

Final project in DPF –
Design Patterns and Frameworks

Architectural Patterns for Interactive Systems

Abstract

Structuring an interactive system consisting of graphical user interface and problem domain components is problematic. Both graphical user interfaces and problem domain components are prone to change – especially in early system development phases. Attempting to solve this problem, we supplement current literature on architecture patterns for interactive systems by reframing two patterns (Model-View-Controller and Presentation-Abstraction-Control) and delineating two new patterns (Application Facade and Application Adaptor). The patterns are presented informally (in short narrative form) and formally (using precise specification). Based on the lesson learnt, we preliminarily discuss relationships, differences, and similarities between the four patterns

Instructor: Amnon Eden

eden@math.tau.ac.il

Klaus Marius Hansen

940589, marius@daimi.au.dk

Michael Thomsen

940574, miksen@daimi.au.dk

Table of contents

INTRODUCTION	6
PATTERN DESCRIPTIONS	7
Application Facade	8
Model-View-Controller (MVC).....	10
Presentation-Abstraction-Control (PAC)	12
Application Adaptor	14
A SAMPLE APPLICATION	16
Shared implementation.....	16
Application Facade	18
Model-View-Controller (MVC).....	21
Presentation-Abstraction-Control (PAC)	24
Application Adaptor	28
FORMAL SPECIFICATION OF ARCHITECTURAL PATTERNS	31
LanguagE for Pattern Uniform Specification (LePUS)	31
Precise Visual Specification (PVS).....	32
Precise Specification of Model-View-Controller (MVC).....	34
Discussion of Precise Specification	39
CONCLUSIONS	40
REFERENCES	41
APPENDICES	43

Table of figures

Figure 1. Lightweight architectural pattern template.....	7
Figure 2. Sample package diagram	7
Figure 3. The Financial History application	16
Figure 4. The problem domain model.....	17
Figure 5. The library code for Application Facade.....	18
Figure 6. The Application Facade for the input window.....	19
Figure 7. The Presentation for the input window	19
Figure 8. The library code for MVC	21
Figure 9. The MVC Model for Financial History	22
Figure 10. The MVC View for the input window.....	22
Figure 11. The MVC Controller for the input window	23
Figure 12. The library code for PAC	24
Figure 13. The PAC hierachy for our application	25
Figure 14. The top-level PAC agent	25
Figure 15. The barChart PAC agent.....	26
Figure 16. The input window PAC agent.....	26
Figure 17. The view coordinator PAC agent	26
Figure 18. The DataInt for the input window.....	28
Figure 19. The FuncInt for the input window.....	29
Figure 20. The Application Adaptor for the input window	30
Figure 21. Visual notation of LePUS (Eden, 1999)	32
Figure 22. Professor and student constraint diagram.....	32
Figure 23. Three-model layering of PVS.....	33
Figure 24. Role-model of a Model, having a set of observer, no abstract operations, and a set of abstract instances.....	33
Figure 25. LePUS specification of MVC	34
Figure 26. LePUS specification of the Observer Pattern.....	35
Figure 27. Precise Visual Specification of MVC role invariant	36
Figure 28. Possible specification of PAC.....	39

Table of Appendices

Appendix 1. The User interface code	43
Appendix 2. The structure of Facade	44
Appendix 3. The structure of MVC.....	45
Appendix 4. The structure of PAC	46
Appendix 5. The structure of Application Adaptor	47

Architectural Patterns for Interactive Systems

Introduction

A large body of software can be regarded as *interactive* – it supports communication between the user and the computer in a way where the user takes action and the system reacts accordingly (Newman and Lamming, 1995). In most modern applications this interaction is manifested in a *Graphical User Interface (GUI)*, which contains a number of *widgets* such as buttons, lists and fields. Often these GUI's are implemented by using a GUI Toolkit library.

When implementing computer systems using an object-oriented language another common component is the *Problem Domain Component* which models the part of the world that the system supports (Madsen et al., 1993).

This report investigates a number of architectures for interactive systems. The study is based on literature studies and experience from a recent research project (Christensen et al., 1998). Realising from this experience that current pattern literature on interactive systems (e.g. Buschmann et al. (1996) and Coldewey (1998)) were problematic in that it

- did not cover all used and usable architectures for interactive systems
- described architectural pattern verbosely, ambiguously, and incompletely
- did not relate and compare identified architectures well

Facing this architectural patterns for interactive systems have been identified, implemented, reframed, and to some extent formalised.

Concretely, this report will:

- present the four architectural patterns MVC, PAC, Application Facade, and Application Adaptor in a lightweight form.
- show a concrete usage of the four patterns in the construction of a small interactive application.
- show an application of the two pattern formalisms LePUS and Precise Visual Specification towards the MVC pattern, and from this discuss their applicability towards describing patterns.
- provide a very preliminary discussion on the relationship, differences, and similarities between each of the four architectural patterns.

Pattern descriptions

Experiencing that the pattern description format in Buschmann et al. (1996) was too long and that the format in Shaw et al. too short, we have experienced with the length and form of architectural pattern descriptions. The format we present here may be viewed as a condensed version of the format in Gamma et al. (1995) or as a variation on Brown et al.'s "deductive mini-pattern" template.

<p>Name</p> <p>What should this pattern commonly be known as?</p> <p>Problem</p> <p>Which architectural problem lead us to look for a pattern solution? And in which context?</p> <p>Solution</p> <p>How can this problem be effectively solved?</p> <p>Consequences</p> <p>Which benefits and liabilities come with applying the suggested solution?</p> <p>Credits</p> <p>No pattern is invented...</p>
--

Figure 1. Lightweight architectural pattern template

The solution section applies UML version 1.3 (Booch et al., 1998) and extensively use the generic grouping mechanism *packages*. Figure 2 below introduces most of the notation used in this section. The elements of a Concrete Domain package inherits from the elements of a Generic Domain package. The Concrete Domain Package depends on a Database Components package and associates a number of User Interface packages. Note that the diagram leaves the *contents* of the packets unspecified; contents may e.g. be classes or other packages.

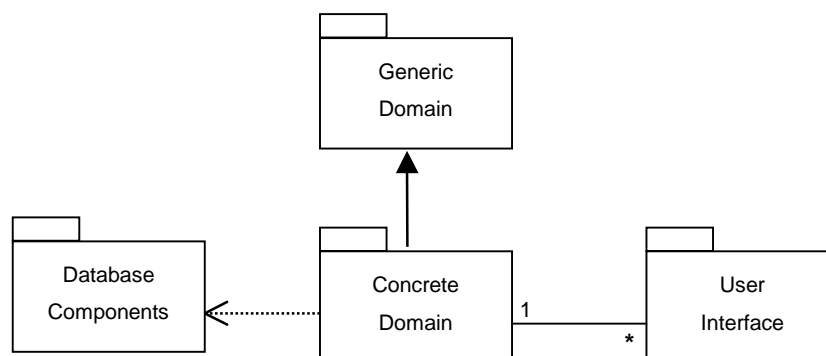


Figure 2. Sample package diagram

Application Facade

Problem

How does one connect the two major parts of an interactive application containing the problem domain related classes and the graphical user interface(GUI), in such a way that the GUI lies on the outside of the system and is invisible to the problem domain related classes?

Solution

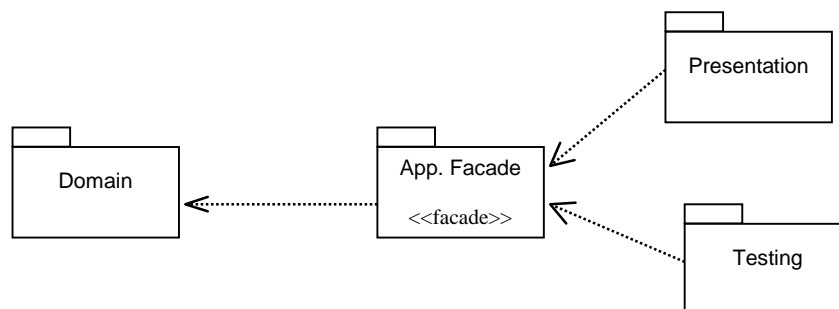
Divide the application into the components *Presentation*, *Testing*, *Application Facade*, and *Domain*. The Presentation component implements the user interface and handles both output to the user and input from the user.

The domain component contains the domain-related classes and functions, and possibly other functionality. It is implemented independently of the Presentation component.

To interface the two components an *Application Facade* component is implemented. This component has an interface to the Domain component, and the Presentation component is implemented so that it collaborates with the Domain component only via this interface. The Application Facade is thus dependent on the Domain component but independent of the Presentation component.

Optionally also implement a *Testing* component that tests the Domain component via the Application Facade component.

High level structure



Consequences

Benefits:

- The separation into Presentation and Application Facade support change in technology, as the Presentation component can be substituted by a new Presentation component when the user-interface of UI Toolkit changes
- The architecture facilitates testing without using the GUI
- Development of the parts that require domain knowledge and the parts that require UI Toolkit knowledge are separated

Liabilities:

- Multiple view consistency is not provided for. The Observer pattern can provide this, by having the Application Facade as subject and several Presentation components as Observers.

Credits

The Application Facade is described in a non-pattern format in (Fowler, 1997).

Model-View-Controller (MVC)

Problem

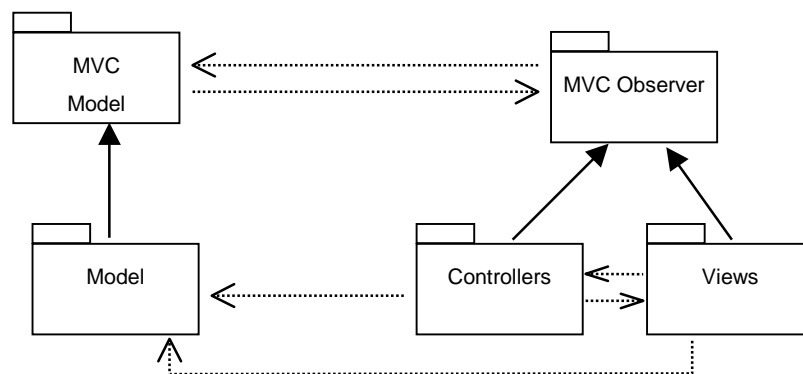
How does one connect the domain specific part with a user interface in such a way that multiple view on the same model is possible and in such a way that handling of user input is separated from the process of displaying output?

