Supporting the Project: A Study into the Modelling of a Project Lifecycle

---

Lee Lewis Herrington (0409657)

This paper invites a 50% weighting of the Total Mark

Submitted to the Department of Computer Science at the University of Warwick

Abstract

The hierarchical structure of a project and the data dependencies between artefacts lends itself very strongly to the ambitions of a new computing paradigm known as Empirical Modelling. In EM we use can use a model to define a project lifecycle, we can say that a project has tasks and that each task has a duration and an assigned resource working to a given calendar. The time it takes a project to complete depends on the duration of the tasks which in turn depend on the number of hours a resource works per week. More formally with respect to EM terminology, the tasks, resources and calendars can be considered to be the observables; the relationship between these observables are dependencies while agents can be likened to the changes in project state. This paper concerns my technical study into the use of Empirical Modeling as a means to manage a simple project lifecycle

Introduction

Empirical modeling is about making artifacts to support human thinking and is heralded by its proponents as a radically different way in which a person interacts with a computer. Conventional interaction is considered impersonal and circumscribed while EM is thought of as personal and cognitive, concerned more with exploring experience and meaning. A number of special-purpose tools, by way of interpreters and APIs, have been developed to support EM. The most extensively used EM tool developed so far is the EDEN interpreter. This tool has been the principal basis for most of the practical work on EM to date. EDEN exists in three principal variants, intended for text-based interaction, line-drawing and window management, and distributed use (EM Principles).

In response to the question "What is Empirical Modelling?" the architects state that "EM is centrally concerned with the way in which understanding emerges through life lived forwards, such understanding enables experience to mould our interaction" (What is EM). This is related to project management in that decisions made at any one time are often based on emerging experience throughout the project lifecycle. Specifically, if a project resource takes x man hours to complete task a, and task b is similar to task a, then task b should roughly take the same length of time. Moreover, with respect to iteration based software development methodologies such as extreme programming (XP), what happens in a single iteration is very much based on the understanding derived from the previous iteration. For this reason a single iteration is effectively an experiment. I start by outlining my objectives, then I attempt to model the objectives noting any problems encountered before finally presenting either a working model or reasons why the objectives could not be achieved. Readers that are new to EM can follow each experiment and attempt to build up the model themselves before comparing their results to iteration(n).e where n is the iteration number. At the end of an experiment the reader will either learn a new EM technique or an understanding of a particular limitation in the technology. Accompanying this paper is a set of Eden files that constitute the iterative development of the model. These files are provided for future readers of this paper to observe how the model was built up and should therefore not be considered part of the submission or used in the marking process. **Run.e** is the final release of the model and a simple user guide (**quickguide.pdf**) is provided to help the reader execute and interact with the model.

Key EM Concepts relating to Project Management

We now discuss some key EM concepts trying to relate them to terms or experiences within the project management domain. By aligning the two concepts it should make it much easier to then develop a working model. Prior to beginning this study I undertook a significant amount of research using papers available on the Empirical Modelling website. The paper that influenced

me the most was ‘Human Computing – Modelling with Meaning’ (Beynon et al 2006). Managing a project has often been considered an art form and so appreciating the partnership between humanities and computing was central to developing ideas for this paper. A second paper that guided my thinking in the later stages of the study was ‘Liberating the Computer Arts’ (Beynon 2001).

2.1 Construal

Central to the theories underlying EM are the published works of various philosophers such as David Gooding and Brian Cantwell Smith. One particular book by Gooding, namely “Experiment and the Making of Meaning” (Gooding 1990), introduced the term *construal* to characterize the artifacts developed by experimental scientists in their preliminary studies of phenomena. The word has been employed in EM to describe artifacts that embody the modeler’s understanding of a situation (EM Key Concepts). Since no two projects are ever the same, can it be said that a project is to a degree a phenomenon being investigated by a project manager? Like a scientist that has conducted prior research, the project manager has explicit knowledge of the world as well as tacit knowledge from previous projects. Regardless of this knowledge he cannot say without experimentation what the effect of changing certain project properties might be. What he essentially construes from such experiments can be used to redefine the project model, thereby increasing the managers understanding of his or her current situation. A construal with respect to project management is therefore an artifact derived from experimentation with project properties. For example, the number of days the project will take to complete can be experimented with by changing the resource assigned to a task or by increasing the length of the working day.

2.2 Dependencies, Observables and Agents

Within the context of EM, an *observable* relates to an item within a model to which we can associate either a discrete value or a status. For example, in the model of a room, we might consider the width of the door to be a discrete observable, while the state of the door in terms of whether is open or closed would be a status. The position of a desk within the room might (in the case of a small room) depend on how wide the door opens. This in EM terms would be an example of a dependency. The position of the desk depends on the width of the door. A reverse dependency might exist such that the desk is so wide; it places a restriction on the width of the door. If the homeowner decided to widen the door frame to cater for wheelchair guests, then without consideration for the desk, the new door might not open. In the model of the room we might have an observable named ‘HitsTable’. This observable is true whenever the area drawn out by opening the door conflicts with the position of the desk.

