A Reference Architecture for Instructional Educational Software

Janelle Pollard and Roger Duke
School of Information Technology and Electrical Engineering
University of Queensland
St Lucia, QLD 4072, Australia
janellep@bigpond.net.au and rduke@itee.uq.edu.au

Abstract

Our extensive research has indicated that high-school teachers are reluctant to make use of existing instructional educational software (Pollard, 2005). Even software developed in a partnership between a teacher and a software engineer is unlikely to be adopted by teachers outside the partnership (Pollard, 2005). In this paper we address these issues directly by adopting a reusable architectural design for instructional educational software which allows easy customisation of software to meet the specific needs of individual teachers. By doing this we will facilitate more teachers regularly using instructional technology within their classrooms.

Our domain-specific software architecture, Interface-Activities-Model, was designed specifically to facilitate individual customisation by redefining and restructuring what constitutes an object so that they can be readily re-used or extended as required. The key to this architecture is the way in which the software is broken into small generic encapsulated components with minimal domain specific behaviour. The domain specific behaviour is decoupled from the interface and encapsulated in objects which relate to the instructional material through tasks and activities. The domain model is also broken into two distinct models - Application State Model and Domain-specific Data Model. This decoupling and distribution of control gives the software designer enormous flexibility in modifying components without affecting other sections of the design.

This paper sets the context of this architecture, describes it in detail, and applies it to an actual application developed to teach high-school mathematical concepts.

Keywords: Reference Architecture, Educational Software, Instructional Software.

1 Introduction

In the mid 1990's many researchers voiced a need to study why there was such a slow uptake of computers (and technology in general) in classrooms (Becker and Pence, 1996, Marcinkiewicz, 1993, Rosen and Weil, 1995). The subsequent studies revealed several major factors contributing to the acceptance or rejection of the use of computers in classrooms. A summary of these factors are:

- beliefs about teaching and pedagogy (e.g. established classroom practices and the intent of curriculum and syllabus);
- access to computers and software;
- relevancy of software;
- time to plan instruction;
- professional and practical computer training;
- technical and administrative support.

(Clark, 2000, Ertmer et al., 1999, Forgasz and Prince, 2001, Norton and Cooper, 2001, Norton et al., 2000, Sarama et al., 1998)

Forgasz and Prince signal in the conclusion of their study the importance of establishing why teachers are using generic software more extensively than mathematics-specific software (Forgasz and Prince, 2001). Our case study work reveals that teachers desire software which:

1. teaches concepts they do not feel are effectively taught with current resources, and
2. are grounded not only in teaching pedagogy in general, but in their personal teaching pedagogy.

(Pollard and Duke, 2002, Pollard and Duke, 2001, Pollard and Duke, 2003)

It is apparent that most current mathematics-software does not meet these requirements. As a consequence teachers tend to use generic software which allows them the flexibility to personally develop their own lessons/ activities on the computer. However this also requires a fairly sophisticated level of computer expertise which most teachers do not currently posses; they often lack the technical skills to personalise/ tailor generic software products to their needs. Our proposal is to change the software to meet the teachers' needs, instead of asking the teachers to tailor complex software packages or change their teaching style to fit the software's prescribed teaching style.

This is not viewed as a complete solution as we subscribe to the view that teachers' use of technology evolves as they gain experience (Hadley and Sheingold, 1993, Marcinkiewicz, 1993). Initially teachers require technology which closely supports their existing teaching styles; this requirement softens over time as their beliefs are expanded with personal experience of what is possible

Copyright © 2005, Australian Computer Society, Inc. This paper appeared at the *South East Asia Regional Computer Confederation (SEARCC) 2005: ICT: Building Bridges Conference*, Sydney, Australia, September, 2005. Conferences in Research and Practice in Information Technology, Vol. 46. Graham Low, Ed Reproduction for academic, not-for profit purposes permitted provided this text is included.

with the technology (Hadley and Sheingold, 1993, Hyde et al., 1994, Marcinkiewicz, 1993, Sandholtz et al., 1997). However to facilitate more teachers beginning to integrate computers into their classroom lessons, we must establish a means of creating educational instructional software which, while well grounded in educational philosophies and teaching pedagogies, remains flexible enough to be tailored to meet each teacher's individual teaching style (Norton and Cooper, 2001).

We propose a domain-specific software architecture (also referred to as a reference architecture) (Hofmeister et al., 2000) which allows educational software to be decoupled into three sections: its interface, the associated educational instructional activities, and the domain-specific data model. We believe software developed using this architecture will be able to be readily modified for each individual teacher.

Our architecture is designed for small mathematic-specific didactic software packages. Didactic packages are tightly coupled to the learning of specific mathematical topics. They sacrifice the ability to solve general mathematical problems in order to concentrate on teaching, often incorporating problems and exercises into their designs (Pree, 1995, Crowe and Zand). Types of didactic packages include assessment, drill, tutorial, game, presentation and situational and procedural simulations (Pollard, 2005).

The IAM architecture is intended to be used when flexibility to change parts of the interface, activities and/or data model is essential, and only small amounts of data are to be stored. The implication is that this architecture should be used when the interface and activities are considered to be more important than the efficiency of managing and accessing the underlying data.

2 Reference Architecture

The proposed reference architecture, Interface-Activities-Model (IAM for short), aims to decouple educational tasks and activities from their domain-specific data model and the interface, to facilitate efficient modification and re-use.

2.1 Overview of the IAM architecture

A conceptual view of the proposed architecture is given in Figure 1.