Solution

Divide the interactive application into three components: *Model*, *View* and *Controller*. The Model contains the core functionality and data and implements the MVC model interface. This interface allows interested parties to subscribe to “change-messages” that are to be sent when the Model’s state changes. The model does not depend on the concrete Views or Controllers, but rather collaborate with these through an abstract MVC Observer interface.

The Views contain the user interface that displays the system’s state to the user, and the Controller handles the part that receives input from the user. The Views and Controllers depend on the concrete Model, as they must update themselves when receiving the update message, and as the Controllers must be able to execute the functions found in the Model.

High level structure



Consequences

Benefits:

- Several views of the same model are very easily implemented
- View and controllers can easily be exchanged – even at runtime

Liabilities:

- The update propagation mechanism potentially leads to a large overhead
- Changes to the model easily propagate and require changes to the rest of the system

Variants

Often it is hard or ineffective to implement the strict separation of handling of input and output. Therefore the View and Controller classes can in some cases be joined, which results in the Document-View variant.

Credits

The article (Krasner and Pope, 1988) is the classic description of MVC in Smalltalk. (Buschmann et al., 1997) describes MVC in a pattern format.

Presentation-Abstraction-Control (PAC)

Problem

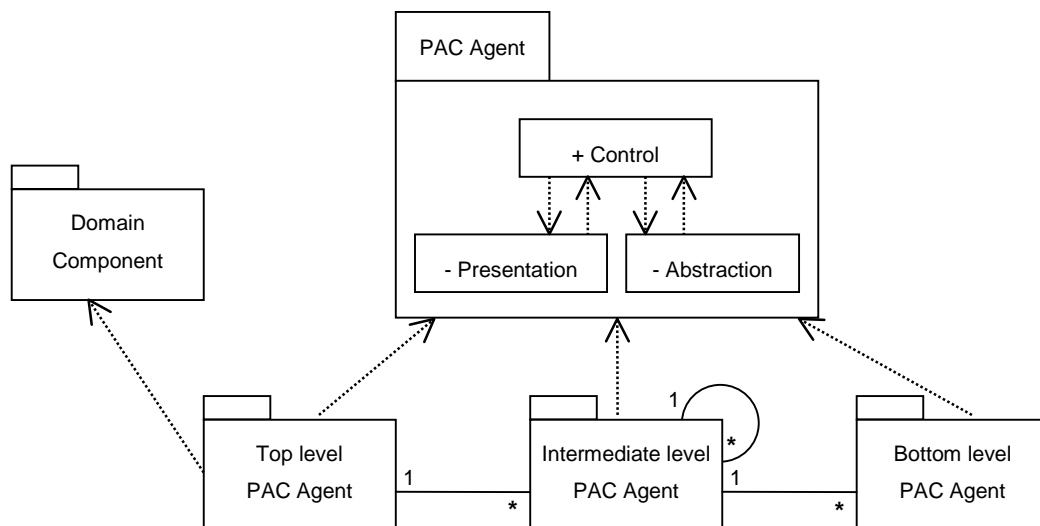
An interactive system may often benefit from a structuring as a set of cooperating components. How does one structure each component so that individual component cooperate effectively and how does one structure the cooperation of all components?

Solution

Divide the application into a hierarchy of *agents*. Horizontally, each agent provides a user interface (Presentation), a model (Abstraction), and a mediator between these parts and between an agent and other agents (Control). Vertically, the agents are structured in a hierarchy consisting of a top-level agent, intermediate level agents, and bottom level agents.

The *top level* PAC agent provides an interface to a global domain component through its control and controls the hierarchy of PAC agents. *Bottom level* PAC agents implement atomic units of semantic concepts from the use domain. These levels are coordinated or composed or both by *intermediate level* PAC agents.

High level structure



Consequences

Benefits:

- The main benefit of this pattern is that it separates different semantic entities of the use domain effectively. In this way semantic entities may evolve individually

- Since the Presentation and Abstraction part of a PAC agent is hidden from other PAC agents these parts may be exchanged without major changes to other agent.

Liabilities:

- The system complexity increases since semantic entities are implemented entirely by PAC agents. Also the hierarchic structure of PAC agents may be complicated to coordinate and control
- Since communication in between the PAC agents may go through several other agents efficiency of the interactive system may be affected by the use of this pattern.

Credits

Coutaz (1987) describes the PAC architecture in its original format and a C implementation. Buschmann et al. (1996) gives PAC a pattern format.

Application Adaptor

Problem

How does one structure an application so that the user-interface components and problem domain component can be developed independently and so that changes to the UI require minimal change to the rest of the system?

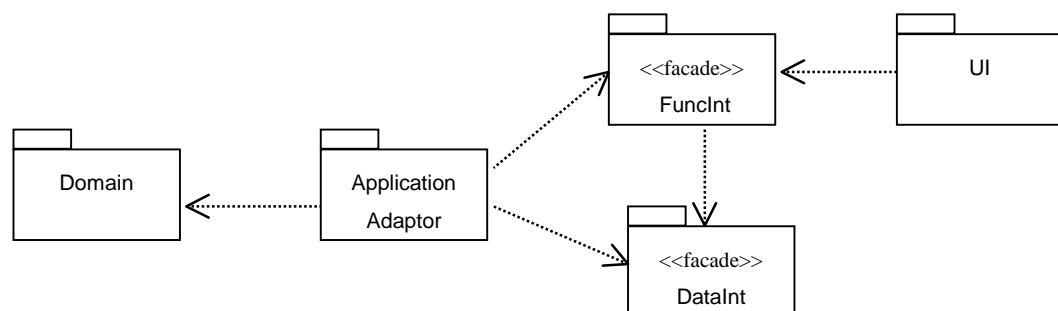
Solution

Implement the Domain component as a separate and independent component.

Divide the implementation of the user interface into several parts: *UI*, *FuncInt* and *DataInt*. The UI contains the declarations and instantiations of the concrete controls/widgets found in the user interface. The *DataInts* contain a data interface for each screen, while the *FuncInts* contain a functional interface for each screen.

Finally create the *Adaptors* which subscribe to events in the functional interface and upon invocation of these events execute the appropriate functions in the Domain component and transfer data from/to the Domain component to/from the Info components.

High level structure



Consequences

Benefits:

- The UI classes are independent of the Domain, *FuncInt*, and *DataInt* components, and can therefore be constructed and compiled separately
- The Domain classes are independent of the UI, but has the *Controllers* as direct dependants. They can therefore to some extent be constructed and compiled separately

Liabilities:

- The interface to the UI represented by the *FuncInt* and *DataInt* takes some time to develop and maintain
- The architecture provides no mechanism for multiple view consistency per se

Credits

The Application Adaptor pattern has been used in several projects at Department of Computer Science, University of Aarhus. Most recently it has been used in the construction of a large system in the Dragon project (Christensen et al., 1998). Lennert Sloth suggested its use in that project.

A sample application

In order to investigate the patterns a simple interactive system has been developed using the Mjølner System (Knudsen et al., 1994) implemented in Beta (Madsen et al., 1993). The four patterns MVC, PAC, Application Facade and Application Adaptor have been used.

The system implements a simple financial history for keeping track of expenses. The application has three views as shown in Figure 3: A view for adding and deleting data, a view showing the financial information as a pie chart, and a view showing the financial information as a bar chart.

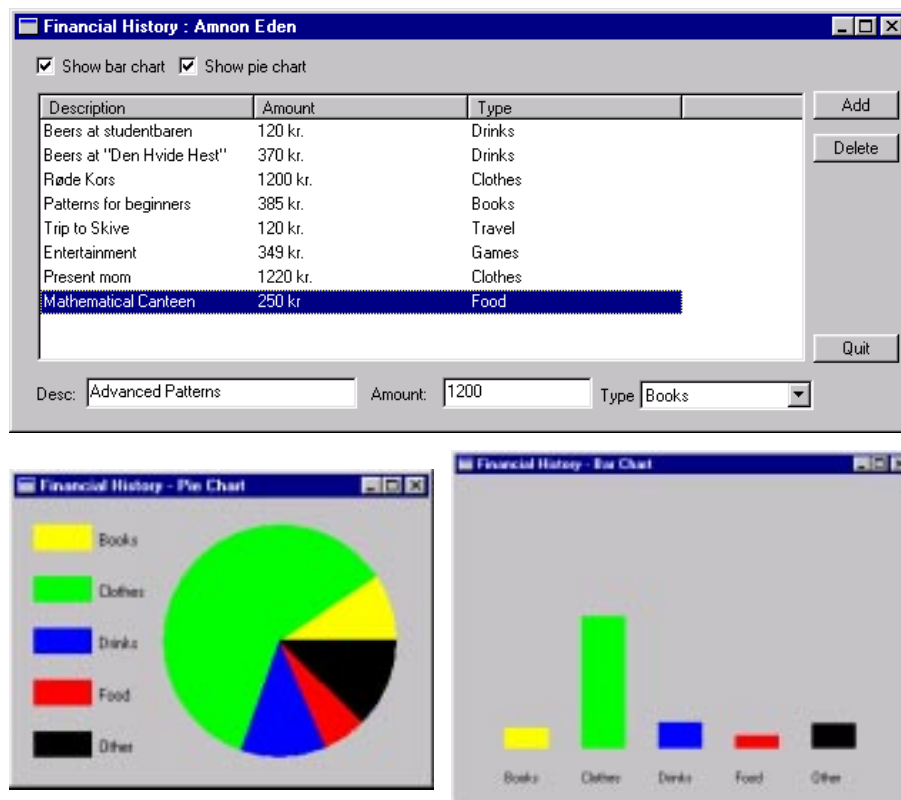


Figure 3. The Financial History application

Shared implementation

To ease the implementation the four applications have been implemented such that they all share the same problem domain model code and the same user interface code.

Problem domain model

The problem domain model consists of four simple classes: Person, FinancialHistory, FinancialElement and Quantity.

Person models an individual and the class is associated to one or more financial histories.

A financial history models a specific history of expenses, and it contains a description and is associated to one or more financial elements.

A financial element models one expense, and contains a description, an amount and a type.

The model is shown in UML class diagram notation below.