An *agent* is an entity in the domain being modeled that is perceived as capable of initiating state-change. In developing an EM model, our perspective on agency within the domain evolves with our construal (EM Key Concepts). In the room model we could say that the home owner is an agent. When the homeowner enters the room he changes the opened state of the door. In the case whereby the door has been widened, the observable 'Hits Table' may then be set to true thereby alerting the modeler. Of course this is a simple example and could be satisfied without the need for a model. However, consider the situation when modeling the building on an entire house where there are hundreds of dependencies.

Given a general understanding on these three major EM concepts, let us now briefly try and relate them more specifically to project management. This is really the first step in determining whether or not a project lifecycle can be modeled. Rather than define exhaustively at this stage all of the project properties, let us summarize that properties such as the duration of a task, the value of a cost and the score of a risk can be considered to be the observables. The roll up of these observables to high level observables such as the total project cost or the total project duration would then be dependencies. Agents can be likened to the changes in project state either by the project manager externally or by some pre modelled action triggered by the change in value of an observable.

2.3 Spreadsheets

With respect to project management, spreadsheets are really an entry level tool especially for managing task dependencies. An industry example of this is the Microsoft project management application which builds upon a simple spreadsheet. The start date of one task as defined in a cell can be dependant on the end date specified in another cell. To a certain degree, an EM model can be likened to a spreadsheet in so much as a cell generalises to an observable and the formulate bound to a given cell captures dependencies amongst observables. However though EM subsumes activities associated with a spreadsheet's creation and use, spreadsheets have sophisticated forms of dependency based on geometric organisation of cells that are not easy to express in definitive scripts (EM Glossary). It is not envisaged that when I attempt to model the lifecycle of a project I will be trying to develop an abstract spreadsheet for creating dependencies between tasks. This is clearly beyond the scope of this study but could form further work and will therefore be discussed again the relevant section.

2.4 Definitive Scripts & Notations

We have talked about modelling a project in EM but before we can begin to do that we need to understand the available tools and languages. A definitive script is essentially a collection of definitions stored in a text file. A single definition refers to an observable which can be a

constant value, a dependency or a function. For example we could say A = 2 and B = 3 and C = Maximum (A+B). Here A and B are defined explicitly while the value of C comes from an evaluation of the maximum function. Modeling with definitive scripts is first and foremost concerned with the development of private artifacts to aid the modeler's understanding, but it can also serve to express the viewpoint of other agents on interaction within a situation, subject to a process of projection on the part of the modeler (EM Glossary). When a model has been defined as one or more definitive scripts it is then interpreted by a tool known as tkeden. The same tool can then be used to perform redefinitions. It is these redefinitions that are in essence where the experimentation comes in. A model that defines a project lifecycle might have a definition concerning the number of available resources. As the project gets closer to the deadline, the manager would want to redefine the number of available resources to see what effect it might have on the schedule.

A number of special-purpose tools, by way of interpreters and APIs, have been developed to support EM. The most extensively used EM tool so far developed is the EDEN interpreter which has been the principal basis for most of the practical work on EM to date. EDEN exists in three principal variants intended for text-based interaction, line-drawing and window management as well as distributed use (EM Principles). EDEN is essentially the Engine for interpreting Definitive Notations and is the primary software tool of the Empirical Modeling research group. We build models with it, using a variety of definitive notations that it implements (Eden notation). As we build up our model of a project we will use the EDEN engine to evaluate the model and make changes based on what we observe. Other notations have been introduced to provide line drawing and windowing capabilities notably Donald and Scout.

2.5 Data Structures in Eden

The first step in representing a project lifecycle as an empirical model is to determine the data structures that will be used to represent the project artefacts. This is quite difficult because a *list* is the only structured data type available and there are no abstract data types. One possible way in which we can model a project is to engineer the data as a set of lists whereby each element of the list contains a list of its own. For example we might structure a project as a list of tasks and resources. Each task would consist of a title, the number of man hours, the resource and a projected start date. From this simple set of data structures we could then add more interesting observables whose values would depend on functions operating over the set of data. For example, we might define: ProjectDueDate is findLatestTask().

Developing the Model – An Iterative Approach

As a relatively young language, Eden naturally has limitations and quirks which at times can make it difficult to achieve the desired objectives. What might first appear to be a straight forward idea can present extensive frustration especially when interoperability is required between definitions made in Eden and other notations such as Scout or Donald. As a result, what I propose to do is to work iteratively in the development of the project model. As the iterations progress I will look to introduce greater complexity and more interesting functionality. Ultimately what I am trying to develop is dashboard interface that presents a summary of the project lifecycle. In addition I would also like to present to the user a series of operations which they can perform to vary the contents of the dashboard. For example, let us assume that a resource has been assigned to a task who only works three days a week but costs £3 per hour. By changing the resource to someone who works four days a week but costs £5 per hour, what effect might that have on the project deadline and the total costs? Taking into the possibility of say a late fee? It might be that the cost saving of using a different resource saves more money overall, even when the late fee has been factored in.