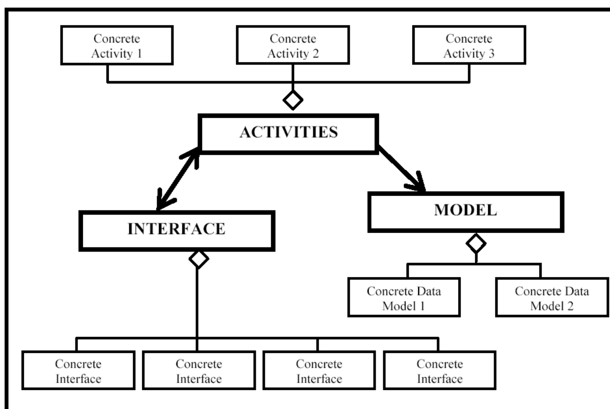

Figure 1: Conceptual view of the IAM architecture

Three existing design patterns assist in the formulation of this IAM reference architecture, namely the Façade, State and Strategy patterns (Gamma et al., 1995).

Their involvement in the structure of the design is discussed in Section 3.2. The elements contained within the architecture, and their meaning, are explained in Table 1.

Interface *The overall facade for the interface components*
Concrete Interface Component *A self contained component object which is responsible for one complete section of the interface. (e.g. a panel with a group of textfields.)*
Activities *The front handler for the educational activities*
Concrete Activity *A collection of tasks, where a task is a single request by the computer to be fulfilled by the student (e.g. the computer asks the student to convert a table of data points into ordered pairs). A Concrete Activity is a collection of these tasks which together satisfy one logical educational aim.*
Model *The front for all possible domain-specific data models.*
Concrete Data Model *The domain-specific data model/repository stores the underlying data required for the activities. (e.g. a collection of information about all the attributes of a straight line, gradient, intercepts, etc. or a collection of questions and answers.)*

Table 1: Summary of elements

2.2 Architectural component interactions

The following four points summarize how the architectural components communicate:

1. Activities can interact with both Interface and Model. However Interface and Model have no knowledge of each other's existence. Activities retrieves information from the Model and passes necessary information to the Interface.
2. Each Concrete Interface Component informs Interface of changes to itself, which in turn informs Activities (as per the State pattern).
3. Each Concrete Activity has a unidirectional link to Model (as specified in the Strategy pattern).
4. Each Concrete Activity has a unidirectional link to the Interface façade.

This is further explained in Figure 2.

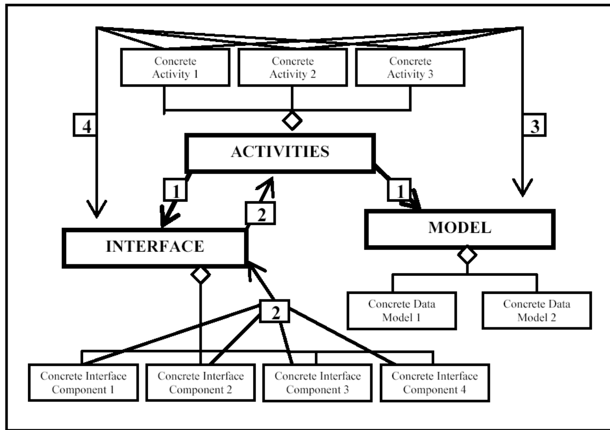

Figure 2: Architectural component interaction

2.3 Example

A portion of the software, "Exploring the relation between line representations", which was co-designed with a high-school mathematics teacher as part of a broader case study, is used here to explain the IAM architecture. (See (Pollard and Duke, 2003, Pollard, 2005) for a description of the case study and details of how the software was designed.)

The software aims to assist year 10 students grasp the concept that a table of data, ordered pairs, a graph and an equation are four different ways of representing the same mathematical information. The overall structure of the software with relation to the IAM architecture is shown in Figure 3.

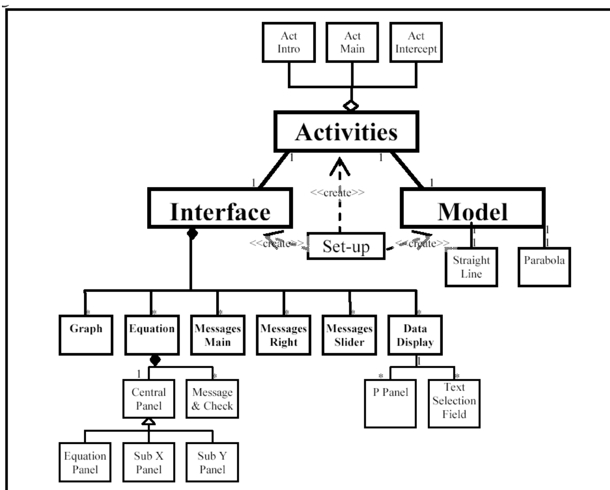

Figure 3: Example of IAM Architecture

The *Set-up* object creates *Interface*, *Activities* and *Model*, and the relationships between them. These classes then initialize their respective concrete components.

From Activities view the interface of the "Exploring the relation between line representations" software would be referred to in the following terms (as shown in Figure 4).

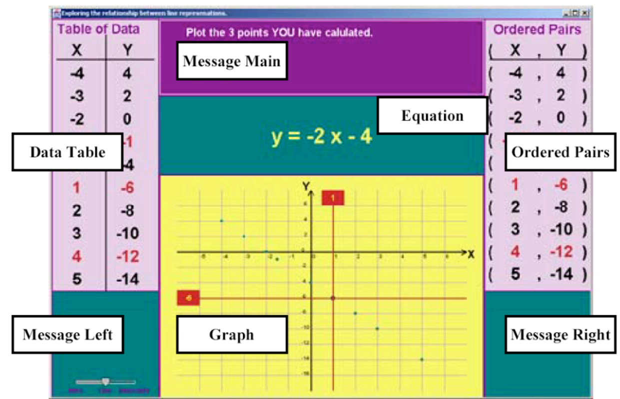

Figure 4: Example of how Activities views the interface.