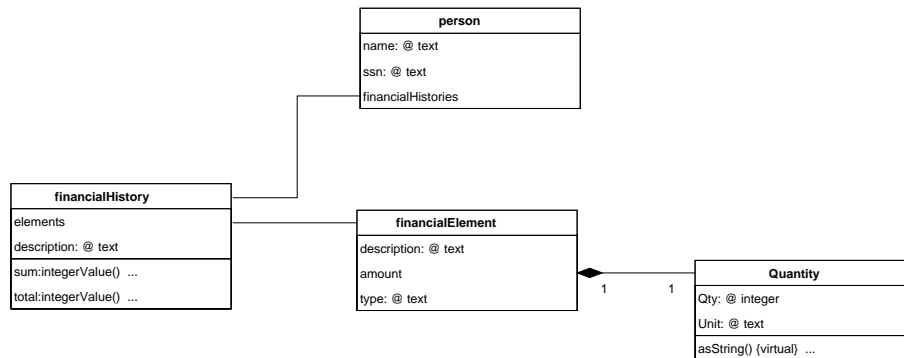


Figure 4. The problem domain model

User interface

The user interface consists of the three screens shown in the beginning of this section.

All three screens have been constructed using the Mjølner GUI Builder. The generated code is shown in Appendix 1.

Application Facade

The Application Facade architectural pattern has four main components: Domain, Application Facade, Presentation and Testing. The implementation of the pattern goes in four major steps, and uses the library code shown in Figure 5. It should be noted that these steps cannot and should not be taken sequentially but rather be the basis of an iterative development.

```

facade:
  (#
    subjectType:< object;
    theSubject: ^subjectType;
    load:< (# do INNER #);
    save:< object
  #);
presentation:
  (#
    facadeType:< facade;
    theFacade: ^facadeType;
    init:< (# enter theFacade[] do INNER ; load #);
    load:< object
  #)

```

Figure 5. The library code for Application Facade

1. Implement the problem domain model

For the problem model we will use the shared implementation described on page 16.

2. Implement the Application Facade

The Application Facade acts as a facade to the problem domain model. It should therefore be possible to call all functions of interest from the Application Facade and all interesting state should also be visible. Figure 6 shows the Application Facade for the input window (the topmost windows of the windows shown on page 16). Here the functions ‘scan’, ‘delete’ and ‘add’ are available as well as the state ‘name’, ‘total’ and ‘sum’.

```

financialFacade: facade
  (#
    name: ^text;
    subjectType:: financialHistory;
    total: integerValue (# do ... #);
    sum: integerValue (# type: ^text enter type[] do ... #);
    scan:
      (#
        currentAmount: ^text;
        currentType: ^text;
        currentDescription: ^text;
        currentObject: ^object
        do theSubject.elements.scan (# ... #)
      #);
    delete: (# theObject: ^object enter theObject[] do ... #);
    add:
      (#
        description: ^text;
        amount: ^text;
        type: ^text;
        elm: ^financialElement
        enter (description[],amount[],type[])
        do ...
      #)
  #)

```

Figure 6. The Application Facade for the input window

Furthermore all Application Facades – as shown in Figure 5 also contain the methods ‘init’, ‘load’ and ‘save’. ‘Init’ performs all relevant initialisation, ‘load’ fetches state from the problem domain model and makes it accessible from the Application Facade and ‘save’ saves the internal state in the Facade in the problem domain model.

2. Implement the Presentation component, either as a wrapper around existing user interface code or from scratch.

As described above we already have most of the user interface generated from the GUI builder. We will use this code to create the presentation components.

The presentation for the input window (see Figure 7) is implemented as a subclass of the presentation library class. This subclass holds an instance of the input window as defined by the code generated by the GUI builder. It furthermore implements the virtual functions ‘load’ and ‘init’. ‘Load’ is bound to call ‘load’ in the user interface and ‘init’ is bound to open the window.

Furthermore a number of events are subscribed to in the specialisation of the window. In these events the actual functionality is called. Take as an example the ‘add button’ event: First an expense is added to the Facade, then the state of the Facade is saved and finally the Presentation is loaded with the new state from the Facade.

Figure 7. The Presentation for the input window

```

theInputPresentation: @presentation
(
  #
  ui: @inputView
  (
    #
    addAction: addPushButton.mouseUpAction
    (
      #
      do
        (descEditText.contents, amountEditText.contents,
         typeOptionButton.currentLabel)->theFacade.add;
        theFacade.save;
        load
      #);
    deleteAction: deletePushButton.mouseUpAction ...;
    quitAction: quitPushButton.mouseUpAction ...;
    barChartAction: showBarChartCheckBox.stateChangedAction ...;
    pieChartAction: showPieChartCheckBox.stateChangedAction ...;
    open:: ...;
    load:
    (
      #
      do
        financesListView.clear;
        theFacade.scan
        (
          #
          do
            (currentDescription[], currentAmount[], currentType[],
             currentObject[])->financesListView.new
          #);
        theBarChartPresentation.load;
        thePieChartPresentation.load
      #)
    #);
    facadeType:: financialFacade;
    load:: ...;
    init:: ( # do ui.open #)
  #);
)

```

3. Optionally implement the Testing component

For effective testing implement a program that systematically calls all methods in the Facade and after each call test whether the state of the Facade is as expected.

The testing component has not been implemented in our sample application.

Final structure

The resulting structure of the main program structured according to the Application Facade architectural pattern is shown in a UML Class diagram in Appendix 2.

Model-View-Controller (MVC)

The MVC pattern may be implemented using a framework consisting of an abstract model, an abstract observer, and concrete views and controllers as sketched in the library code of

Figure 8.

```

MVCmodel:
  (#
    private: @ (# dependents: @list (# element:: MVCobserver #) #);
    addDependent: (# anObserver: ^MVCobserver enter anObserver[] do ... #);
    removeDependent: (# anObserver: ^MVCobserver enter anObserver[] do ... #);
    changed: (# do private.dependents.scan (# do current.update #) #)
  #);
MVCobserver:
  (#
    update:< (# do INNER #);
    myModel: ^modelType;
    modelType:< MVCmodel
  #);
MVCview: MVCobserver
  (#
    myController: ^MVCcontroller;
    controllerType:< MVCcontroller;
    init:<
      (# theModel: ^modelType
        enter theModel[]
        do
          (if theModel[] <> none then
            theModel[]->myModel[];
            THIS(MVCview)[]->theModel.addDependent;
            &controllerType[]->myController[];
            (theModel[],THIS(MVCview)[]->myController.init
            if);
            INNER
          #)
      #);
MVCcontroller: MVCobserver
  (#
    myView: ^viewType;
    viewType:< MVCview;
    init:< (# enter (myModel[],myView[]) do INNER #)
  #)

```

Figure 8. The library code for MVC

1. Implement the MVC Model

The MVC model contains the data and functionality. Using our library code we create a subclass of the MVC Model class, and inside this class add a pointer to an *entry point* in the problem domain model. By an entry point we denote a central object of interest from which other associated objects can be reached. In our application we choose the ‘financial history’ class as the entry point as our main window displays *one* financial history at the time, and all other information displayed in the window can be reached from it. This results in the simple MVC Model shown in Figure 9.

```

financialModel: MVCmodel
  (#
    theFinancialHistory: ^financialHistory;
    getHistory:
      (# personName: ^text
        enter personName[]
        do
          personName[]->db.getFinancialHistory->theFinancialHistory[];
          changed
        #)
    #);

```

Figure 9. The MVC Model for Financial History

2. For each window implement the MVC View

The next step involves creating a MVC View for each window/view in the application. As our application has three windows we must implement three MVC Views.

These are constructed as specialisations of the MVC View library class (Figure 8). In the specialisation the empty virtual methods `init` and `update` should be defined, and a MVC Controller should be supplied (see step 3.).

Take again as example the input window, for which the implemented MVC View is shown in Figure 10. This concrete View implements the `init` method to simply open the window which the View is nested inside. Furthermore the 'update' method is implemented to clear the listbox on the screen and then refill it with the current expenses found in the current financial history. Finally the concrete controller is bound as explained in step 3 below.

```

theInputView: @inputView
  (#
    theController: MVCcontroller (# ... #);
    theView: @MVCview
      (#
        modelType:: financialModel;
        controllerType:: theController;
        update::
          (#
            do
              financesListView.clear;
              myModel.theFinancialHistory.elements.scan
                (#
                  do
                    (current.description[],current.amount.asString,
                     current.type[],current[])->financesListView.new
                  #)
                #);
            #);
        init:: (# do open #)
      #)
  #);

```

Figure 10. The MVC View for the input window

3. For each MVC View which receives input, implement the corresponding MVC Controller

MVC Controllers handle the input from the user, and thus it is only windows which contain interactive elements that need controllers. In our application this leaves us only the input window.

The concrete Controllers are required to implement the `init` and `update` methods from the interface. The `'init'` method initialises the Controller and the `'update'` method is used when the reactions to the input should change depending on the current state of the Model (e.g. situations such as only allowing deleting when there is something to delete).

In our case the concrete input Controller has been implemented as in Figure 11. The implementation is quite simple. It defines a number of GUI event actions, and the `init` method then adds these actions to the GUI event-handler. The `update` method is not refined, as this window does not have any functionality that depends on the state of the Model.

```

theController: MVCcontroller
  (#
    addAction: addPushButton.mouseUpAction ...;
    deleteAction: deletePushButton.mouseUpAction ...;
    quitAction: quitPushButton.mouseUpAction ...;
    barChartAction: showBarChartCheckBox.stateChangedAction ...;
    pieChartAction: showPieChartCheckBox.stateChangedAction ...;
    enableAddAction: descEditText.basicEventAction ...;
    enableDeleteAction: financesListView.basicEventAction ...;
    modelType:: financialModel;
    init::
      (#
        do
          &addAction[]->addPushButton.appendAction;
          &deleteAction[]->deletePushButton.appendAction;
          &quitAction[]->quitPushButton.appendAction;
          &barChartAction[]->showBarChartCheckBox.appendAction;
          &pieChartAction[]->showPieChartCheckBox.appendAction;
          &enableAddAction[]->descEditText.appendAction;
          &enableDeleteAction[]->financesListView.appendAction;
          addPushButton.disable;
          deletePushButton.disable
        #)
      #);
  #);

```

Figure 11. The MVC Controller for the input window

Final structure

The resulting structure of the main program structured according to the MVC architectural pattern is shown in a UML Class diagram in Appendix 3.