One of the benefits of this paper is that in addition to trying to model a project lifecycle it is also a technical study in the use of EDEN. The paper clearly documents where the use of definitions has proved useful and also what technical problems have had to have been overcome. Essentially the paper is a journey into the unknown, as we progress through iterations we should hopefully learn more about EM and our model will improve as that knowledge increases.

3.1 Iteration One

The objectives of this iteration are to define some simple definitions based on project properties as well as some simple functions for calculating the amount of days and weeks that a project is going to take to complete. In addition I would like to develop a very basic dashboard interface using the scout notation that presents the values derived by the summation functions.

Findings & Results

In this first iteration I have been able to define some simple project properties and introduce two basic data structures built using lists. I have also modeled simple calendars to define days and hours with respect to full and part time workers. Resources have been modeled using lists whereby one of the elements of the list is a calendar. This effectively offers the potential later on for the project manager to set every resource to the full time calendar which is something

that might happen if the project was running late. Tasks have been modeled as lists, each with duration in terms of man hours as well as an allocated resource. Since each resource works to a given calendar, the project manager might choose one resource over the other in order to speed up the delivery date. Several functions have been defined that provide the data used on the summary dashboard. For example, the *totalcost* of the project is obtained from the *sumTaskCosts ()* function. This function iterates over all of the tasks and uses the rate per hour of the assigned resource to determine how much the task is going to cost; this value is then accumulated over all of the project tasks to give the total cost.

A very simple scout implementation of the dashboard has been introduced showing a summary of the project properties. What is interesting is that when you use a redefinition to change the resource assigned to a given task, the dashboard immediately updates (see problems encountered below). Since this is a key requirement of the dashboard, it is reassuring to be able to demonstrate it at such an early stage in the modeling. The following annotated screen shot shows the simple project dashboard. Though limited and simple looking, a number of important concepts have been established which I will attempt to build on in iteration 2.

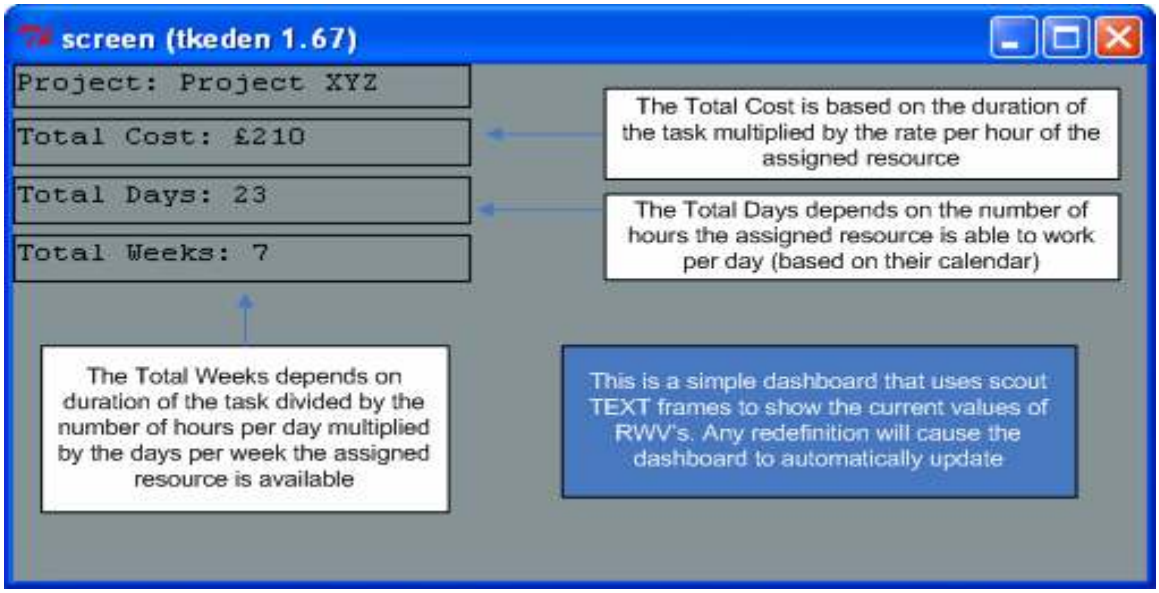

Figure 3.1.1 – Iteration 1 Dashboard Screen Shot

Problems Encountered

During this first iteration I have encountered a number of restrictions that have forced me to rethink what is going to be possible in future iterations. One of the most significant restrictions concerns the use of definitions in lists. This problem has been previously identified in the

frequently asked questions section of the EM website[1]. In my early attempts at the model I defined tasks as follows:

1. task1 = ["Develop a cellular prototype",40,resource2];
2. task2 = ["Research chemotaxis",30,resource2];
3. tasks = [task1,task2]

I then wrote a function bound to *tasks* that would update the total project duration whenever a value in one of the individual tasks changed. In simple terms the problem is that there is no notification on the tasks definition when something changes in one of the child tasks. This is because the contents of the *tasks* read write variable (RWV) becomes fixed when they are assigned. As a result I am required to have a procedure bound to each task that calls a project update function whenever the values in that task change.