An example of a task, as seen in Figure 5, requires the student to input all the values in the Table of Data as ordered pairs. The task is complete when all values have been correctly entered.

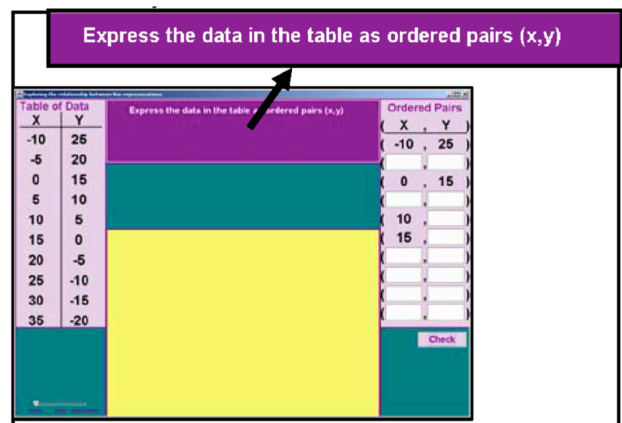

Figure 5: Example of a task

An example of a completed Concrete Activity (specifically Activity Intro) would be when the following tasks are fulfilled:

1. The data table has been expressed as ordered pairs.
2. The points have been plotted on the graph.
3. Three x values have been substituted into the equation.
4. Three y values have been substituted into the equation.

These four tasks achieve the overall logical aim of relating the table of data to the ordered pairs, to the graph and to the equation. The main activity aims to teach students to calculate their own x and y values and represent them in the four forms. The final activity, intercepts, aims to help students calculate the x and y intercepts and represent them in the four forms.

In Summary, each Concrete Activity holds the behaviour of the application for an educational instruction. A Concrete Activity (a single education exercise) is made up of a series of steps/tasks.

Each step has:

- an initial state (the state in which the interface and activity starts, e.g. whether the graph is visible or not),
- conditions for progressing through the step (error handling, etc.), and
- a condition for checking if the step is completed (what state the application will be in at the end of the step).

The *Model* in this example has two possible domain-specific data models associated with it, Straight Line and Parabola. These models are responsible for producing all required data about a function whether it is a straight line or a parabola. Every time a new straight line function is calculated, the information/data in Table 2 is calculated and stored for access by the Activities. In this example the data model remains unchanged for the duration of its life. Once created, the information is accessed, but not changed. However, in other applications it may be necessary to modify the data model. This would occur through *Activities* sending updates to the *Model*.

The formula The values for m,c,a and/or b in $y = mx + c$ or $y = ax^2 + bx + c$.
The intercepts y0 and x0 when x = 0 and y = 0 respectively.
Min and Max values The minimum and maximum x and y values of the function within the specified range.
StepX and StepY The incremental steps for x and y (which depend on the function).
Array of X and an array of Y points A list of x and y points which will be used in the data table, ordered pairs to be plotted, etc.

Table 2: Data Summary

2.4 Consequences of the IAM Architecture

The Interface-Activities-Model has the following benefits:

- Activities (and tasks) are encapsulated making them easy to rearrange/modify/add or remove, at the task or activity level.
- Interface components can be easily added/removed or modified since they contain minimal behavioural functionality. (Essentially, little beyond displaying information passed to the component and retrieving the component's current internal state is required.)
- The Model (specifically the data model) can be replaced or modified without affecting the Interface or Activities as long the data model's interface remains constant.

The drawback of this design is an increase in the amount of information/data required to be passed between objects. This is a reasonable sacrifice for instructional educational software, which generally has high interface costs but low data storage requirements. See Section 4 for further discussion of the architecture's viability.

3 IAM Architecture and Existing Patterns

This reference architecture is built on three main design patterns, Façade, State and Strategy. It also maintains two types of current state information structures and one domain specific model. This section describes how the design is constructed and governed by the underlying design patterns, and the impact of distributing and maintaining information.

3.1 Overview of Information Storage

To fully understand the significance of our architecture one is required to look at the location of information storage and what is meant by the terms 'model' and 'current state'. In this design there are three areas which hold information: Concrete Interface Components, Activities and Concrete Data Models. The distribution of the data is summarized in Figure 6. The information stored in each of these locations is very different.

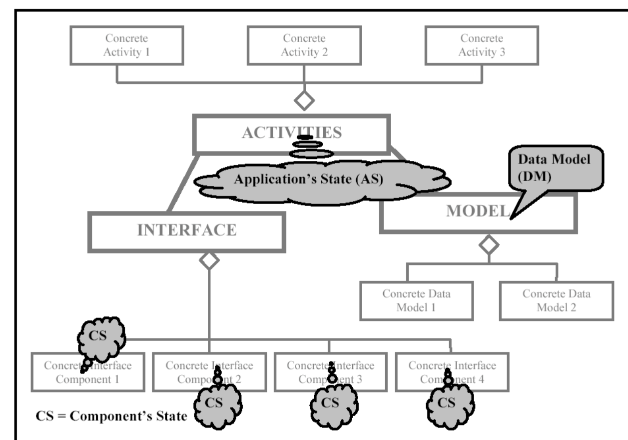

Figure 6: Information storage distribution

The data storage areas are defined as follows:

- *Component's State (CS)* - is stored within the Concrete Interface Components and holds the current state of the component e.g. the current values of the text-fields in the panel.
- *Application's State (AS)* - is stored within Activities and contains the current state of the activity being performed, e.g. the questions which a student has answered correctly, incorrectly or are currently unanswered.
- *Data Model (DM)* - is stored within the Concrete Data Model and holds the solution and initialization data for the activities, e.g. the details of the function along with the current set of data points.

