Uses of Empirical Modelling in a frisbee simulation

Abstract

This paper shall describe how Empirical Modelling was used in constructing a small simulation of a frisbee game. The purpose of the paper is to publish why Empirical Modelling is distinct from other forms of software development. What makes it so different? For what reasons should Empirical Modelling be a widely accepted form of creating software artifacts? The document will state what aspects of Empirical Modelling were particularly useful in the construction of the frisbee model. How does Empirical Modelling help in constructing models in general? Does it help at all? If Empirical Modelling was not suitable for building the model, why not? Would these failings be important in the development of other models? The paper shall highlight how the development of the EM model would differ from the development of an object-oriented equivalent. Is there a noticable improvement over the current accepted way by which software is engineered?

1 Introduction to Empirical Modelling

Empirical Modelling (EM) is an approach to computer-based modelling, oriented around observation and experiment. There are three main concepts: observable, agent, and dependency(**?**).

Observable This is any feature of the model that can be perceived and identified, and are subject to change.

Agent Anything that can change the state of the system or model.

Dependency A dependency is some relationship between observables, such that changing one observable has a determined effect on observables dependent on it.

Empirical Modelling attempts to establish a correspondence between two sets of observables:

- Those that are observed in the real world, and are the subject of the model, and
- The reference set of observables that are defined by the features of the computer model.

Establishing the correspondence is an iterative process, where the model is experimented with, compared with the subject, and thereby refined. The idea is to make the model in such a way that if observables are changed in the subject and there are effects as a result of those changes, then the computer model exhibits similar behaviour if the corresponding set of observables are changed.

The modeller uses techniques of observation and agent oriented analysis. This involves determining the observables, agents and dependencies in the subject situation. To achieve this, the modeller would construct an LSD specification(**?**).

The modeller identifies the entities in the subject that he perceives as agents in the system. He also discerns any observables that may be important in the model. These are recorded within the specification. There are five parts to an agent's definition in LSD.

Oracles These are the observables that act as stimuli for the agent.

Handles These observables can be redefined by the agent (handles). Usually some of them are included in the list of oracles.

States Observables that are closely associated with the agent.

Derivates The relationship between observables that are characteristic of the agent's interaction with the environment.

Protocol The list of possible state-changing actions the agent can undertake.

The agent definitions record the observations and relationships between them that have been identified by the modeller. The definitions are subjective, in that modellers may have different opinions as to how observables are related to each other, and hence may produce different LSD specifications.

The next stage is the construction of the computer model. This is done by using definitive notations and scripts. A definitive notation is a language that allows an observable to be expressed as a variable, and expresses relations between observables as definitions, hence the term *definitive notation*. A definitive script is a set of definitions such that, when passed to an interpreter, the model runs.

2 Software development with object orientation

Different primitives exist in the object oriented paradigm. As the name suggests, the main primitive notion is that of the object. Software constructed using object oriented analysis consists of many objects communicating with one another, exhibiting predetermined patterns of behaviour.

In mainstream software development, there are five principal stages undertaken by the programmer.

Analysis Examination of a real world system for the purpose of creating a design for the software.

Design According to what was found in Analysis, a design of the software. In the context of object-oriented programming, a specification is built, showing relationships between objects and their lifetimes.

Implementation The actual coding stage, where the program is built.

Testing Written code is compiled and tested to check for errors, and any conditions where the software fails.

Maintenance The software has been released for widespread use. Minor changes may be introduced at this stage.

In the classical model of software development, the progression from one stage to the next was very strict. For example, once the design stage is complete and coding is underway, developers are not supposed to backtrack to the design stage. It was found however, that this did not really occur in practice, and with good reason. During one of the later stages it is possible for an unforeseen problem to be encountered, that forces the development to return to a previous stage.

In this document, the stages of interest are that of analysis, design and implementation.

When constructing an object-oriented program, the developer follows an approach not too dissimilar from that of the empirical modeller. During analysis, the requirements of the software are inspected, and certain entities are identified, as possible objects to be recreated in the program's design. The software is then designed, using a language to specify objects' relationships with each other. The specification language of choice at the time of writing is the Unified Modelling Language (UML). This specification language, together with accompanying documentation is then interpreted by programmers and transformed into the software required.

The subsequent sections shall outline what the frisbee model is, and how empirical modelling was used to construct it. Comparisons with the object-oriented mode of development will be drawn as the discussion progresses.

3 Analysing and specifying the frisbee model

This model attempts to simulate the scenario when two members of an Ultimate Frisbee team need to pass the disc to each other. In the game, the person holding the disc is not allowed to run with it. They have to throw the disc to a teammate within eight seconds of receiving the disc. A player preparing to receive the disc runs toward some space on the pitch, away from a nearby defender. The defenders have the goal of intercepting the disc while it is in flight, by either knocking it to the ground or by catching it. In this model there are only two participants, the thrower and the catcher. The model takes place on a two-dimensional playing field, as if an observer were watching a real game from high above the pitch.