Presentation-Abstraction-Control (PAC)

The pattern has as the only components the PAC agent which has three parts: Presentation, Abstraction and Control. The implementing follows in 5 main steps. The implementation uses the PAC library code as shown Figure 12.

```

PACagent:
  (#
    topLevel: ^PACagent;
    presentation:< (# init:< object #);
    abstraction:< (# init:< object #);
    control:< (# init:< object #);
    thePresentation: @presentation;
    theAbstraction: @abstraction;
    theControl: @control;
    upperType:< PACagent;
    upper: ^upperType;
    lowerType:< PACagent;
    lower: @list (# element:: lowerType #);
    init:<
      (#
        enter upper[]
        do
          lower.init;
          theAbstraction.init;
          theControl.init;
          thePresentation.init;
          INNER
        #)
      #)
  #)

```

Figure 12. The library code for PAC

1. Implement the problem domain model

We will use the shared implementation of the problem domain model. It will in step 3. be made part of the top-level agent.

2. Define a strategy for organizing the PAC hierarchy

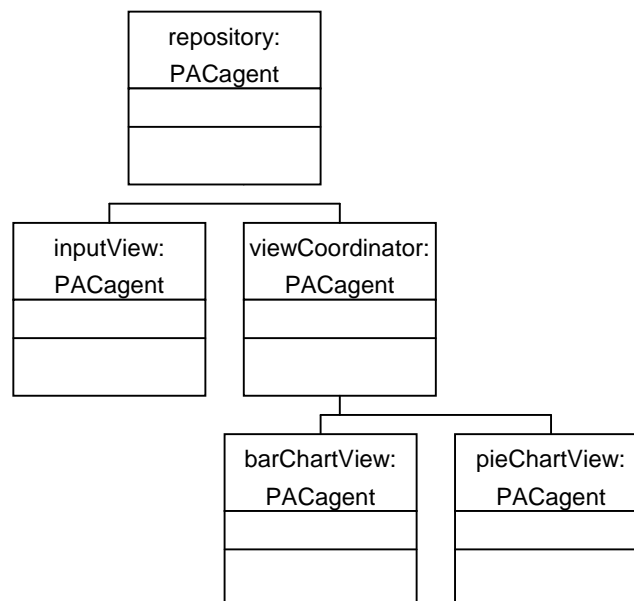


Figure 13. The PAC hierachy for our application

3. Implement the top-level PAC agent

The top-level agent represents the functional core of the system. It should therefore contain an interface to the problem domain model that was created in step 1. Furthermore the top-level agent initialises the agents below it.

In Figure 14 our top-level agent is shown. The seven methods at the bottom represent the interface to the problem domain model, and the initialisation method creates the input agent and the view coordinator agent.

```

repository: PACAgent
  (#
    theInput: ^input;
    theViewCoordinator: ^viewCoordinator;
    init::
      (#
        do
          &input[]->theInput[]->lower.append;
          &viewCoordinator[]->theViewCoordinator[]->lower.append;
          lower.scan
            (# do THIS(repository)[]->current.init #)
        #);
    abstraction::
      (#
        fh: ^financialHistory;
        db: @database;
        init:: (# do db.getFinancialHistory->fh[] #)
      #);
    terminate: ...;
    addElement:
      (# elm: ^financialElement enter elm[] do ...; ... #);
    removeElement:
      (# elm: ^financialElement enter elm[] do ...; ... #);
    sum: integerValue ...;
    total: integerValue ...;
    scanDistribution: ...;
    scanElements: ...;
    showBarChart: ...
  #);

```

Figure 14. The top-level PAC agent

4. Implement bottom-level PAC agents

The bottom-level agents represent self-contained semantic concepts. Bottom-level agents have no lower agents by definition.

In our application we choose that each of the three windows each represent a self-contained concept, so for each of these an agent is implemented.

Figure 15 shows the agent containing the “bar chart” window and Figure 16 shows the agent containing the “input” window. Both agents implement the presentation part, which holds the actual window contents. The implementation is simple, e.g. the “bar chart” presentation creates an instance of the barChart window (described on page 17), and the simply provides a method ‘setData’ which asks the top-level agent for the current state and then sets the window accordingly.

The input agent furthermore implements the control part to process the input that is received from the user. For each event in the user interface a method is implemented, which represents the functionality to be executed upon the event.

```

barChart: viewAgent
  (#
    presentation::
      (#
        theBarChartView: @barChartView
        (# setData: (# ... #) #);
        init::
          (#
            do
              theBarChartView.open;
              theBarChartView.setData;
              theBarChartView.hide
            #);
        show:: (# do theBarChartView.show #);
        hide:: (# do theBarChartView.hide #)
      #);
    control:: ...
  #);

```

Figure 15. The barChart PAC agent

```

input: PACAgent
  (#
    upperType:: repository;
    presentation:: ...;
    control::
      (#
        scanElements: (# current: ^financialElement do ... #);
        addElement: upper.addElement (# #);
        removeElement: upper.removeElement (# #);
        showBarChart: (# value: @boolean enter value do ... #)
      #)
  #);

```

Figure 16. The input window PAC agent

5. Implement the intermediate-level PAC agents

The intermediate-level agents are agents which perform coordinating activities. In our application we need an agent to provide state consistency between our three views. This is done by the “view coordinator” agent shown in Figure 17.

The implementation is simple: When it receives the ‘changed’ message it simply forwards this message to its lower views.

```

viewCoordinator: PACAgent
  (#
    init:: ...;
    upperType:: repository;
    lowerType:: viewAgent;
    showView: (# viewType: ##viewAgent enter viewType## do ... #);
    hideView: (# viewType: ##viewAgent enter viewType## do ... #);
    sum: integerValue ...;
    total: integerValue ...;
    changed: (# do lower.scan (# do current.changed #) #)
  #);

```

Figure 17. The view coordinator PAC agent

Final structure

The resulting structure of the main program structured according to the PAC architectural pattern is shown in a UML Class diagram in Appendix 4

Application Adaptor

As described in the pattern section, the Application Adaptor has five main components: UI, FuncInt, DataInt, Adaptor and Domain. The four steps in implementing this pattern are:

1. Implement the problem domain model and user interface

For both the problem model and the user interface we will use the shared implementation described on page 16.

2. Implement the dataInt

The DataInt provides a data interface, which reflects the data that is displayed in the user interface. In Figure 18 the data interface for the input window is shown. It consists of a simple attribute for each data element in the user interface. As seen e.g. the “Show pie chart” checkmark is represented by a boolean and the listbox with the expenses is represented by a list of items, which each has a text for the description, an integer for the amount and a text for the type.

```

financialInputInfo:
  (#
    showBarChart: @boolean;
    showPieChart: @boolean;
    descText: @text;
    amountText: @integer;
    typeText: @text;
    itemsList: @list
    (#
      element:: itemInfo;
      remove:
        (# theElm: ^element enter theElm[] do theElm[]->at->delete #)
    #);
    itemsCurrentSelection: ^itemInfo;
    itemInfo: (# desc: @text; amount: @integer; type: @text #)
  #)

```

Figure 18. The DataInt for the input window

3. Implement the funcInt

The FuncInt provides a functional interface, consisting of a number of empty virtual functions – one function for each event of interest in the user interface. Figure 19 shows the functional interface for the input window. E.g. the event that the “Add” button was clicked is represented by a virtual method named ‘onAdd’.

Furthermore the FuncInt should create an instance of the data interface (here the instance is called theInfo), and it should implement two methods: setView and getView.

Both the setView and the getView methods map data between the data interface and the user interface: setView set the contents of the user interface to the contents of the data interface and getView does the opposite.

```
financialInputView:
  (#
    theInfo: @financialInputInfo;
    setView:< ...;
    getView:< ...;
    private: @ (# theView: @inputView #);
    init:< ...;
    open:< (# do init; private.theView.open; INNER #);
    close:< (# do INNER ; private.theView.close #);
    onAdd:< (# do INNER #);
    onDelete:< (# do INNER #);
    onQuit:< (# do INNER #);
    onShowBarChart:< (# do INNER #);
    onShowPieChart:< (# do INNER #)
  #)
```

Figure 19. The FuncInt for the input window

4. Implement the Adaptor

The Adaptor is somewhat more complex as it connects the other components. Generally the implementation of the Adaptor should:

- Add presentModel/updateModel methods, which map data from the problem domain model to the data interface (presentModel) and from the data interface to the problem domain model (updateModel).
- Override/refine those virtual methods in the FuncInt, which represent events of interest
- Implement the general application functionality inside the appropriate event methods
- Implement multiple view consistency if needed

The Adaptor for the input window is shown in Figure 20. In this implementation the Adaptor for the input view refines all five virtual event methods from the functional interface.

Take as example the “onAdd” event, that implements the functionality required when the user presses the ‘Add’ button to add a new expense. The implementation first call getView to have the state of the UI copied into the data interface. It then reads the state of the data interface and creates a new model object and appends this object to the problem domain model. After this it calls update which again calls presentModel to have the state of the domain model copied into the data interface, then calls setView to copy this state into the user interface and finally calls update on the two other views.

```

(#
  db: @database;
  ui: @guienv
  (#
    financialInput: @financialInputView
    (#
      update:
        (#
          do
            theHistory[]->presentModel;
            setView;
            financialBarChart.update;
            financialPieChart.update
          #);
        presentModel: ...;
        onAdd::
          (# newElm: ^financialElement
            do
              getView;
              &financialElement[]->newElm[];
              theInfo.descText->newElm.description;
              theInfo.amountText->newElm.amount.qty;
              theInfo.typeText->newElm.type;
              newElm[]->theHistory.elements.add;
              update
            #);
          onDelete:: ...;
          onQuit:: ...;
          onShowBarChart:: ...;
          onShowPieChart:: ...;
          open:: ...
        #);
    financialBarChart: @financialBarChartView ...;
    financialPieChart: @financialPieChartView ...;
    theHistory: ^financialHistory;
    onStartApplication:: ...
  #)
do ui
#)

```

Figure 20. The Application Adaptor for the input window

Final structure

The resulting structure of the main program structured according to the Application Adaptor architectural pattern is shown in a UML Class diagram in Appendix 5

Formal Specification of Architectural Patterns

Current specifications (e.g. Shaw (1996) and Buschmann et al. (1996)) of architectural patterns are problematic. Generally speaking, the descriptions are fuzzy or ambiguous, it is hard to see how to get from specification to implementation, and equally hard to decide whether a given implementation implements a given pattern.

