Architectural Patterns for Problem Frames (Report)

Christine Choppy[‡], Denis Hatebur[§] and Maritta Heisel[¶]

Abstract

Problem frames provide a characterisation and classification of software development problems. Fitting a problem to an appropriate problem frame should not only help to understand it, but also to *solve* the problem (the idea being that, once the adequate problem frame is identified, then the associated development method should be available). We propose software architectural patterns corresponding to the different problem frames that may serve as a starting point for the construction of the software solving the given problem. We show that these architectural patterns exactly reflect the properties of the problems fitting to a given frame, and that they can be combined in a modular way to solve multi-frame problems. We also provide alternative architectures to cope with specific system characteristics (e.g. distribution).

1 Introduction

Pattern-orientation is a promising approach to software development. Patterns are a means to reuse software development knowledge on different levels of abstraction. They classify sets of software development problems or solutions that share the same structure.

Patterns have been introduced on the level of detailed object oriented design [10]. Today, patterns are defined for different activities. *Problem Frames* [15] are patterns that classify software development *problems*. *Architectural styles* are patterns that characterise software architectures [1, 19]. They are sometimes called "architectural patterns". *Design Patterns* are used for finer-grained software design, while *frameworks* [8] are considered as less abstract, more specialised. Finally, *idioms* are low-level patterns related to specific programming languages [3], and are sometimes called "code patterns".

Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, instead of starting from scratch each time.

It is acknowledged that the first steps of software development are essential to reach the best possible match between the expressed requirements and the proposed software product, and to eliminate any source of error as early as possible. Therefore, we propose to use patterns already in the requirements elicitation phase of the software development life-cycle, as advocated by Fowler [9] or Sutcliffe et al. [21, 22].

M. Jackson [14, 15] proposes the concept of *problem frames* for presenting, classifying and understanding software development problems. A problem frame is a characterisation of a class of problems in terms of their main components and the connections between these components. Once a problem is successfully fitted to a problem frame, its most important characteristics are known.

[‡]LIPN, UMR CNRS 7030, Institut Galilée - Université Paris XIII, France, email: Christine.Choppy@lipn.univ-paris13.fr

[§]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: denis.hatebur@uni-duisburg-essen.de and Institut für technische Systeme GmbH, Germany, email: d.hatebur@itesys.de

[¶]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: maritta.heisel@uni-duisburg-essen.de

Gaining a thorough understanding of the problem to be solved is a necessary prerequisite for solving it. However, when using problem frames, one can even hope for more than just a full comprehension of the problem at hand. Since problem frames are patterns, they represent problem structures that occur repeatedly in practice. Hence, it is worthwhile to look for solution structures that match the problem structures represented by problem frames.

The construction of the solution of a software development problem should begin with the decision on the main structure of the solution, i.e., a decision on the software architecture. Our aim is to exploit the knowledge gained in representing a problem as an instance of a problem frame in taking that decision. For each problem frame, we propose a corresponding architectural pattern that takes into account the characteristics of the problems fitting to the given problem frame.

Our architectural patterns structure the software into layers. Of course, this is not the only possible way of structuring, but a very convenient one. We have chosen it because a layered architecture makes it possible to divide platform-dependent from platform-independent parts, because different layered systems can be combined in a systematic way, and because other architectural styles can be incorporated in such an architecture. That choice has been validated in several industrial projects, dealing for example with smart cards, protocol converters, web/mail-servers, and real-time operating systems.

Like problem frames, our architectural patterns must be instantiated to develop a solution for a concrete problem. The structure provided by an architectural pattern constitutes a concrete starting point for the process of constructing a solution to a problem that is represented as an instance of a problem frame.

The rest of the paper is organised as follows: after introducing the basic concepts of our work in Section 2, we discuss related work in Section 3. In Section 4, we present the architectural patterns associated with each problem frame. In Section 5, we present an example and show how the different solutions of multi-frame problems can be combined. In Section 6, we conclude with a discussion of our approach and directions for future research.

2 Basic Concepts

In this paper, we relate architectural patterns to problem frames. As a notation for our architectural patterns, we use composite structure diagrams of UML 2.0 [24]. In the following, we give brief descriptions of these three ingredients of our work.

2.1 Problem Frames

Jackson [15] describes problem frames as follows:

> 'A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement.'

Solving a problem is accomplished by constructing a "machine" and integrating it into the environment whose behaviour is to be enhanced.