The main agents in this model are judged to be the disc, catcher, thrower, and the target. The target is the most obscure agent out of the four. The target represents some space that the thrower and the catcher focus upon. For the catcher, it is a region in which he believes he may receive the disc, even though a defender may be nearby. For the thrower, it is a region to throw the disc towards. Although in a real game, the target may be a large space, in the model, for the purposes of reducing complexity, the target is merely a point on the field.

In a real game, it is usually the catcher who determines where the target is. He may either signal the target's location to the thrower, or merely dash towards it, leaving it up to the thrower to deduce what needs to be done. On the other hand, the thrower can indicate a target to potential catchers. Either way, some agreement is struck between the two agents on

the position of the target. In the model, the target is determined at random, or by the modeller himself by clicking on the playing field. The reason the target is seen to be an agent, not merely an observable, is that the target influences the velocity of the catcher and of the disc.

Most of the crucial observables in the model are the respective agents' velocities and their positions. Also *clock*, which represents the current time, and *trigger_time*, which is when the disc is thrown. Figure **??** shows the LSD specification for one of the agents, the disc.

The other agents are similarly defined. The catcher is the most interesting of all the agents, and it was not obvious how to model it correctly. At an early stage, its specification was left vague, to be determined via experimentation.

In the subject situation, once a target has been agreed on by the thrower and the catcher, the thrower moves toward the target, usually in an effort to outmaneouvre the marking defender. Once the thrower judges that it is safe to pass to the catcher, he throws the disc (towards the target). When this occurs, the catcher judges the disc's flight and attempts to intercept it. Once the disc is thrown, he forgets entirely of the target and focusses on moving towards the disc.

At early stages, when first writing the specification, there may be indecision about how an issue can be resolved. Using the existing specification (in parts written in English rather than in terms of logic) was quite beneficial, since the modeller could quickly translate most of the specification into a definitive script to allow him to investigate the issue of concern. Once he had determined whether a course was suitable by interacting with the model and experimenting with it, the specification could be revised so that a better definitive script could be derived from it.

Writing the LSD specification was very useful, as it served as a means to clarify what was involved in the model. It helped to determine what information was needed in order to perform a calculation, and crucially, it allowed the modeller to identify what dependencies existed. This is very similar to how UML serves the program designer in OOP. UML clarifies what object oriented entities there are in the problem domain. There are no clear advantages of using LSD in an EM context, over using UML in an OOP context.

4 Definitive scripts and tkeden

A definitive notation is the empirical modelling equivalent of a programming language. Definitive scripts are the equivalent of a program's code. The definitive script is fed into an interpreter tkeden, which starts the model.

The notations used were a combination of Eden, SCOUT, and DoNaLD. Eden is a general notation, as opposed to the rather specialised notations of SCOUT or DoNaLD. SCOUT is designed to define and deal with screen layouts, whereas DoNaLD defines 2D geometry(**?**). DoNaLD was the main notation used, since most operations were to do with manipulating points on a 2D plane, and displaying those results on a 2D drawing of a field. Tkeden interprets all of these notations, indeed several notations can be used in the same definitive script.

There is a glaring disadvantage with using a mix of notations for a single model. It is uncomfortable for a developer using a conventional programming to use another language within the same project. Often different languages require different mindsets, they require a developer to think in different ways. It would be confusing to have to declare one part of an object class in C++ and another part in Java for example. Even though the two languages are very similar, they require subtle shifts in the programmer's thinking.

Different notations can confuse the modeller, particularly since some of the notations have a specialised scope, and for some definitions, an alternative notation has to be used. This is not helpful if a modeller knows how something could be defined in one notation yet that notation does not support what the modeller requires. No more than one notation should be required in a definitive script.

Figure **??** shows an example where the modeller wished to perform a simple calculation. The line beginning with `##` denotes how the calculation could be expressed in DoNaLD. However, the procedure `max(x, y)`, is not defined within that notation. Therefore, the general notation, Eden, had to be used to define it as shown on the line below. This imposes an unnecessary burden on the modeller to discover appropriate Eden functions to write the definition.