Given that architectural patterns describe solutions to recurrent architectural problems two ways of using contemporary means of formalisation are obvious then, namely using

- an Architecture Description Language (ADL) such as Wright (Allen, 1997) to describe the architectural solution of the pattern
- a specification technique for design patterns noting the similarities between architectural and design patterns.

In this context we have chosen the second approach for two reasons: First, current descriptions of architectural patterns are mostly reference architectures (Bass et al., 1998) in the sense they map a division of functionality together with data flow between pieces onto software components and the data flows between the components. Architectures describe structures of concrete systems and may thus be instantiations of reference architectures. Current ADLs do not focus on the description of reference architectures. Second, architectural patterns are specified as patterns and share many motifs (Eden et al., 1998) with design patterns.

We have applied two design pattern specification languages, namely LePUS ((Eden et al., 1998) and (Eden et al., 1999) and the precise visual specification language of Lauder et al. (1998).

Language for Pattern Uniform Specification (LePUS)

LePUS builds on the observations of regularities – *motifs* – in design patterns. Such abstractions include *functions*, *methods*, *relations* between functions and

$$\exists(x_1, \dots, x_n) : \wedge \mathfrak{R}_i(y_i, \dots, y_{i_n}); x_i \in \text{functions, classes}; y_i \in (x_1, \dots, x_n), \mathfrak{R}_i \in \text{relations}$$

methods, and *uniform sets*. Design patterns are specified as a formula in higher order monadic logic that specifies the so called *lattice*, i.e. structure and collaboration, of the design pattern. The LePUS formalism is described in detail in (Eden, 1999). Here we will primarily use the equivalent visual notation (Figure 21 below) of LePUS and explain the diagrams whenever necessary.

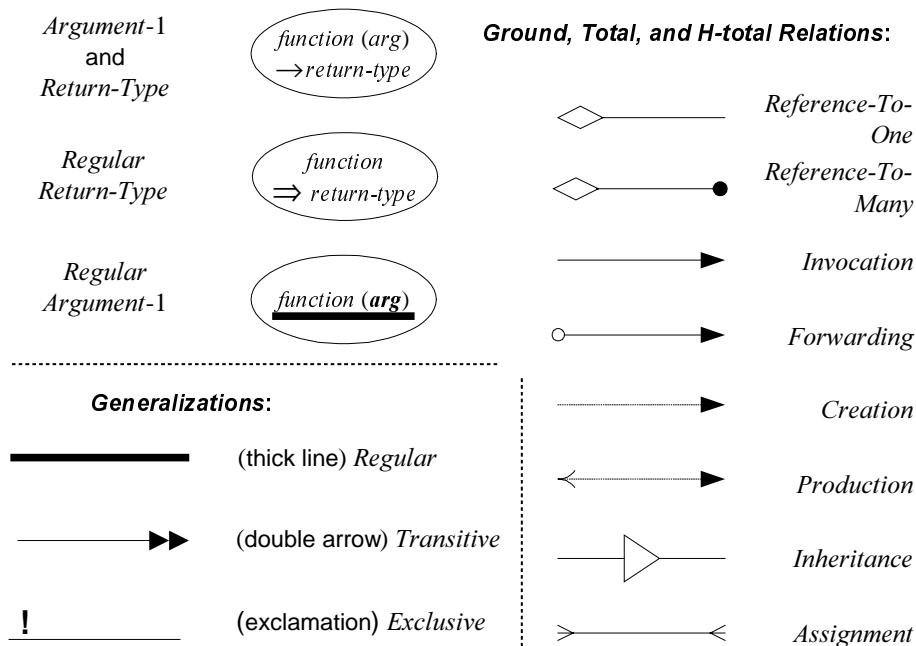


Figure 21. Visual notation of LePUS (Eden, 1999)

In order to use LePUS as a specification language for a certain type of pattern one must in principle determine the motifs of that type of patterns, i.e.

- Determine the ground entities of the specific patterns
- Determine the characteristic relations between these

Since we, at least for the patterns in Buschmann et al. (1996), have seen many of the same motifs as described by Eden et al. (1998) in design patterns (Gamma et al., 1995) we have chosen to use the LePUS formalism “as is”.

Precise Visual Specification (PVS)

PVS extends *constraint diagrams* as introduced by Kent (1997). Constraint Diagrams offers as an extension of the UML a visual notation for expressing invariant constraint on class models. The Constraint Diagrams use a variant of the Venn notation for specifying instances of classes. That a professor has many students and that these students have the professor as one of their professors may e.g. be specified as shown in Figure 22 below.



Figure 22. Professor and student constraint diagram

Realising that current pattern specifications such as Gamma et al. (1995) and Buschann et al. (1996) are based on specific implementations of patterns, Lauder et al. (1998) introduce a layered three model specification of design patterns (Shown as a UML meta-model in Figure 23.)

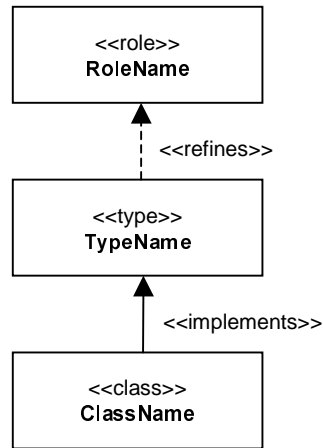


Figure 23. Three-model layering of PVS

Each layer specifies expresses a pattern at a certain level of abstractness, from concrete state and semantics (class layer) over application-domain-specific abstract state and semantics (type layer) to abstract state and semantics (role layer). Each layer is presented as UML class diagrams with an extra compartment added to the class thing showing (abstract) state. On each level Venn diagrams are used to represent arbitrary sets.

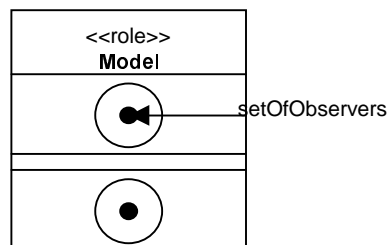


Figure 24. Role-model of a Model, having a set of observer, no abstract operations, and a set of abstract instances

Precise Specification of Model-View-Controller (MVC)

As an experiment MVC as presented in Buschmann et al. (1996) has been specified in both LePUS and PVS.

LePUS Specification

A LePUS visual specification of MVC is shown in Figure 25 below.

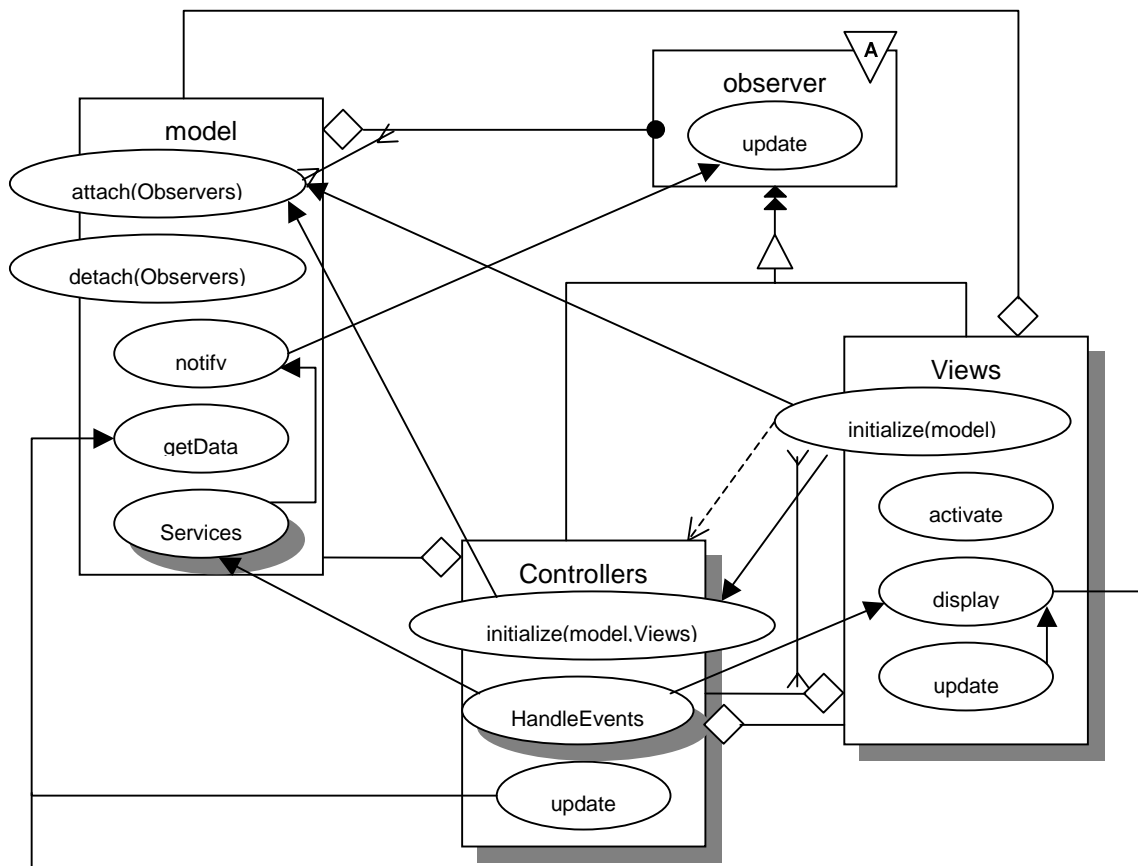


Figure 25. LePUS specification of MVC

Note, however, that an equivalent LePUS formula may be written down, although this is not done here; the visual notation seems more intuitive and easier to handle. From the diagram it may e.g. be deduced that

- There exists a *hierarchy* of observers with an *abstract* observer base class. The subclasses are divided into two different *sets of classes*: Controllers and Observers. The observer class defines an update function that its subclasses override.
- There exists a model class with four functions (attach, detach, notify, getData) and a *set* of functions Services. The model has references to a number of observers. These observer references are *assigned* by the attach method that is called from initialize on Views and Controllers.