For each problem frame a diagram is set up (cf. left-hand sides of Figures 2, 3, 5, 6, 7, and 9). Plain rectangles denote application domains (that already exist), rectangles with a double vertical stripe denote the machine domains to be developed, and requirements are denoted with a dashed oval. They are linked together by lines that represent interfaces, also called shared phenomena. Jackson distinguishes *causal* domains that comply with some laws, *lexical* domains that are data representations, and *biddable* domains that are people. Jackson defines five basic problem frames (*Required Behaviour, Commanded Behaviour, Information Display, Workpieces* and *Transformation*), and we consider them together with one variant (*Commanded Information*). In order to use a problem frame, one must instantiate it, i.e., provide instances for its domains, interfaces and requirements. A description of these problem frames is given in Section 4 where an associated architecture is proposed.

2.2 Architectural Styles

According to Bass, Clements, and Kazman [1],

> the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Architectural styles are patterns for software architectures. A style is characterised by [1]:

- a set of component types (e.g., data repository, process, procedure) that perform some function at runtime,
- a topological layout of these components indicating their runtime interrelationships,
- a set of semantic constraints (for example, a data repository is not allowed to change the values stored in it),
- a set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

When choosing an architecture for a system, usually several architectural styles are possible, which means that all of them could be used to implement the functional requirements. In the following, we propose concrete architectural patterns for each basic problem frame in order to provide a concrete starting point for the further development of the machine. These architectural patterns are based on a *Layered* architecture.[1] The components in this layered architecture are either *Communicating Processes* (active components) or used with a *Call-and-Return* mechanism (passive components). That design decision is taken in a later step of the development. We also show how the *Repository* and the *Pipe-and-Filter* architectural styles can be mapped to the layered architecture (see Figures 8 and 10). We use UML 2.0 composite structure diagrams (see Section 2.3) to represent architectural patterns in addition to concrete architectures.

2.3 Composite Structure Diagrams

Composite structure diagrams [24] are a means to describe architectures. They contain named rectangles, called *parts*. Theses parts are components of the software. In an object-oriented implementation components are instantiated classes. Each component may contain other (sub-) components. Atomic components can be described by state machines and operations for accessing internal data. In our architectures, components for data storage are only included if the data is stored persistently. Otherwise they are assumed to be part of some other component. Parts may have *ports*, denoted by small rectangles. Ports may have interfaces associated to them. Provided interfaces are denoted using the "lollipop" notation, and required interfaces using the "socket" notation. Figure 1 shows how interfaces in problem diagrams are transformed into interfaces in composite structure diagrams. The partial problem diagram shown on the left-hand side of

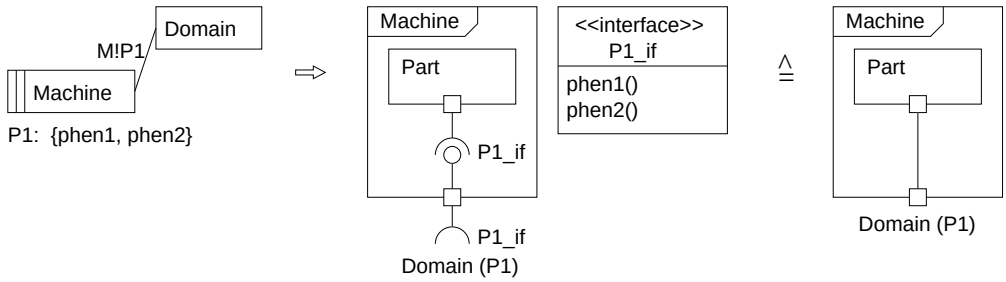


Figure 1: Notation for Architectures

[1] The mentioned architectural patterns are described in [19].

Figure 1 states that the phenomena *phen1* and *phen2* shared between the machine and a domain are controlled by the machine. In the composite structure diagram (with associated interface class) shown in the middle of Figure 1, this is expressed by a required interface *P1_if* of the *part* component of the machine, which is the same as for the whole machine. Shared phenomena controlled by a domain correspond to provided (instead of required) interfaces of the *part* and the machine, respectively. Because of this direct correspondence, we do not use the socket and lollipop notation in the following, but use connectors between ports as shown on the right-hand side of Figure 1. These connectors can be implemented e.g. as data streams, function calls, asynchronous messages or hardware access.