The key to why this architectural design strongly promotes component reuse is in how different data is stored in the various locations. Concrete Interface

Components are the easiest to re-use since they store their own current state data but rely on other classes to determine their behaviour. For example two columns of text-fields could be used to collect ordered pairs for plotting on a graph. Alternatively two columns of text-fields could be used to collect odds for a race day sweep, storing the horse's number and their current pay out. How the information/data is handled differs greatly, but the interface input and display remains the same. The way the data is handled is the responsibility of Activities. As such the domain specific functionality is totally decoupled from the interface components, which ensures that the interface components are generic and easy to re-use.

Activities holds the entire current state of user interaction with the educational instruction (Application's State). As a consequence the Concrete Activities are easy to modify. Concrete Activity only holds enough information to interface with the Application's State, Interface and Data Model. The Concrete Activities are written at a high level, focusing on defining the behaviour of the system.

The final data storage location is the Data Model. The Data Model stores a static snapshot of all the data required to fulfil the Concrete Activities information needs. The Data Model can be likened to a database of useful information; a repository. For example the Data Model may contain one hundred questions and their answers. The Application's State holds the subset of these hundred questions which are currently being presented to the student. (Depending on the requirements of the software package, the Application's State may also have to hold a history of the previous application states so as to facilitate backtracking and other more advanced history features.) The important point is that the dynamic state of the application is stored in the Application's State, whereas the more static information/data is held in the Data Model repository.

One final point about the Data Model is that the information does not have to be static throughout the life of the application. An example is a Data Model repository of information about straight line functions (as discussed in Section 2.3). Each function stores a collection of relevant information, (e.g. its gradient, or a set number of data points, etc.). The Concrete Data Model holds the algorithms for dynamically generating the information about a straight line and deposits it into the Data Model repository. At a later stage the Concrete Data Model may be asked to generate another straight line. This information would be placed in the Data Model repository, overwriting the previous information in the Data Model. So while the Data Model repository itself is not static, only one Data Model exists at any given time. These dynamic aspects can be exploited by the algorithmic qualities of mathematics, where questions can be randomly generated to meet a supplied criterion.

In theory, any dynamically generated Data Model repository could be converted into a static Data Model repository by adding all the information from a pre-defined number of iterative initiation calls to the Concrete Data Model at the start of the software set up, thus forming a single static Data Model repository. For example one could call the straight line function Concrete Data Model five times and have all the information placed in the Data Model repository. However, it is simply more efficient on memory and storage space to generate each entry of the repository/database as required by relating the data to an algorithmic process (stored in the Concrete Data Model).

3.2 Related Design Patterns

3.2.1 Façade Design Pattern

The disadvantage of making the interface components so generic is that *Activities* needs to know about the interface (e.g. which component holds the data for the current x value being substituted into the equation). This disadvantage is greatly reduced by utilizing the Façade design pattern (Gamma et al., 1995). *Interface* becomes the Façade to the *Concrete Interface Components*, as shown in Figure 7.

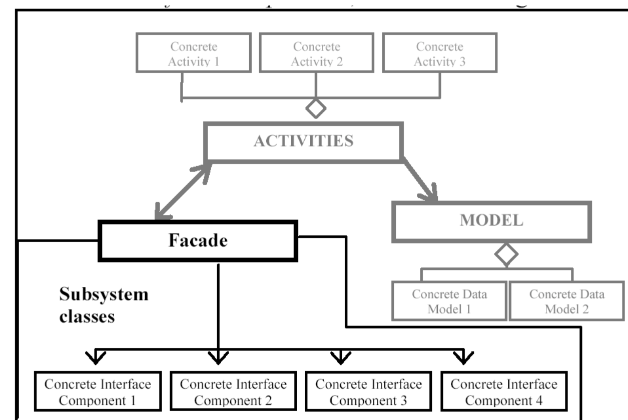

Figure 7: Application of the Façade Design Pattern

The intent of the Façade design Pattern (Gamma et al., 1995) is to *"provide a unified interface to a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use."* The façade provides the single entry into the interface and its components. As a consequence the interface holds all possible update and retrieval methods which can be used to access the interface. This allows the activities to access or change the Component State of each of the *Concrete Interface Components* without explicitly or directly having to know what components exist and what data they maintain.

The Interface object forms a very important function. It is the façade between the way Activities views the interface and how the interface views itself. Why two views? Because Interface is the merging junction between two different perspectives. The Concrete Interface Components are components chosen by software engineers and are typically made up of fundamental building blocks of programming (e.g. text-fields, sliders, radio buttons, etc.) and are subsequently thought about in these terms. The Concrete Activities on the other hand are descriptions of how the student is to interact with the computer and in turn how the computer is to respond to the student. At this level we are concentrating on how the educational instruction will occur. From Activities' perspective the ideal would be to enable teachers to write the educational instruction for the software. For this to

occur one requires two different views of the interface, namely that of the teacher and that of the software engineer.