- A Controller has a reference to a View and a Controller is referenced from a Controller. These references *commute* (not shown in the diagram), i.e. a Controller's View's Controller is the original controller and vice versa.
- A View *produces* and initializes a controller.
- Service requests to the model yields notifications which in turn yields update on Views and Controllers. This means that Controllers and Views after a call to a service reads data via `getData` from the model.

Although the above diagram looks succinct a few problems in the specification may be noted. The current notation is e.g. not capable of stating the dynamic property that model, Views, and Controllers are synchronized in state after a notify call. LePUS's focus on static properties is however also one of its strengths: The diagrams visualisation is restricted. However, certain dynamic properties might be specified by adding additional association types. Two examples of this is sequence as in the initialize function of the Views and selection in the `handleEvent` function of the controllers.

Assuming the specification of the Observer Pattern in (Eden et al. 1998, Figure 26 below) and the specification of MVC are adequate, the two specifications share many commonalities. In fact the only differences are that MVC does not specify an abstract model and that Observer does not elaborate on the Observers hierarchy. If, however, a description of MVC does specify the existence of an abstract model, as our presentation indeed does, then the resulting MVC pattern specification would, interestingly, be a refinement of the Observer Pattern. A presentation of MVC would indeed benefit from using this fact.

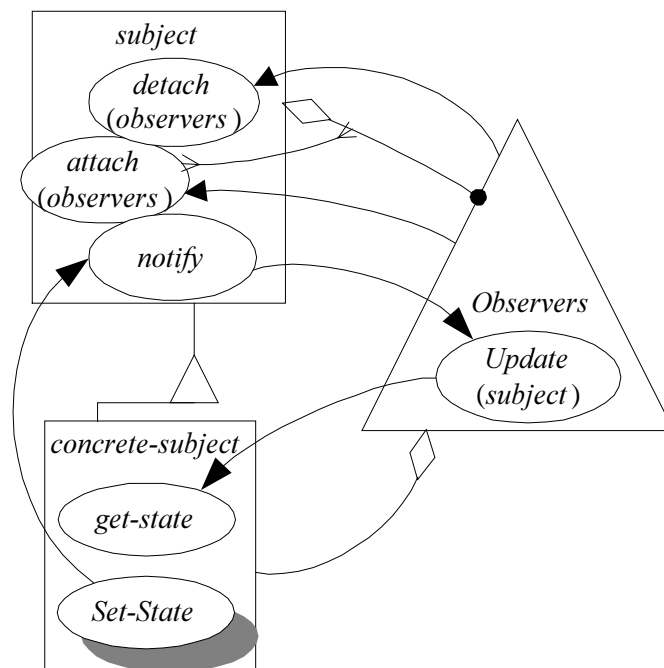


Figure 26. LePUS specification of the Observer Pattern

PVS Specification

Figure 27 below shows a tentative precise visual specification of an MVC role model. Note that in order to use this pattern the role model must be refined into a type model that must then be refined into a class model. This process is not shown here, Laueder et al. (1998) may be consulted for specifics⁷.

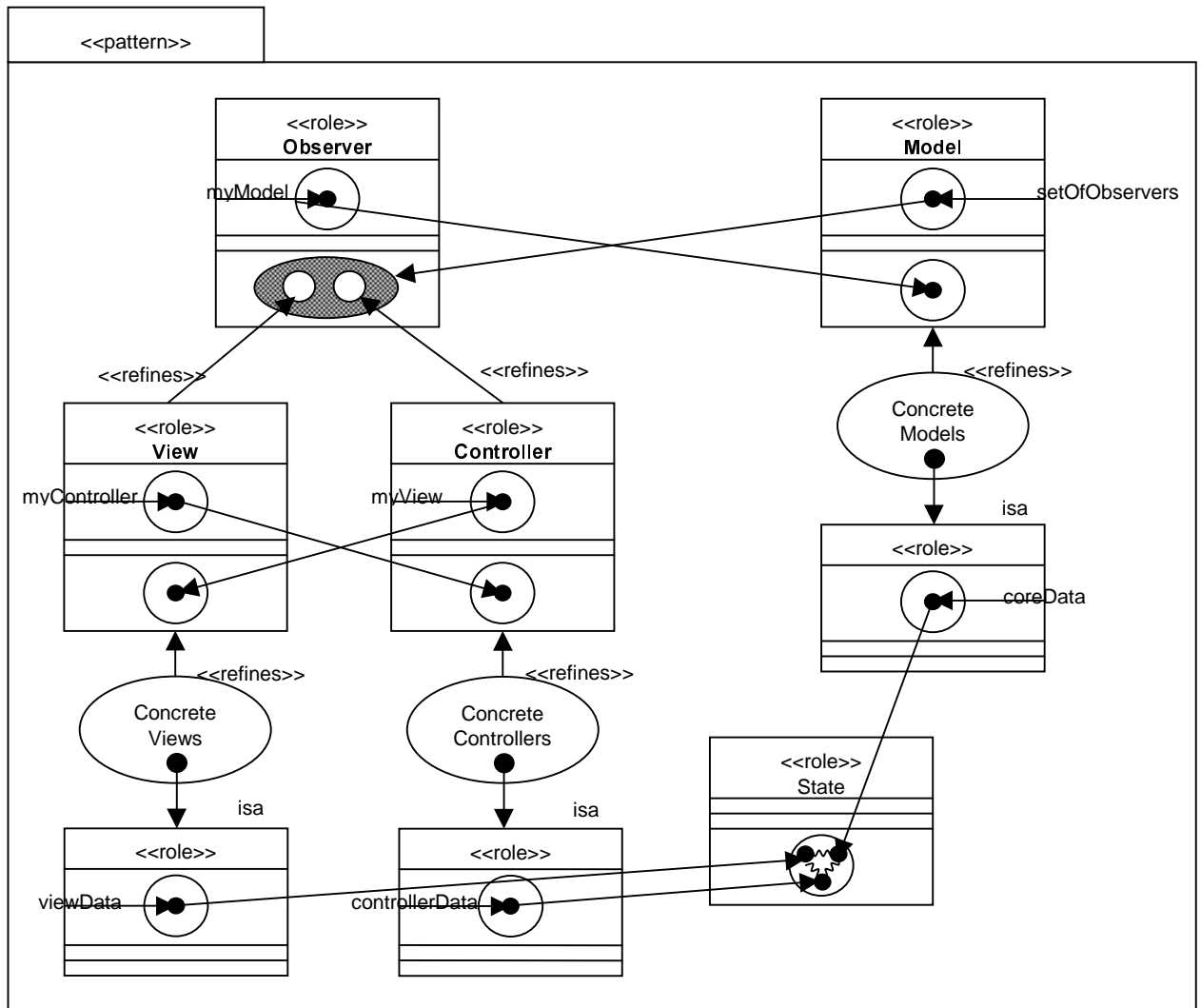


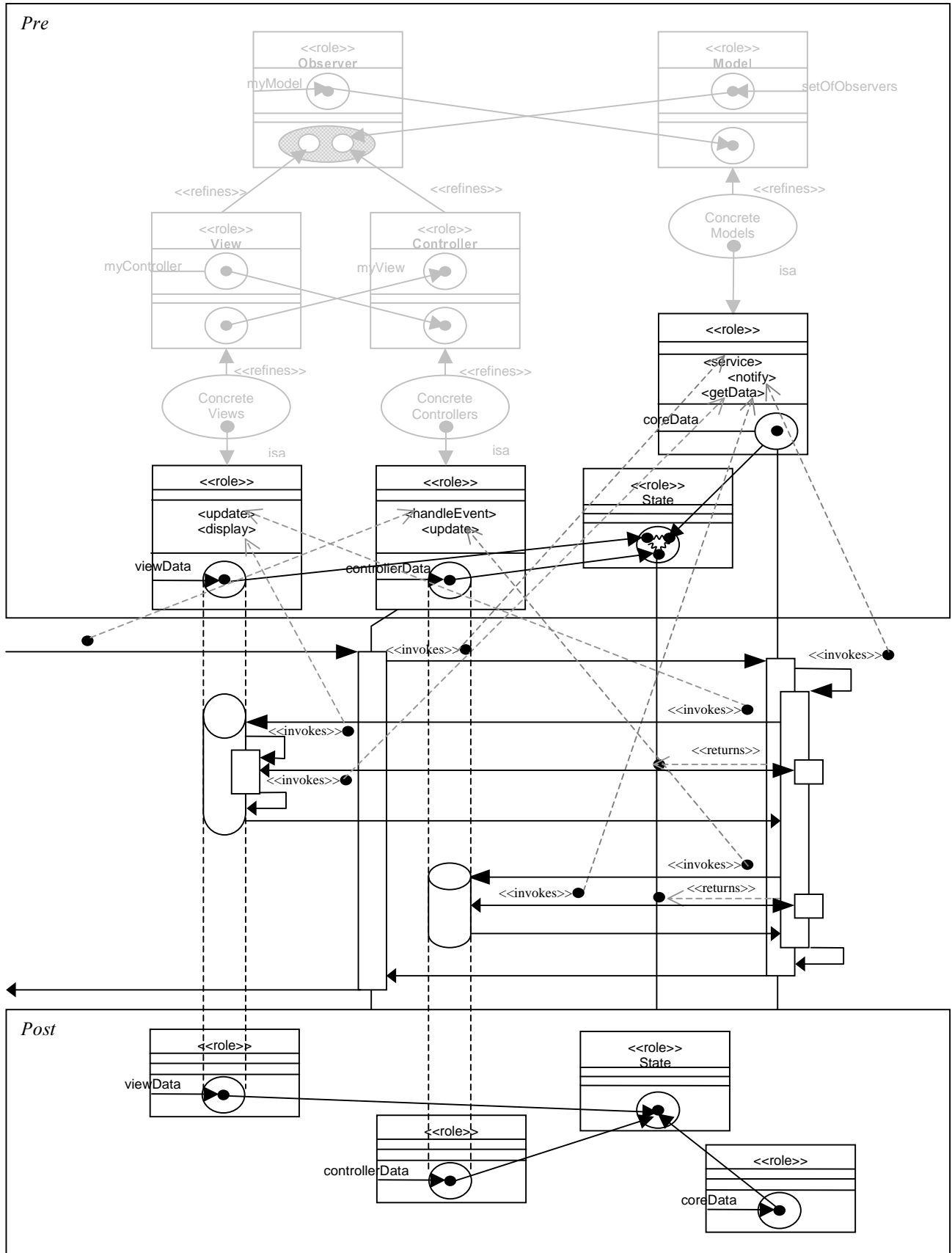
Figure 27. Precise Visual Specification of MVC role invariant

The specification provided by PVS on this level is purely about abstract data and constraints on that: No processing is specified. The pattern is shown as a package diagram stereotyped <<pattern>>. Comparing the specification to that made in LePUS one sees that it specifies at least the same *static* constraints albeit on a more abstract level: Observers has an association to a model, a model associates a set of Observers, Views and Controllers are a partitioning of Observers, and Views and Controllers associate each other symmetrically. Furthermore it is specified that concrete Controllers, Views, and Models

associate abstract instances of the same State role. These instances may or may not be the same at a given time.