The architecture of software is multi-faceted: there exists a structural view, a process-oriented view, a function-oriented view, an object-oriented view with classes and relations, and a data flow view on a given software architecture. We use the structural view from UML 2.0 that describes the structure of the software at runtime. After that structure is fixed the interfaces need to be refined using sockets, lollipops and interface classes to describe the possible data flow. Then the corresponding active or passive class with its data and operations can be added for each component. Thereby the process-oriented and object-oriented views can be integrated seamlessly into the structural view. That approach and the corresponding process are described in [13].

3 Related Work

Since patterns were introduced, the question of how to make the best use of them in the software development process inspired a number of research activities. Here, we mainly consider those related with the use of problem frames, also in relationship with architectural styles.

Along the idea to integrate problem frames in a formal development process, Choppy and Reggio [7] show how a formal specification skeleton may be associated with some problem frames (Translation and IS as named in [14]). Choppy and Heisel show in [5, 6] that this idea is independent of concrete specification languages. In that work, they also gave heuristics for the transition from problem frames to architectural styles, so as to provide finer structures when moving from the requirements specification to the detailed specification or the design phases. In [5], they give criteria for (i) helping to select an appropriate basic problem frame, and (ii) choosing between architectural styles that could be associated with a given problem frame. In [6], a proposal for the development of information systems is given. The system decomposition is done along different identified use cases.[2] Then, to each use case related to database updates or queries is associated an update or query problem frame (specifically tailored for information systems). A component-based architecture reflecting the repository architectural style is used for the design and integration of the different system parts.

The approach developed by Hall, Rapanotti et al. [11, 18] is quite complementary since the idea developed there is to introduce architectural concepts into problem frames (introducing "AFrames") so as to benefit from existing architectures. In [11], the applicability of problem frames is extended to include domains with existing architectural support, and to allow both for an annotated machine domain, and for annotations to discharge the frame concern. In [18], "AFrames" are presented corresponding to the architectural styles Pipe-and-Filter and Model-View-Controller (MVC), and applied to transformation and control problems.

Finally, let us mention Lavazza and Del Bianco [16] who do not look for architectures, but provide a description of commanded and required behaviour problem frames in UML-RT focusing on active objects or "capsules" communicating through ports (defined by protocols), and they provide a real time version of OCL, called OTL.

[2] Hall et al. [11] base problem decomposition on different requirements statements, which is a similar idea unless some requirements statements refer to a same use case.

4 Architectural Patterns

We now present the six most important problem frames and give their proposed corresponding architectural patterns. These architectural patterns are not pure instances of some architectural style, but they combine elements of different architectural styles to adequately reflect the problem characteristics as given by the respective problem frame. As noted in Section 1, providing an architecture decides on the main structure of the solution. Therefore, while moving from a problem frame (which describes the problem structure) to an associated architecture, we add some proposed design elements. These design elements are not part of the problem frame, but they are proposed as part of the software solution.

Of course, our architectural patterns are not the only possible way to structure the machine domain solving the problem that fits to a given problem frame. However, the kind of (layered) architecture we propose has proven useful in practice (see for example [4, 13, 23]), and allows for combining solutions to different subproblems of complex problems in a systematic way. It is also flexible enough to be combined with other architectural styles.

Our pattern-based software development process using problem frames and architectural patterns proceeds as follows: first, a context diagram showing the problem context is set up (for an example, see Figure 11). Then, the overall problem is decomposed into subproblems that fit to problem frames. This decomposition results in a set of problem diagrams that are instantiated frame diagrams (see examples in Sections 5.1–5.4). For each subproblem, a specification for the machine domain must be derived, thus addressing the frame concern. Each machine domain corresponding to a subproblem is then structured by instantiating the architectural patterns we propose in the following. The instantiated patterns must afterwards be merged to obtain the architecture of the machine solving the overall problem. In Section 5.5, we sketch how this is achieved. Finally, the components of the combined architecture must be specified in more detail, and it must be shown that the combined architecture fulfils the specifications of all subproblems. This last phase is not addressed in the present paper.

The quotations at the beginnings of the subsections are taken from Jackson [15].

4.1 Required Behaviour

The following problems fit to the *Required Behaviour* problem frame:

> 'There is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control.'

The corresponding frame diagram is shown on the left-hand side of Figure 2. The "`C`" in the frame diagram indicates that the *Controlled domain* must be causal. The machine is always a causal domain (so an explicit "`C`" is not needed). The notation "`CM!C1`" means that the causal phenomena `C1` are controlled by the Control machine `CM`. The dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference.