Taking a concrete illustration from the example presented in Section 2.3, there are two similar panels, the Data Table and the Ordered Pairs panel. A teacher would refer to the interface when describing an educational activity in terms like 'we will ask the students to re-write the points in the Data Table as ordered pairs in the Ordered Pairs list.' Since the functionality of the two panels are very similar, the underlining interface components do not distinguish between the Data Table and the Ordered Pairs panel. They are both instances of the Data Display component (see Figure 8). The Interface façade allows Activities to refer to the Data Table and Ordered Pairs panels and matches them up with their subsequent instances of Data Display. See Figure 9.

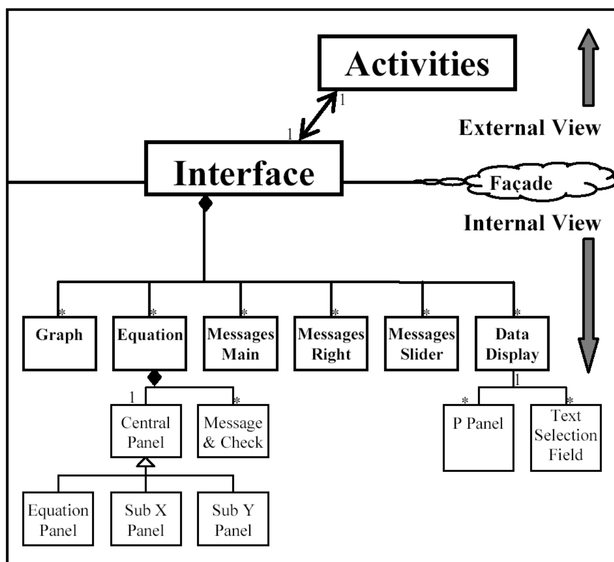

Figure 8: The Interface Façade, Concrete Interface Components and Activities

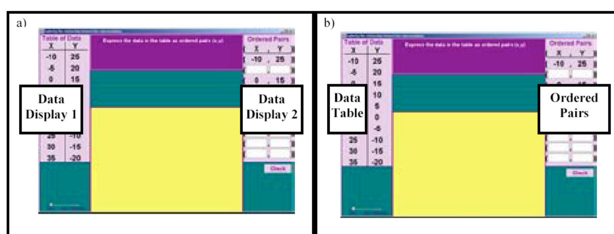

Figure 9: a) Concrete Interface Components naming of panels b) Activities reference to panels

Another example of how the Concrete Interface Components differ from the Activities when referring to the same objects is the way messages (activity instruction and feedback) are handled. For the purposes of re-use and ease of referring to parts of the interface, the Interface façade provides a conversion between the two views for handling the setting and clearing of messages on the interface. Interface contains three methods: setMessage, clearMessage and clearAllMessage.

Instead of referring to the messages via the panel (or sub panel as in the case for the equation panel), Activities refers to them by location (see Figure 10). For example, if a message is to appear under the Ordered Pairs panel, a Concrete Activity sends the following command to Interface - setMessage ("Your message", Right, Bottom). Interface sends the message to be displayed to the display panel closest to the specified area.

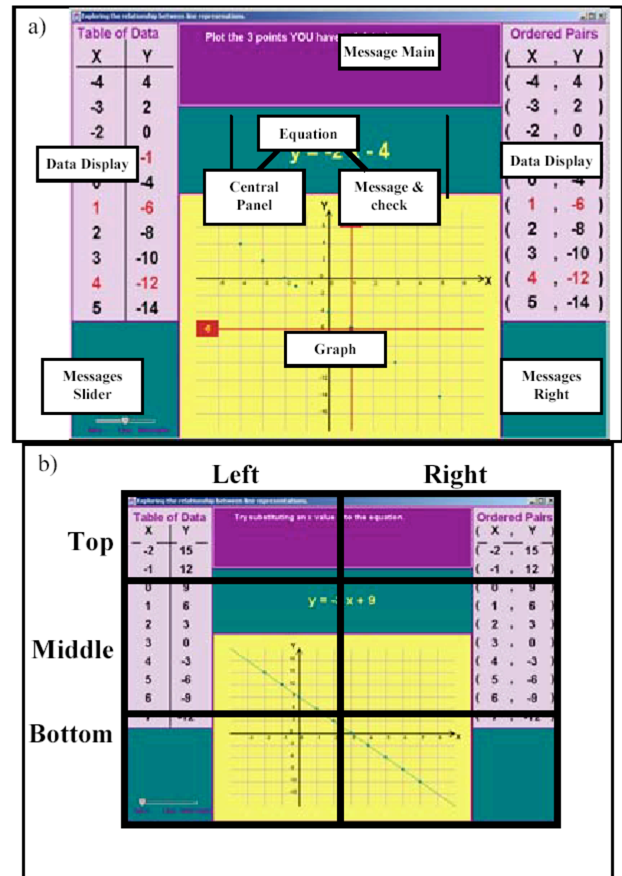

Figure 10: a) Interface's internal view of the panels, b) Activities' view of the interface for messages

3.2.2 State Design Pattern

Most *Concrete Interface Components* will typically be structured in conformance with the Model-View-Controller (MVC) pattern (Gamma et al., 1995, Pree, 1995) (see Figure 11). Because the model in this MVC usually contains the "application-specific state **and** behaviour" (Pree, 1995) this is a problem since in our case the behaviour is not stored with the specific component. Instead the model (or controller) directly contacts Activities to request its behaviour. This actually makes designing easier since the behaviour of the software is described at the (top) level at which the student and computer interact, rather then at the lower level of inter-component interactions.