*(By Iteration six I discovered the 'Is' assignment eg **Tasks is [Task1,Task2]** which partially solved this problem, see iteration six for reasons why this was only a partial solution)*

3.2 Iteration Two

In this second iteration I would like to generate a table in the scout interface below the dashboard that lists all of the project tasks. The tasks should appear in a kind of spreadsheet and indeed the cells for the task name and duration should be editable. Any changes to these values should be immediately reflected in the project dashboard. If achieved, this simple functionality will prove the concept of low level changes 'rolling up' to the high level project observables.

Findings & Results

As you can see in figure 2.0 the tasks are displayed as a kind of spreadsheet. For each task, I draw three scout textboxes. When scout has finished drawing the table I then use the setText () function in Eden to insert the task values into each of the boxes. Assigned to each scout textbox is an event handler which is executed whenever the contents of the textbox change (achieved by adding SENSITIVE: ON to the scout textbox). This event handler obtains the value from the textbox using the Eden getText () function and updates the task definition accordingly.

[1] http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/eden/faqs/ (Item 15)

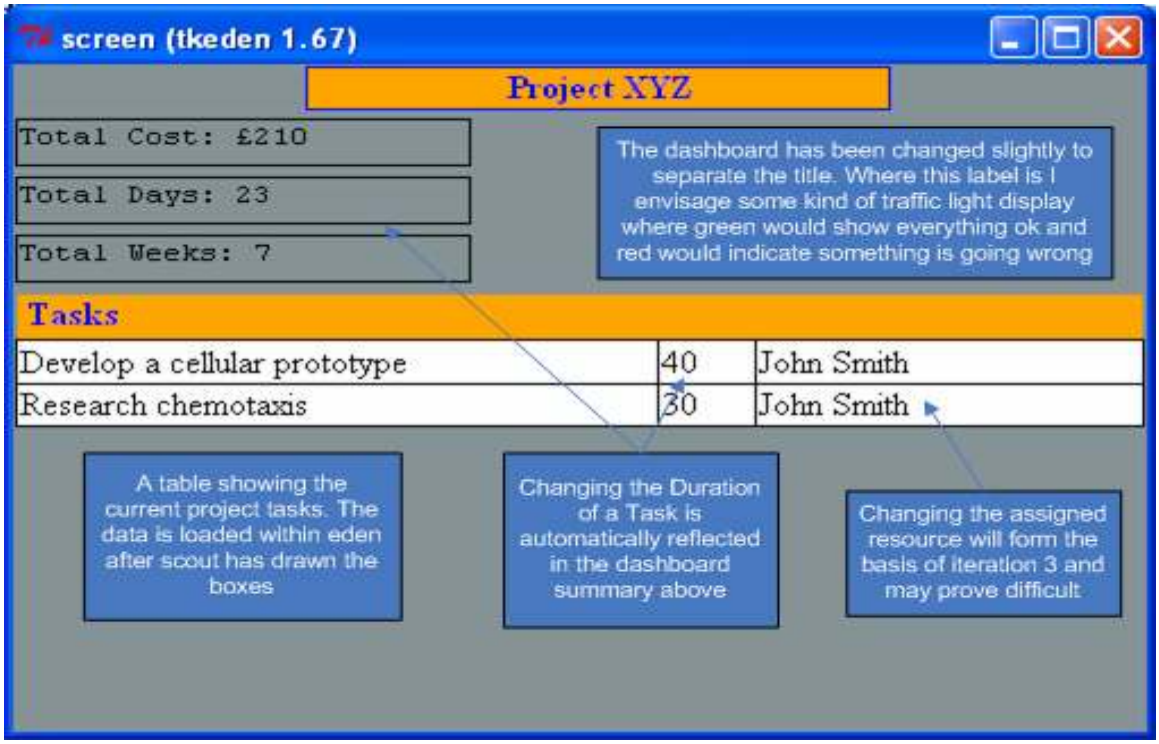

Figure 3.2.1 – Iteration 2 Dashboard Screen Shot

By changing the task definition, a redefinition occurs which automatically updates the dashboard summary above the table. In addition I have also moved the project title into the top centre of the screen to frame the window.