A solution to this would be to allow the modeller to define datatypes, and operations over those types in his notation of choice. For instance, it would be helpful to be able to define a `matrix` type with accompanying multiply, and add operations. It would have been more convenient in the previous situation to define the `max(x,y)` operator for two numbers `x` and `y`. Such features would allow modellers themselves to extend the potential uses of a definitive notation. It would not be necessary to switch from DoNaLD to Eden, or from Eden to SCOUT, because each notation could express what another can express.

```
 agent disc() {

oracle

(time)     clock
(time)     trigger_time
(real)     thrown
(vector2d) target_pos      // the position of the target
(vector2d) thrower_pos     // the thrower's position

state

(vector2d) disc_pos        // position of the disc as time goes by
(vector2d) init_disc_pos  // initial position
(vector2d) disc_velocity   // the disc's velocity
(real)     thrown           // whether the disc has been thrown

handle

(vector2d) disc_pos

derivate

disc_pos = disc_velocity * max(clock - trigger_time, 0)
+ init_disc_pos

disc_velocity =
(target_pos - thrower_pos) * ((throw_speed)*(thrown)
div dist(target_pos - thrower_pos))

protocol

(mode >= CATCHING) -> thrown = 1.0
(mode >= CATCHING) -> thrown = 0.0
}
```

Figure 1: LSD describing the disc agent

```
## disc_pos = disc_velocity * max(clock! - trigger_time!, 0) +
               init_disc_pos

_disc_pos is vector_add(scalar_mult(_disc_velocity,
    max(clock - trigger_time, 0)), _init_disc_pos);
```

Figure 2: Two equivalent definitions in DoNaLD and Eden.

The idea of dependency lends a large favour to empirical modelling. With dependency, it is only necessary to describe the formula required to calculate a variable.

$$s = ut + \frac{1}{2}at^2 \quad (1)$$

For example, assume that the formula in equation **??**, is written in definitive notation. If any one of the variables u, t or a change, s will be updated according to the formula. If t changes, not only does s have to be updated, but also every other variable which depends on t. In object-oriented languages, a bulky method or procedure would have to be defined to change all variables that depend on t. Additionally, if at any stage the program's code is revised, and more variables are added and are dependent on t, it is easy to forget to update the method to change those new dependants. With dependancy, updates are handled by the interpreter, therefore the modeller does not have to consider if the variables are up to date or not. Dependancy guarantees that they will be. Since bulky methods do not have to be written, dependency provides the added benefit that definitive scripts are kept concise, hence a modeller can rapidly construct models.

It was stated earlier that constructing the model is an iterative process. One aspect of this is that a model's variables can be queried as it is being run. This is a very useful feature that tkeden provides. Using this, one can quickly grasp how a model works. When querying a variable, various pieces of information are printed; the value of the variable at the present time, the variable's definition, and which other variables are dependent on it.

This information can used during the iterative process of constructing the model, to ensure that it is working as expected, or to determine how particular problems arise. Also in some cases variables can be redefined without adversely affecting the model, since, due to dependency, all relevant values will be updated after the change.

In the context of modern software development, variable querying is only usually provided in program debuggers. Even then it is generally not encouraged to change a variable's value as the program runs. In EM, it is an important part of the interpreter. The main reason for this is it allows someone who has never used the model to learn how it works, and investigate how changes can affect other observables.

It was noticeable that in definitive notations, there are no constructs to limit the visibility of variables. This is counter to one of the major principles of object-oriented programming, namely that an attribute of an object should not, in general, be visible to other objects. This is encouraged in OOP languages to prevent data being changed unintentionally, or force data to be updated correctly by accessing appropriate methods. Due to the concept of dependency, this protective measure is unnecessary. Provided that a variable is defined correctly, and it is not redefined as the model runs, that variable will always store correct, relevant data. In OOP programs, omitting the change of a variable, or mistakenly changing a public variable is a common cause of obscure bugs, especially in complex software.

5 Conclusions

In conclusion, empirical modelling is promising as an alternative to mainstream software engineering as we know it today. There are many similarities between the two paradigms; analysis of the problem space, specification of the solution (the model or the program), and the implementation of the solution.

Major benefits of empirical modelling over object oriented programming in particular are:

1. Dependencies. The notion that a variable updates automatically when the variables it is dependent on change, is an excellent tool. Instead of having to declare procedures to update variables, the environment handles the problem for the modeller. This allows scripts to be shorter and clearer. Dependencies also allow a modeller to experiment with his model by changing values or definitions as it is running.

2. Definitive notations. The fact that variables are defined once as formulae serves to make a model easier to understand, since it is defined in one place. The simplicity of the notations also allows models to be developed quite rapidly.

3. Iterative improvement of the model by experimentation. This is a key feature of EM, allowing a modeller to analyse his model by observing how variables change, and proves to be extremely useful.

However there are some issues that would need to be addressed before there would be widespread adoption of EM.

A significant milestone would be to design every definitive notation to encompass any foreseeable use for it. The notation should not merely be designed for a specific problem or scenario. As the notations are at present, one would not believe that they are designed to cater for many common uses of computing, and

this limits their appeal. In general, software developers of any kind would rather work with one language to construct a piece of software. They would usually prefer not have to make a (possibly unsettling) paradigm shift to work with the alternative notation.

On the whole empirical modelling makes an encouraging change to working with computers. If developed further, it could be more appealing to people in general, not just to scientists. It would be a delight to see that day come.