Notice, that the notation is rather verbose even for the role invariant diagram. The elaboration of this diagram into a *role-model sequence diagram* (next page) makes the verbosity even more evident. These sequence diagrams generalises the notion of operations as abstracts sets with constraint into constraint on behavioural semantics. Thus the diagram on the next page constrains the pattern further by giving to role-model diagrams that are satisfied before and after a sequence of operations corresponding to pre- and post-conditions in programming languages.

The example shows (following Buschmann et al. (1996)) the sequence of actions following an event handled by a controller. In this way it is here possible to specify that update is invoked on all observers following a notify and that Model and associated Controllers and Views abstractly share the same State after notify has returned. At least in this sequence of operations. So here lies a possible dilemma in this specification form: The constraints on sequences only applies to that certain sequence while the role model aims at being as abstract as possible.

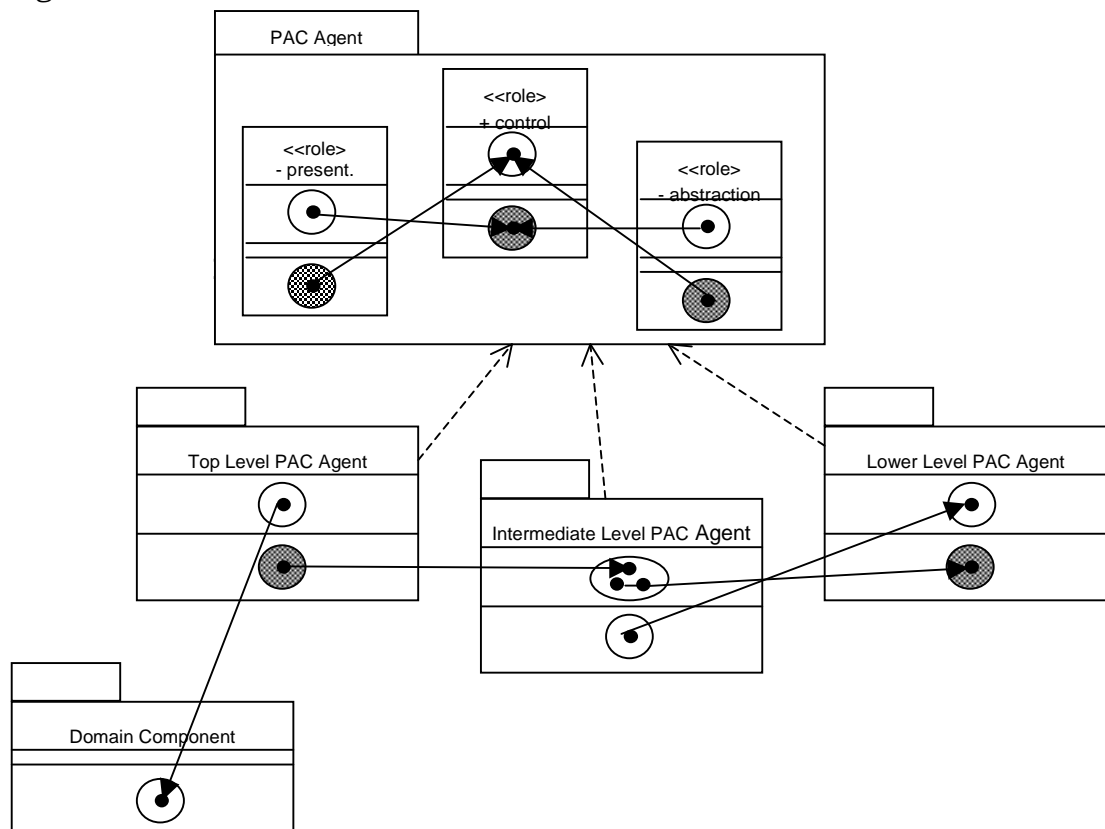


Discussion of Precise Specification

The initial experiments with formalising architectural patterns shows that it may indeed be worthwhile further investigation. Especially LePUS provided an interesting result, and both specification languages force the applicant to consider in depth essential and non-essential elements of patterns.

However, there are elements of software architecture that cannot (directly) be specified in either languages. Following Bass et al. (1998) we regard a software architecture as the structures of a system, comprising software components, their visible properties, and their relationships. In this way other views on a software architecture than call-structure and class and instance relationships such as module structure, physical structure, process structure and more becomes relevant. We therefore believe that a formalism for capturing architectural pattern precisely and purely needs to incorporate other abstractions than the two used in this report do. Specifically, experiments with the specification of the three other patterns – that are in many ways “fuzzier” than MVC in their informal specification – shows that the abstraction of LePUS and Precise Visual Specification may not be adequate for precisely specifying architectural patterns.

Interestingly, however, Precise Visual Specification extends the UML and the UML has proposed notations for describing software architectures especially focusing on Kruchten’s 4+1 view of software architecture (Kruchten, 1995). Initially one could consider specification such as below. Note that this figure very much relies on an informal semantics of abstract state and instance of packages.



Conclusions

The work presented in this report has been initiated by a dissatisfaction with current specifications of architectural patterns – and especially architectural patterns for the construction of interactive systems. This has led the project down two paths: a formal and an informal.

Informally, we have tried to delineate a space of intended readers for pattern descriptions. We do not believe that patterns can be learned or internalized by reading a text. Instead only usage (implementation) or specification (formalisation) provide a solid basis for pattern understanding. Thus we have divided our descriptions into two distinct forms: a short narrative form and a presentation of implementations sharing a common base, in this case a graphical user interface and a problem domain model. In doing this we have found package diagrams much more useful than UML class diagrams (or OMT diagrams) in specifying architectural patterns; the general grouping and association mechanism of packages is more tuned towards architectural specifications than are concrete (or abstract) classes.

Formally, we have just initiated an investigation of precise specification of architectural patterns. Although our current investigations have used contemporary means of design pattern specification this exercise has nevertheless been quite useful: it has highlighted certain commonalities between patterns (e.g. MVC refines the Observer pattern, Application Facade may refine the Application pattern.) Also, the analytical approach employed in precise specifications has spurred a number of questions: What is the significance of differing direction of dependencies in especially Application Facade and Application Adaptor, how important is the encapsulation of the problem domain model (MVC has none), how many “models” are generally used, which levels of description does one apply when specifying architectural patterns, and more.

We thus feel that there is a great deal more to choosing an architecture for an interactive system than picking one of a number of architecture patterns. In particular we would like to combine *unit operations* (Bass et al. 1998) for deriving software architectures with the patterns we have looking at in order to attempt to create an applicable pattern language for software architecture of interactive systems. Also the route of precise specification of architectural patterns seems fruitful.

References

- (Allen, 1997) Allen, R.J. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.
- (Bass et al., 1998) Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*. Addison-Wesley, 1998.
- (Brown et al., 1998). Brown, W.J., Malveau, R.C., McCormick III, H.W., Mowbray, T.J. *AntiPatterns. Refactoring Software, Architectures and Projects in Crisis*. John Wiley & Sons.
- (Booch et al., 1998) Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
- (Buschmann et al., 1996) Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- (Christensen et al., 1998). Christensen, M., Crabtree, A., Damm, C.H., Hansen, Klaus. M.H., Madsen, O.L., Marquardsen, P., Mogensen, P., Sandvad, E., Sloth, L., Thomsen, M. *The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping*. In Proceedings of ECOOP '98, Springer Verlag.
- (Coldewey 1998) Coldewey, J. *User Interface Software*. In Electronic Proceedings of PLoP'98,
- (Coutaz, 1987) Coutaz, J. *PAC, an Object-Oriented Model for Dialog Design*. In Bullinger, H.-J., Shackel (Eds.) *Proceedings of INTERACT'87*. Elsevier Science Publishers B.V., 1987.
- (Eden, 1998) Eden, A.H. *Precise Specification of Design Patterns and Tool Support in their Application*. Ph.D. Dissertaion Draft, Tel Aviv University. <http://www.math.tau.ac.il/~eden/dissertation.doc.zip>.
- (Eden et al., 1998) Eden, A.H., Hirshfeld, Y., Yehudai, A. *LePUS – A Declarative Pattern Specification Language*. Technical Report 326/98, Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.
- (Eden et al., 1999) Eden, A.H., Hirshfeld, Y., Yehudai, A. *Towards a Mathematical Foundation for Design Patterns*. In preparation. <http://www.math.tau.ac.il/~eden/bibliography.html#ecoop99>.
- (Fowler, 1997) Fowler, M. *Analysis Patterns*. Addison-Wesley, 1997.
- (Gamma et al., 1995) Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- (Kent, 1997) Kent, S. *Constraint Diagrams: Visualizing Invariants in Object-Oriented Models*. Proceedings of OOPSLA'97, ACM Press.
- (Knudsen et al., 1994) Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B. *Object-Oriented Environments. The Mjølner Approach*. Prentice Hall, 1994.