Problems Encountered

Initially I wanted to achieve the objective of presenting a table of tasks by iterating over the list of tasks and generate the scout definitions for drawing the table dynamically. Unfortunately scout window definitions simple cannot be generate dynamically or at least as not as far as my investigations found and therefore I had to find a different way of drawing the table. As outlined in the findings above, I had to explicitly define the required window frames for each task. Clearly this means the solution is not scalable since to add a new task to the project, one must update the model and insert the required scout window frames.

A second problem identified in scout was that when you define a text box you cannot bind its contents to a definition. For example, what I am trying to do is to provide a textbox that allows the user to change the number of man hours for a given task. The current number of man-hours is stored in a list assigned to the task, eg Task1 = ["Develop a cellular prototype", 40, resource2]. I cannot however, define Task1[2] = TextBox1 but must instead add an event handler to TextBox1 such that when its contents change I then manually update the value of

Task1[2]. This effectively means the goal of EM to allow a modeler to use definitions to define relationships rather than hard coding the relationship is not achievable in respect to Text Boxes.

3.3 Iteration Three

What I would like to achieve in this iteration is to provide a simple mechanism for changing the resource currently assigned to a task. Typing the name of the resource into the spreadsheet would be cumbersome and error prone therefore we need some kind of dialog showing the list of available resources from which one can be selected. When the resource changes the dashboard should change because each resource works to a different calendar and also charges a different hourly rate.

Findings & Results

My initial idea for achieving the iteration objective was to develop a drop down box that allowed the user to select a resource from a list of those available. This proved particularly complex due to the limitations of scout and so in the end I decided the easiest metaphor to adopt was one of 'cycling' through the available resources. At the end of each row in the table of tasks is a simple orange button (a scout textbox). A single click on the button executes an event handler which changes the resource allocated to the task to the next one in the available list. In order to support the changing of a resource I have written two ancillary functions. The first function takes the name of the current resource from the scout textbox and returns its index in the list of available resources. The second function takes this index value and returns the next resource from the list (modulo resource list length). The task is updated with the new resource and a redefinition of the project dashboard then occurs.

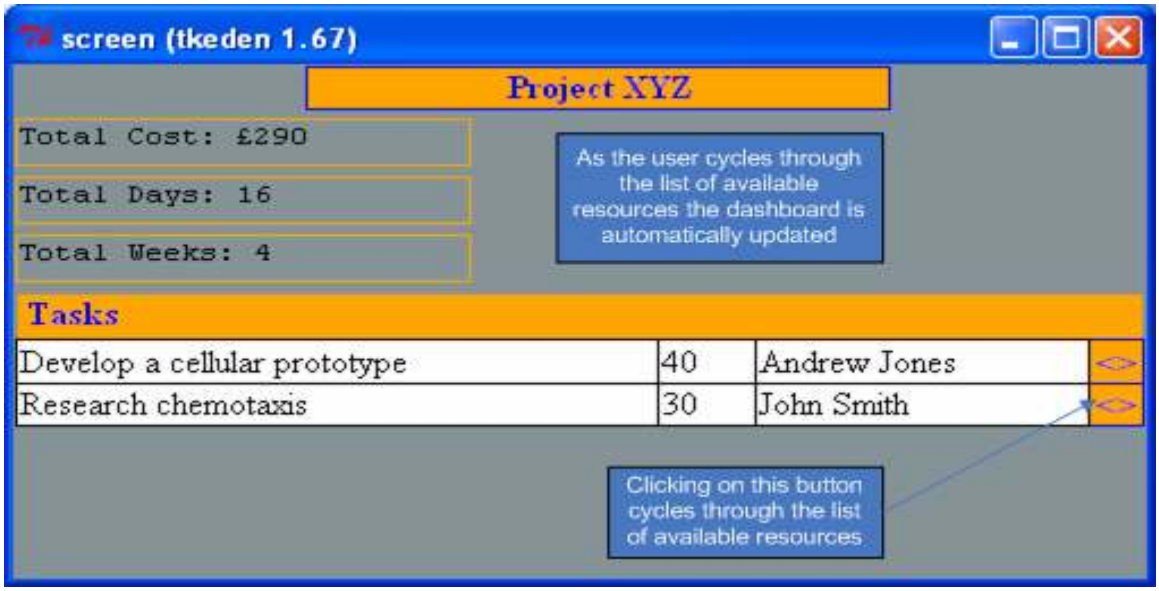

Figure 3.3.1 – Iteration 3 Dashboard Screen Shot

The technique developed to facilitate changing the resource is actually very straight forward and is likely to be used again in the next iteration when we attempt to provide an interface for modifying resource properties.

Problems Encountered

One of the concerns identified in this iteration was the amount of Eden and scout code being duplicated in order to generate the interface and provide the event handlers to update definitions. The proposition of creating a dependency hierarchy such that definitions are updated automatically without the need for ancillary functions is proving difficult to achieve. The root of the problem comes from the need to explicitly refer to read write variables when executing re-definitions. What I will need to do in later iterations is determine whether methods such as *execute()* can be used to reduce some of this duplicate code and re introduce a more structured dependency hierarchy.

