWin1 is
    otherPlayer.equals(state[0][0]) &&
    otherPlayer.equals(state[1][0]) &&
    otherPlayer.equals(state[2][0]);

public boolean playerWin is
    lineWin1 || ... || lineWin8;
```

As well as being able to encapsulate these dependencies together, visibility can be used to only expose certain definitions. The line win definitions are internal to the object and are used to determine if there is at least one win which is exposed and can be used in either procedural code or in a definition.

5 Review

The idea behind both the translator and the environment is to allow a simpler transition both from a Java world to Empirical Modelling and vice versa. The tools presented help to demonstrate possible solutions to the current EDEN challenges discussed. Objects allow to more easily include AI algorithms within models that combine both procedural and definitive scripts. It enables modellers to group related definitions and build up relationships between them to allow shared dependencies. This can make it much easier to begin to understand how each part of the model interacts before experimentation takes place meaning a better learning experience. Finally it enables definitions to be defined dynamically allowing to build more complex models.

One of the disadvantages however with the translation tool specifically is it requires some knowledge of how to program traditionally with Java to be able to fully understand how to model a construal. LiveEdit attempts to resolve this issue but is currently limited in it's functionality of not being able to fully build a model that can be designed using JaDEN .

The paper has presented the incorporation empirical modelling within an object oriented language. Dupont (2004) presents their own implementation of objects within EDEN itself, however it is currently difficult to utilise and misses out on other principles such as inheritance and visibility of definitions. Future projects could look at both extending the tools provides here and simplifying the method described by Dupont (2004).

6 Conclusion

The use of Empirical Modelling, its use of dependencies and relationships provides a unique way of developing models such as the OXO game build for this paper that would traditionally be build using procedural code. As this paper has discussed, no one tool is going to provide a one size fits all but it is hoped that the proposals presented provide a good basis for the discussion of incorporating objects in the future of Empirical Modelling.

References

Meurig Beynon. Realising software development as a lived experience. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 229–244. ACM, 2012.

Jean-Pierre Dupont. *Script partitioning in the comprehension and development of Empirical Modelling artefacts*. PhD thesis, 2004.