Each *Concrete Interface Component* informs *Activities* that it has received user input or interactions. The *Concrete Activity* determines the desired response to the event and informs the Interface façade of what (if anything) has to be updated. This relies on an underlying State Pattern (Gamma et al., 1995). The intent of the State Pattern is to *"allow an object to alter its behaviour when its internal state changes. The object will appear to*

change its class." Figure 12 relates the State pattern to the IAM architecture.

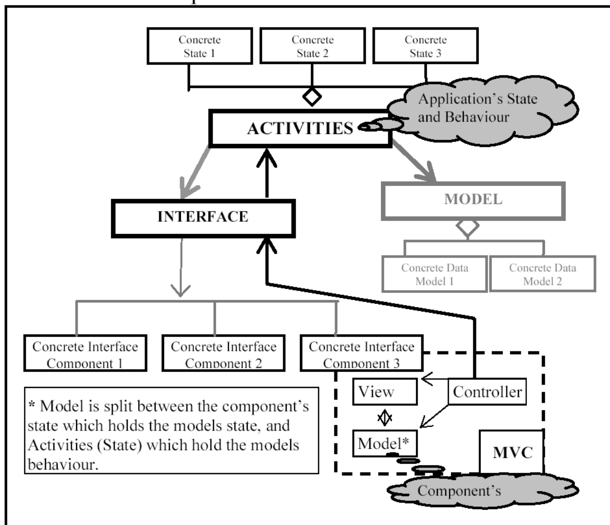

Figure 11: MVC and Domain Model

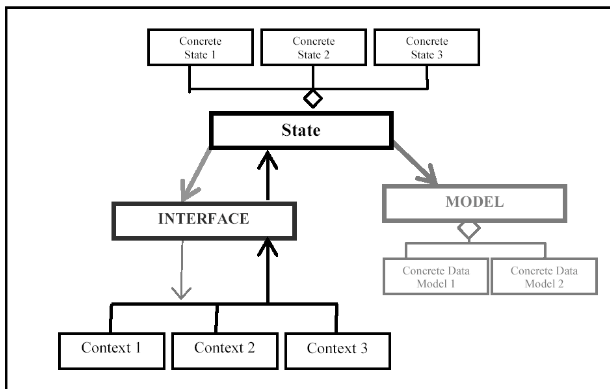

Figure 12: Application of the State Design Pattern

This pattern allows the *Concrete Interface Components* to send *Activities* a message when they are changed, without knowing in what way the event will be handled. *Activities* calls on the *Concrete Activities* to handle the requests depending on what state it is in, e.g. in our case study whether it is performing Activity Intro or Activity Intercept.

To summarise; a *Concrete Interface Component* would typically be patterned as a MVC, however the model would only hold the component's current state (CS - Component State). The component's behaviour is separated from the CS and controlled by *Activities*. Hence controllers in the *Concrete Interface Components* inform *Activities* if they have been triggered (as well as their model and/or view if necessary). *Activities* holds the application's specific current state (AS - Application State) and the behaviour associated with the activities. *Activities* passes events to its concrete state (a *Concrete Activity*) where the event is handled.

This would normally involve:

1. collecting any required information from the *Interface*;
2. comparing the information with the Data Model (DM);
3. updating the Application's State (AS) in *Activities*; and
4. updating the *Interface* to reflect the changes make to the Application State (AS).

3.2.3 Strategy Design Pattern

Strategy Pattern (Gamma et al., 1995) intends to *"define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."* In our circumstance this is how Activities accesses the Data Model (DM). There may be one or more *Concrete Data Model* and as such the Singleton pattern (Gamma et al., 1995) may be required to ensure only one *Concrete Data Model* is created at a given time, depending on the application.

The Strategy pattern describes in detail how to achieve this aim and the issues associated with the encapsulation of the algorithm. The pattern could also be applied to situations where the "Concrete Strategy" is not an algorithm, but instead a database or another domain-specific data model, provided that the objects' interfaces all contain commonalities and can therefore be made interchangeable. Figure 13 demonstrates how this strategy can be applied by the architecture.

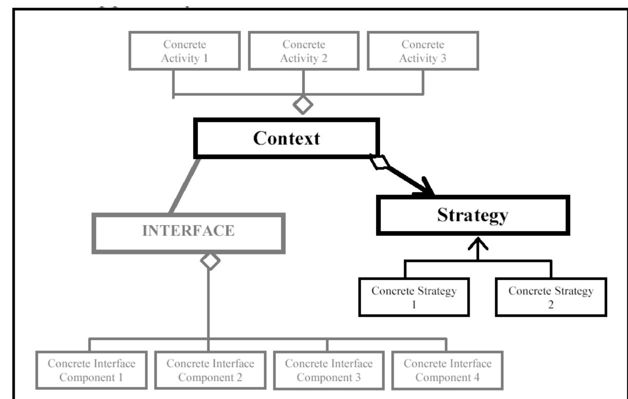

Figure 13: Application of the Strategy Design Pattern

3.3 Comparing IAM to MVC

Interface-Activities-Model (IAM) may appear to be very similar to Model-View-Controller (MVC). Wolfgang Pree in "Design Patterns of Object-Oriented Software Development" (Pree, 1995) defines MVC as in Figure 14.