3.4 Iteration Four

The purpose of this iteration is to consolidate the model developed in the previous three iterations. I mentioned in the problems encountered section of the last iteration that I was concerned about the duplication of definitions and the inability to employ a strong definition dependency throughout the model. What I am looking to do in this iteration is to streamline the code and introduce more native dependencies between definitions.

Findings & Results

During further investigation into the Eden language I discovered point (16) on the Frequently Asked Question section of the website. My concern about the number of similar definitions has clearly been identified in the past and it would appear that the *execute()* procedure is one way to address this. Not only can it execute basic Eden functions, but it can also execute Donald code as well. The following example demonstrates creating n number of Donald lines 10 spaces apart.

```
proc p {
  para num;
  auto i;
    for (i=1; i<=num; i++) {
      execute("%donald
               point p"//str(i)//"
               p"//str(i)//" = {10+"//str(i*10)//",10}
               line l"//str(i)//"
               l"//str(i)//" = [p"//str(i)//",p"//str(i)//"+{0,500}]
              ");
```

```
        }
}
```

Based on the above example I attempted to replace all of the definitions that construct the table of tasks as shown in figure 2.0 with a single function that accepts a list of tasks and constructs the table dynamically. Despite my best efforts I was unable to write the function such that it would be accepted by the interpreter.

While streamlining the code, a fundamental error in the calculation of the dashboard summary fields was identified. The simplicity used in calculating the number of days and weeks that the project will take assumes a resource can work on two tasks at the same time which is clearly not possible. For example, if two tasks take two hours and three hours respectfully, and the resource works two hours a day, then the project should take three days to complete. In iteration three the dashboard shows two days which is clearly incorrect. The functions effected by this issue were sumTaskDays() and sumTaskWeeks(). In this iteration I changed these functions such that they now track the current workload of the resource using a simple accumulation list.

One of the strongest lessons to come out of this iteration was that it is easy to create spaghetti loops in operations. That is to say you might need to access the value of a definition within a proc but to do so might cause a redefinition event to occur such that immediately after it ends, the interpreter executes the operation again thus causing a loop. The majority of the time trying to solve the calculation problem was really taken up finding a suitable mechanism for accumulating the work load of each resource. Attempts to store the value in the actual resource definition failed miserably because of scope problems.

The first part of this iteration concerned the reduction of definitions in the model by using the *execute()* function. A number of approaches using this function have been adopted that have significantly reduced the amount of code in the current version of the model. Better manipulation of lists has allowed me to introduce a single action to monitor a list of tasks rather than have an action for each task. The layout of elements in scout has also been dramatically overhauled. Fixed points have been replaced by a relative positioning system which has made it much easier to add new tasks to the table.

Problems Encountered

Use of the *execute* function has been used extensively; however, it should be pointed out that the *execute()* function cannot generate event handler procedures on the fly which is a severe limitation. The result of this is that adding a new task to the project means you have to set up the row of scout textboxes in the table and add event handlers explicitly. Furthermore, a

restriction on the *screen* variable in scout means that frames cannot be dynamically added or removed from the window.

3.5 Iteration Five

The next feature that I would like to introduce into the model is a traffic light frame onto the project dashboard. In the model I have defined baselines for the project costs, total days and total weeks. If the current project costs, total days or total weeks exceed the baseline then I would like a traffic light to indicate this. A green light indicates no more than one baseline has been breached, an orange light indicates two baselines have been breached while a red light indicates all three baselines have been breached.

Findings & Results

Originally I tried to use a Donald viewport to draw a traffic light but a bug in the interpreter (see problems encountered below) meant I had to rethink my strategy. In the end I decided to use simple scout textboxes and bound the background colour of the textbox to the current value of the traffic light status. The code to calculate the traffic light status was relatively simple.

```
/* compare to the baseline */
t1 is (totalcost>basecost)? 1:0;
t2 is (totaldays>basedays)? 1:0;
t3 is (totalweeks>baseweeks)? 1:0;

/* Amber if two conditions over the baseline */
t4 is ((t1 && t2) || (t1 && t3) || (t2 && t3)) ? 1 : 0;

/* Red if all three conditions over the baseline */
t5 is (t1 && t2 && t3) ? 1 : 0;

/* Traffic Light status */
tls is (t5) ? "Red" : (t4) ? "Amber" : "Green" ;
```

The bgcolour property of the traffic light box shown on the project dashboard was then bound to the tls definition. Whenever the project properties change, the traffic light status is automatically re-calculated. The following figure is a composition of the three possible traffic light statuses.

Figure 3.5.1 – Showing the different traffic light status on the project dashboard

Problems Encountered

The use of a Donald viewport in the scout window caused the Eden interpreter to shutdown without any explanation. Inside the Donald window I was drawing a very simple circle and since Eden does not generate an error log I found it impossible to trace the error.