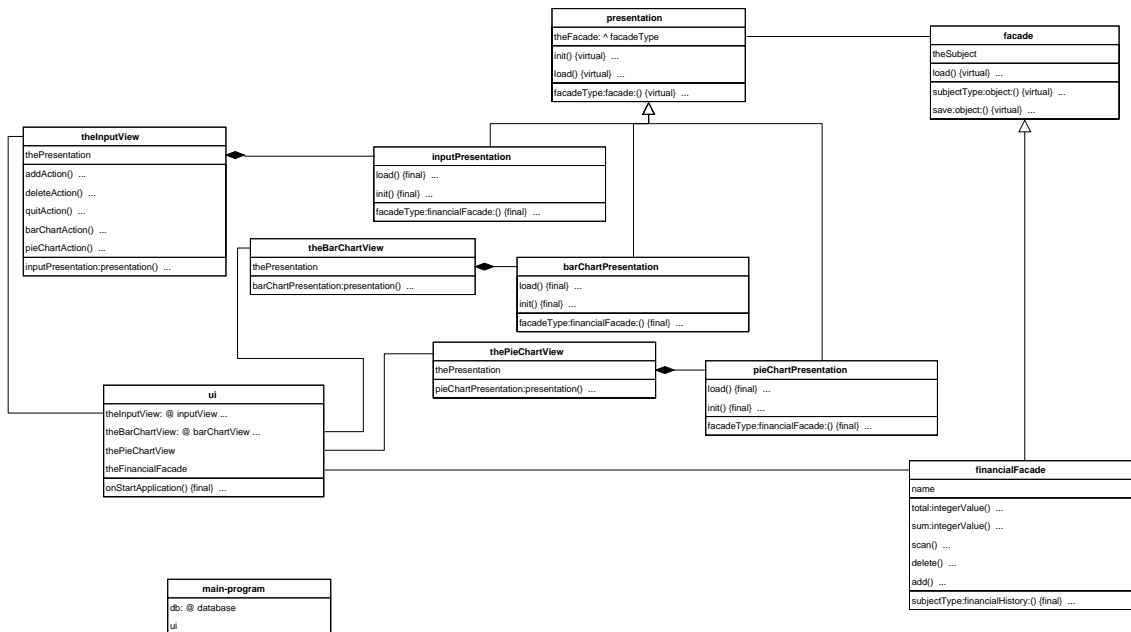
- (Krasner and Pope, 1988) Krasner, G.E., Pope, S.T. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. JOOP, Volume 1(3), August/September 1988.
- (Kruchten, 1995) Kruchten, P. *The 4+1 View Model of Architecture*. IEEE Software, November 1995.
- (Lauder et al., 1998) Lauder, A., Kent, S. *Precise Visual Specification of Design Patterns*. Proceedings of ECOOP'98, Springer Verlag.
- (Madsen et al., 1993) Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: *Object-Oriented Programming in the Beta Language*. Addison Wesley, 1993.
- (Newman and Lamming, 1995). Newman, W.M., Lamming, M.G. *Interactive System Design*. Addison-Wesley, 1995.
- (Shaw, 1996) Shaw, M. *Some Patterns for Software Architectures*. In Vlissides et al. 1996.
- (Vlissides et al., 1996) Vlissides, J.M., Coplien, J.O., Kerth, N.L. (Eds.) *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

Appendices

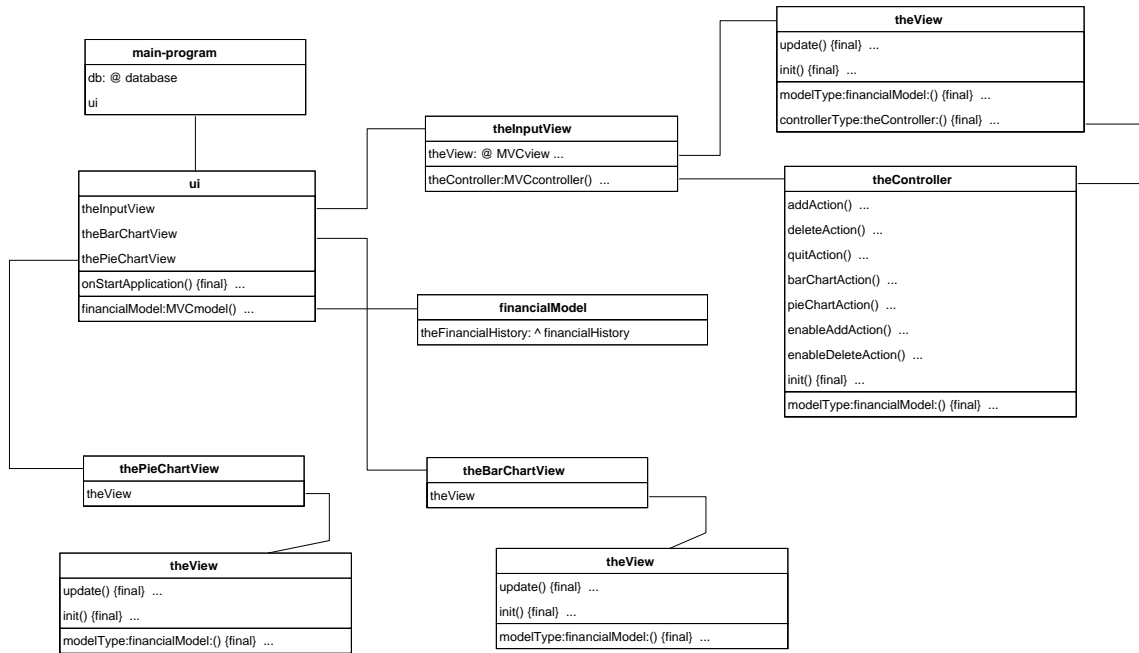
Appendix 1. The User interface code

```
-- guienvLib: Attributes --
inputView: window
  (#
    financesListView: @listView ...;
    addPushButton: @pushbutton ...;
    deletePushButton: @pushbutton ...;
    descStaticText: @statictext ...;
    descEditText: @edittext ...;
    amountStaticText: @statictext ...;
    amountEditText: @edittext ...;
    typeOptionButton: @optionbutton ...;
    quitPushButton: @pushbutton ...;
    showBarChartCheckBox: @checkbox ...;
    showPieChartCheckbox: @checkbox ...;
    open::< ...;
    eventHandler::< ...
  #);
barChartView: window
  (#
    maxBar: ...;
    chart: canvas ...;
    books: @chart ...;
    clothes: @chart ...;
    drinks: @chart ...;
    food: @chart ...;
    other: @chart ...;
    booksStatictext: @statictext ...;
    clothesStatictext: @statictext ...;
    drinksStatictext: @statictext ...;
    foodStaticText: @statictext ...;
    otherStatictext: @statictext ...;
    open::< ...;
    eventHandler::< ...
  #);
pieChartView: window
  (#
    myRect: canvas ...;
    booksRect: @myRect ...;
    clothesRect: @myRect ...;
    drinksRect: @myRect ...;
    foodRect: @myRect ...;
    otherRect: @myRect ...;
    booksStatictext: @statictext ...;
    clothesStatictext: @statictext ...;
    drinksStatictext: @statictext ...;
    foodStatictext: @statictext ...;
    otherStatictext: @statictext ...;
    chart: @canvas ...;
    open::< ...;
    eventHandler::< ...
  #);
```

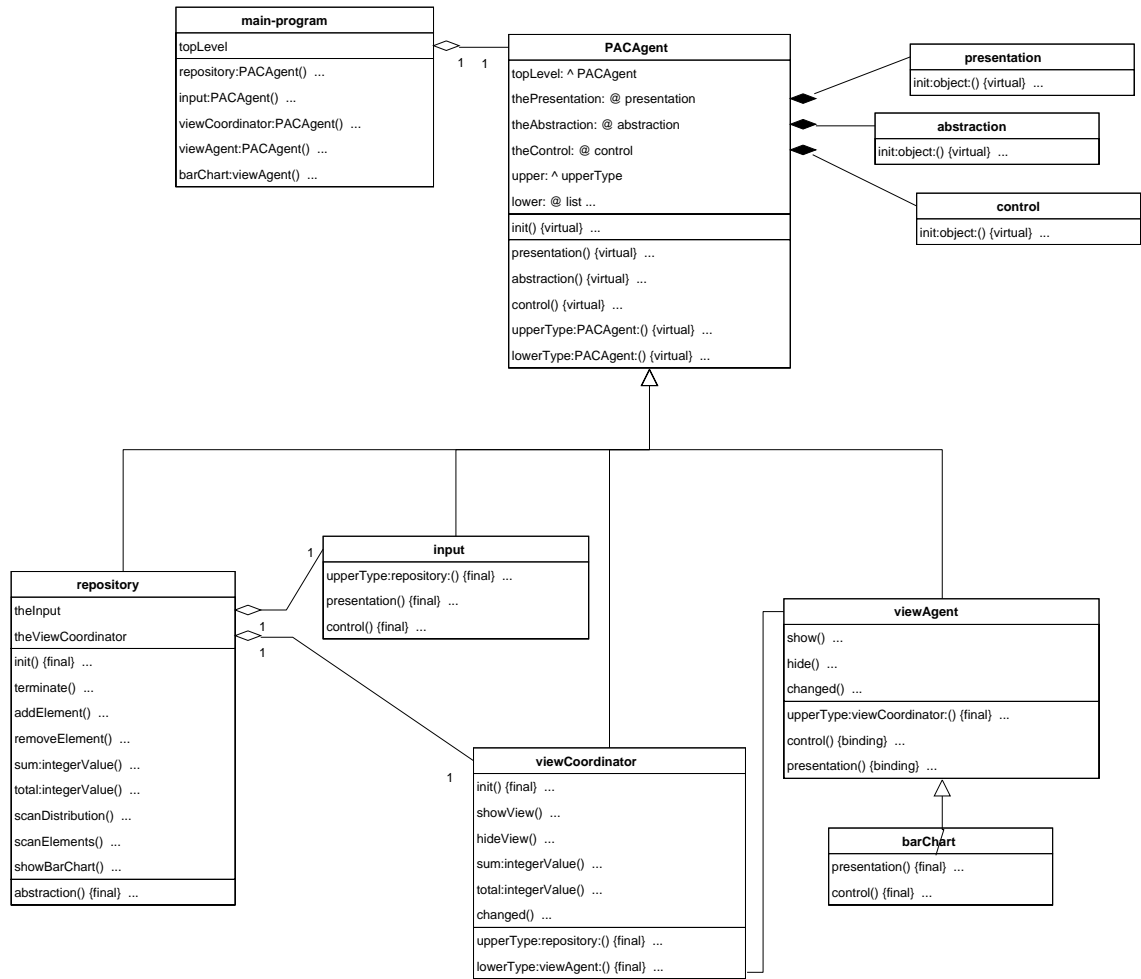
Appendix 2. The structure of Facade



Appendix 3. The structure of MVC



Appendix 4. The structure of PAC



Appendix 5. The structure of Application Adaptor