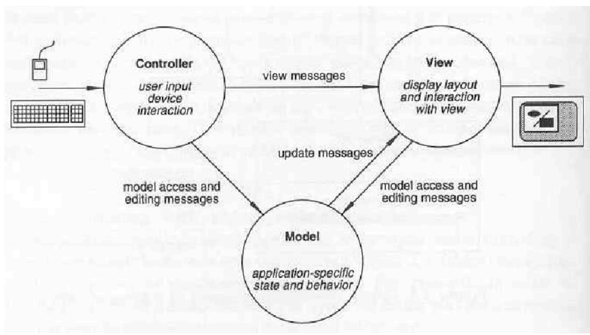

Figure 14: Figure Copied from "Design Patterns of Object-Oriented Software Development" (Pree, 1995)

The Model (in MVC) stores application-specific data. For example, a text processing application stores the text characters in the Model; a drawing application stores a description of the graphic shapes in the Model.

A View (in MVC) component presents the Model on a display, usually the screen. Any number of view components might present the Model in different ways. Each View has to access the information stored in the Model.

Finally, the Controller (in MVC) handles input events such as mouse interaction and key strokes. Each View has an associated Controller that connects the particular View with input devices such as the mouse and keyboard (Pree, 1995). When comparing MVC to IAM by aligning View with Interface, Controller to Activities and Model to Model, the differences between the architectures become apparent.

In MVC the View refers to the drawing of the graphical information on the screen. In IAM the Interface controls and manages the Concrete Interface Components which are responsible for displaying the graphical information.

In MVC the Controller is responsible for informing the Model and/or View of the user's event actions. In IAM Activities does not listen for user events. These events are passed to Activities via the controllers in the Concrete Interface Components.

In IAM, Activities determines how the software is to respond to the user's interaction and updates the Application's State. This is the role associated with the Model in MVC. However the Model in IAM refers specifically to only the domain-specific data (a repository of information), not the application's state or behaviour. The Application's State and responsive behaviour is stored in Activities.

These differences demonstrate that, while there are superficial similarities between MVC and IAM, in essence IAM is a distinct architecture. This is further amplified when one realises that MVC is designed to separate the dependencies within a component whereas IAM is designed to decouple components into three separate parts.

3.4 Summary of the structure of IAM

The IAM architecture is based on a three way split of control. The Interface is in charge of creating a unified view of the application's interface, hiding the underlining detail. Activities is responsible for providing the behaviour of the application and relies on the Model for its knowledge base.

Each Concrete Activity holds the behaviour of the application for an educational instruction. A Concrete Activity (a single education exercise) is made up of a series of steps/tasks. Each step has:

- an initial state (the state in which the interface and activity starts, e.g. whether the graph is visible or not),
- conditions for progressing through the step (error handling, etc.), and
- a condition for checking if the step is completed (what state the application will be in at the end of the step).

The checking of the step's progress relies on accessing the information stored in the following three locations:

- the Application's current State (located in Activities),
- the Data Model (located in the current Concrete Model), and
- the Current State (located in the Concrete Interface Components, but accessed through the Interface Façade).

Interface is the façade between the two views of the interface; internal (the Concrete Interface Components with various naming conventions and various degrees of visibility) and external (single unified view of the interface based on what the software looks like from the outside). The separation of views allows Activities to interact at a very high level without being concerned with how requests are implemented, resulting in a strong decoupling of desired application's behaviour and the actual implementation of these behaviours. The high level view makes it simpler to design education instruction, which in turn makes the didactic material of instructional software quickly changeable.

Since the Concrete Activities hold the application's behaviour, the Concrete Interface Components are devoid of behavioural code dependencies. They are solely constructed from:

- a default state,
- an ability to update/modify their Current State, when informed how, and
- an ability to inform Interface when changes to their Current State occur, and can return requested information about their Current State.

The Concrete Models are repositories of knowledge. In most mathematical software they will be based on an algorithmic process for generating the data dynamically. However, they may also be static databases of information. The data/information/knowledge is accessed through the Model which acts as an interface/façade if there is more than one Concrete Model available.

4 Testing of the IAM architecture

This architecture was tested in three ways. Firstly by interviewing a collection of high school teachers to determine what modification to the software, presented in Section 2.3, would they require in order for them to consider using the product. Secondly these changes were tested against the IAM architecture design by the original software engineer. Thirdly, other software engineers were asked to identify how to make the suggested changes based on the IAM architectural design of the software example. The results were as follows:

The software example from section 2.3, which was developed in partnership with a teacher, produced a software product that the teacher valued. The partnership produced a software package that 78% of the teachers interviewed thought was appropriate as a software topic for computers and that 67% could see had the potential to be applied to their present situations. Despite this, 89% of teachers interviewed desired changes to the software. There were three types of changes:

- minor interface changes,
- new activities, and
- changes in functionality.

The most common changes requested related to changing the interface. The IAM architecture proved very effective at easily facilitating such changes since the architecture decouples (as much as possible) the functionality of the software from the displaying of the information and there are not dependencies between Concrete Interface components.

New Activities typically caused changes in the Interface, Activities and the Model. But because of the decoupling of the components few changes outside the immediate impact area of the activities were required, leaving the other activities unaffected.

Even small changes in functionality typically required wide modification to components unless the changes were specifically incorporated into the original design of the components. Further research needs to be focused on identifying what functionality needs to be incorporated into the design in order to create good, re-usable components.

The final testing of the architecture involved five test subjects with an undergraduate degree in computer science. They were shown a selection of screen shots of the example software and its class diagram (see Figure 3). Without any explanation about the architecture, the test subjects were asked to identify which class (or classes) they would need to change to achieve the suggested modifications, where the modifications were those suggested in the section above (in a random order).

Over half of the classes for the modifications were identified successfully. The minor interface changes were the most accurately identified. Four of the five test subjects grasped the role of Activities in generating modifications related to the introduction of new activities but typically also named Concrete Interface Components which would not need to be changed.