3.6 Iteration Six

In this final iteration I am going to add a table for the resources, consolidate the code and generally finish of the model. The resources table will allow the project manager to change the hourly fee rate for a resource as well as the calendar they work to. Any changes to the resources will automatically roll up to the project dashboard. E.g. if a resource changes more money per hour then clearly the project costs will increase. In addition I am also going to add some utility functions that allow the manager to execute some what if scenarios. E.g. how many days will the project take will it take if just one resource works on the project full time?

Findings & Results

During a review of my study with Dr Meurig Beynon I was alerted to the availability of the 'Is' assignment for definitions. Instead of writing tasks = [task1, task2, task3] which causes tasks to have absolute values, we can write tasks is [task1, task2, task3] to create a dependency. Essentially the 'Is' assignment ensures that any change to an element on the RHS will force a reevaluation of the LHS. I reworked the definitions such that there is now a clear dependency hierarchy from calendars all the way up to tasks. Any change to a task or a resource triggers the proc bound to either the tasks or resources lists which in turn calls a proc to update the project dashboard. I think the message here is that with EM one is constantly learning and is therefore constantly fixing previous misconceptions and mistakes.

As you can see in figure 3.6.1 the final model is now fully functional and real construal's can be obtained by executing some of the possible experiments. The project manager can click on the 'two (2)' resources button and all tasks will be assigned to just two resources. Clicking on the button again will select two different resources. Since resources work to different calendars, the dashboard and traffic light status will change accordingly. Every resource can be forced to work to a specific calendar by clicking on the name of that calendar in the second experiment. The question of how long the project will take if everyone works in panic mode can then be explored.