Changes in functionality were the most revealing of how the test subjects interpreted the class diagram. Most test subjects did not grasp from the class diagram that the Concrete Interface Components are objects which do not contain behavioural methods. However after this was explained they more accurately identified which components needed to change in order to reflect the desired changes in functionality.

The IAM architecture withstood the suggested modifications by the interviewed teachers. Less than a quarter of the suggested changes required the re-design of components, where well over half of these same changes would have cause major re-designing if we had used the traditional Frame-Panel architecture. Furthermore, the IAM architecture, with the exception of the separation of the behaviour of the interface into Activities, proved relatively intuitive to other software engineers.

5 Conclusions

Our IAM architecture has created a way of developing software which can be readily adapted to meet teacher's epistemological and pedagogical orientation and their teaching practices. In order to make software adaptable, we modified the structure of the software architecture to allow for easy coupling and de-coupling of various aspects of a software package.

The IAM architecture decouples software into three sections: its interface, the associated educational instructional activities, and the domain-specific data model. The key to the IAM architecture is the way in which the software is broken into small generic encapsulated components with minimal domain specific behaviour. The domain specific behaviour is decoupled from the interface and encapsulated in objects which relate to the instructional material through tasks and activities. In essence we create two types of objects: those with their current state (Concrete Interface Component) and those which control the behaviour of other objects (Concrete Activities). This is made possible by the Concrete Activities accessing the Concrete Interface Component through the Interface facade. Hence from the behavioural objects (Concrete Activities) perspective, they are only defining the behaviour for one object, Interface (the Interface façade then passes the behavioural information to the appropriate Concrete Interface Component).

The domain model is also broken into two distinct sub-models: Application State Model and Domain-specific Data Model. This decoupling and distribution of control provides the software designer with flexibility in

modifying components without affecting other sections of the design.

The IAM architecture proved to be acceptable to other software engineers, with the test sample of software engineers correctly identifying more than half of the locations within the architecture that needed to be modified to implement suggested changes. With full documentation of the architecture we are confident that software modifications would become even more straightforward. The IAM architecture has been proven to readily handle:

- minor interface changes,
- new activities,
- changes in functionality, and
- changes to the type of software.

6 References

Becker, J. and Pence, B. (1996): Mathematics teacher development: Connections to change teachers' beliefs and practices *Proc. Proceedings of the 20th Conference of the International Group for the Psychology of Mathematics Education*, Spain, 103-117

Clark, K. (2000): Urban Middle School Teachers' Use of Instructional Technology. *Research on Computers in Education* **33**: 178-196.

Crowe, D. and Zand, H. (2000): Computers and Undergraduate Mathematics I: setting the scene. *Computers and Education* **35**: 95-121.

Ertmer, P. A., Addison, P., Lane, M., Ross, E. and Woods, D. (1999): Examining teachers' beliefs about the role of technology in the elementary classroom. *Journal of Research on Computing in Education* **32**: 54-73.

Forgasz, H. and Prince, N. (2001): Computers for secondary mathematics: Who uses them and how? *Proc. The Australian Educational Researcher*, Fremantle ACER.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*, USA Addison-Wesley.

Hadley, M. and Sheingold, K. (1993): Commonalties and distinctive patterns in teachers' integration of computers. *American Journal of Education* **101**: 261-315.

Hofmeister, C., Nord, R. and Soni, D. (2000): *Applied Software Architecture*, USA Addison-Wesley.

Hyde, A., Ormistion, M. and Hyde, P. (1994): In *Professional development for teachers of mathematics* 49-54Reston, USA.

Marcinkiewicz, H. R. (1993): Computers and teachers: Factors influencing computer use in the classroom. *Journal of Research on Computing in Education* **26**: 220-237.

Norton, S., Campbell, J. and Cooper, T. (2000): Exploring Secondary Mathematics Teachers' Reasons for Not Using Computers in Their Teaching: Five Case Studies. *Journal of Research on Computing in Education* **33**: 87-109.

Norton, S. and Cooper, T. (2001): Factors influencing computer use in mathematics teaching in secondary schools. *Proc. Proceedings of the Mathematics Education Research Group of Australasia (MERGA) - Numeracy and beyond*, (Eds, Bobis, J., Perry, B. and Mitchelmore, M.), Australia, 386-393

Pollard, J. (2005): A Software Engineering Approach to the Integration of Computer Technology into Mathematics Education. Ph.D. thesis. University of Queensland, Australia.

Pollard, J. and Duke, R. (2001): Effective Mathematics Education Software in the Primary School: A Teachers' Perspective *Proc. 6th Asian Technology Conference in Mathematics*, Melbourne, 177-186 ATCM Inc.

Pollard, J. and Duke, R. (2002): From Maths Problem to Program: What's the best path? , Brisbane, 195-206 Post Press.

Pollard, J. and Duke, R. (2003): Revelations in the design of Educational Mathematical Software *Proc. 8th Asian Technology Conference in Mathematics*, Taiwan, 169-177 ATCM Inc.

Pree, W. (1995): *Design Patterns for Object-Oriented Software Development*, USA Addison-Wesley.

Rosen, L. and Weil, M. (1995): Computer availability, computer experience and technophobia among public school teachers. *Computers in Human Behavior* **11**: 9-31.

Sandholtz, J. H., Ringstaff, C. and Dwyer, D. C. (1997): *Teaching with technology: Creating student-centered classrooms*, New York Teachers College Press.

Sarama, J., Clements, D. and Henry, J. (1998): Network of influences in an implementation of a mathematics curriculum innovation. *International Journal of Computers for Mathematical Learning* **3**: 113-148.