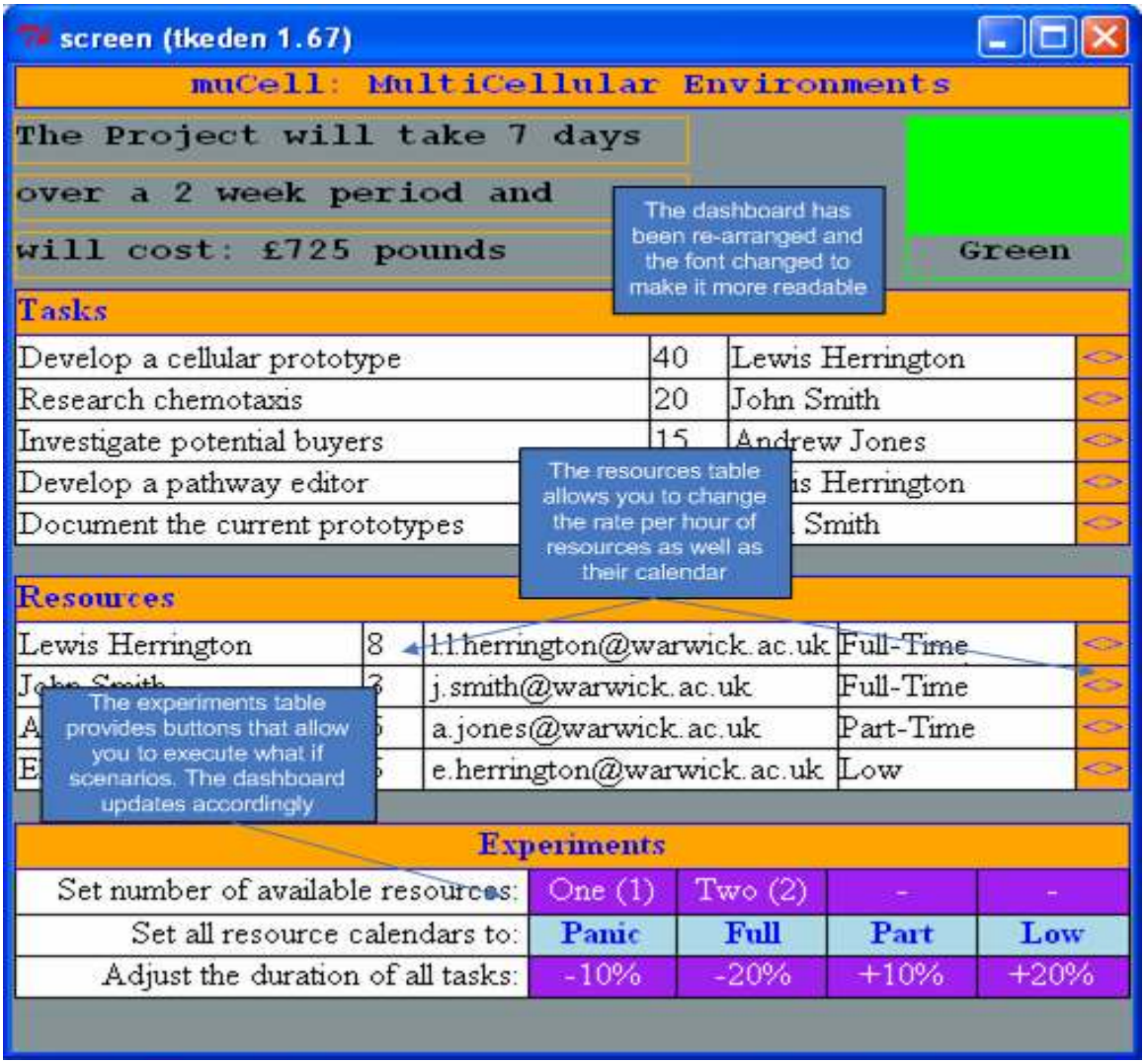

Figure 3.6.1 – Iteration six showing the interface of the final model

Problems Encountered

Unfortunately the 'Is' assignment cannot be used to relate a definition to a function, when I wrote ***totalcosts is sumTotalCosts()*** the interpreter entered a permanent loop. I tried ***totalcosts = sumTotalCosts()*** but this is only executed once and is not effected by changes to resources or task properties. As a result I still have to call a single function to refresh the dashboard when anything changes. A more fundamental problem encountered was that even when using the 'Is' assignment, not all of the relationships are maintained. Though each task is defined as ***TaskN is [Name, Duration, Resource]***, the dependency is not holding and is instead treated by the interpreter as TaskN = [Name, Duration, Resource]. This means that when we change a resource pay rate we have to update each task. Further investigation found that the dependency holds until we change an element in the task eg ***TaskN[3] = New***

Resource. I found in the end that I had to make a compromise between the integrity of having a dependency hierarchy with making the model do something useful.

Conclusions of the Study

When I started this study it was my goal to establish the underlying principles behind modelling a project lifecycle as an empirically based model. Throughout the study I have found a number of possible ways of achieving a goal but each with a slight restriction or quirk that has either required a re-negotiation of objectives or a complicated workaround. My original vision of a perfect dependency hierarchy where a change at one level automatically rolled up along the project proved difficult to implement.

What I have achieved in the study however, is to model a simple project lifecycle that allows the user to answer a set of 'what if' scenarios. The effect of changing task duration or the assigned resource can be immediately observed in the dashboard summary. The interface is much more expressive than a simple spreadsheet as the user can observe directly the effect of tweaking parts of the model without having to run complicated reports. As a side benefit what I feel this study and indeed this paper also offers is effectively a beginner's guide to Empirical Modelling. By stepping through the iterations one by one and viewing the definitive scripts along the way one can very quickly appreciate how to go about developing a model. In terms of further work, I suggest looking into the possibility of networking the model. What I think would be beneficial would be for the project to have a percent complete definition based on task progress entered by remote resources. Other possibilities would include email notifications for resources as well as more complex definitions for the dashboard.

Acknowledgements

I would like to thank Dr Meurig Beynon of the Empirical Modelling group at the University of Warwick for reviewing the fifth iteration of my model and providing advice that proved invaluable while working on the sixth and final iteration.

References

EM Principles (2006) 4th May 2008.
http://www2.warwick.ac.uk/fac/sci/dcs/research/em/intro/principles/

What is EM? (2006) 4th May 2008.
http://www2.warwick.ac.uk/fac/sci/dcs/research/em/intro/whatisem/

Empirical Modeling Glossary (2006) 4th May 2008.
http://www2.warwick.ac.uk/fac/sci/dcs/research/em/intro/glossary/

Key EM Concepts (2006) 4th May 2008.
http://www2.warwick.ac.uk/fac/sci/dcs/research/em/intro/glossary/#keyconcepts

Eden (2006) 4th May 2008.
http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/eden/

David Gooding. *Experiment and the Making of Meaning* (*Luwer Academic Publishers* 1990)

W.M.Beynon, Steve Russ and Willard McCarty. Human Computing: Modelling with Meaning. Literary and Linguistic Computing 21(2), 2006, 141-157

W.M.Beynon, R.C.Boyatt and S.B.Russ. Rethinking Programming. In Proceedings IEEE Third International Conference on Information Technology: New Generations (ITNG 2006), April 10-12, 2006, Las Vegas, Nevada, USA 2006, 149-154

W.M.Beynon. Liberating the Computer Arts. Proc DALI'2001, First International Conference on Digital and Academic Liberty of Information, University of Aizu, Japan, March 2001 (invited paper, 25pp). [072]

Kiran Fernandes, Vinesh Raja, John Keast, W.M.Beynon, Pui Shan Chan and Mike Joy. Business and IT Perspectives on AMORE: a Methodology for Object-Orientation in Re-engineering Enterprises. In Systems Engineering for Business Process Change: New Directions, (ed P. Henderson) Springer-Verlag 2002, ISBN 1-85233-399-5, 274-297