This problem frame is appropriate for *embedded systems*, where the machine to be developed is embedded in a physical environment that must be controlled. The communication between the machine and the physical environment takes place via *sensors* and *actuators*. Thus, only by virtue of sensors and actuators can there be shared phenomena between the machine and its environment. Sensors realize the phenomena `C2` of the frame diagram, i.e., the phenomena controlled by the environment but observable by the machine. Actuators realize the phenomena `C1` of the frame diagram, i.e., the phenomena controlled by the machine and observable by the environment.

For example, we might want to build a machine that keeps the temperature of some liquid between given bounds. Then, the temperature of the liquid would be a shared phenomenon controlled by the environment. The corresponding sensor would be a thermometer. Another shared phenomenon would be the state of a burner. That state would be controlled by the machine, i.e., the machine is able to switch the burner on or off.

Practical work in developing embedded systems [13] has shown that a layered architecture as given on the right-hand side of Figure 2 is appropriate for these systems.

Note that the phenomena `C3` do not occur in the architecture[3], because they do not belong to the interface of the machine domain.

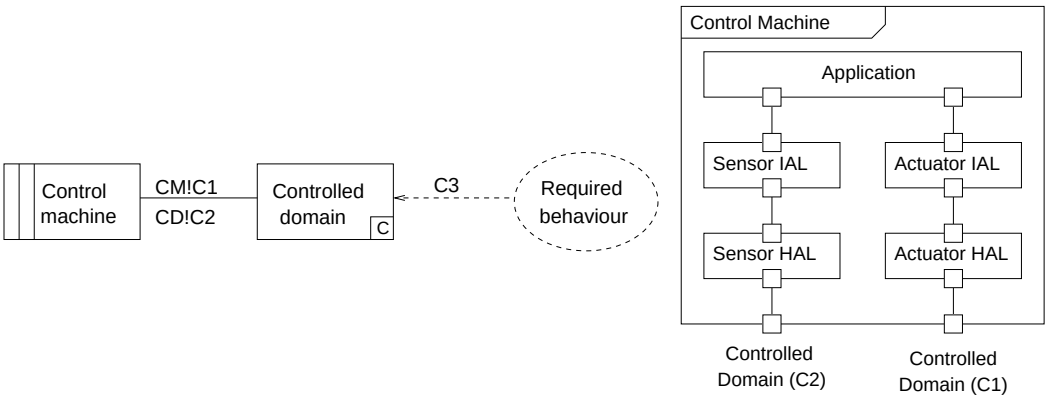


Figure 2: Required Behaviour Frame Diagram and Architecture

The lowest layer is the *hardware abstraction layer* (HAL). This layer covers all interfaces to the external components in the system architecture and provides access to these components independently of the used controller or processor. For porting the software to another hardware platform, only this part of the software needs to be replaced.

The hardware abstraction layer is used by the *interface abstraction layer* (IAL). This layer provides an abstraction of the (low-level) values yielded by the sensors and actuators. For example, a frequency of wheel pulses could be transformed into a speed value. Thus, in the interface abstraction layer, values for the monitored and controlled variables (see [17]) of the system are computed. It is possible that these variables have to be computed from the values of several hardware interfaces. For safety-critical software components, the interface abstraction layer will usually make use of redundant arrangements of sensors and actuators.

The highest layer of the architecture is the *Application* layer. This layer only has to deal with variables from the problem diagram. Therefore, the system requirements can be directly mapped to the software requirements of the application layer, as described by Bharadwaj and Heitmeyer[2].

Thus, the architecture shown on the right-hand side of Figure 2 represents an adequate structure for the `Control machine` of the left-hand side of Figure 2. For special kinds of embedded systems, that architecture could be refined. However, a refinement of the architecture would also correspond to a refinement of the corresponding problem frame. The architecture shown here has the same degree of generality as the problem frame.

4.2 Commanded Behaviour

The following problems fit to the *Commanded Behaviour* problem frame:

> 'There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.'

The corresponding frame diagram is shown on the left-hand side of Figure 3. The "`B`" indicates that the domain `Operator` is a biddable domain, and the phenomena `E4` are the operator commands.

[3]In the following, we use the word "architecture" instead of "architectural pattern" for reasons of readability. It is clear, however, that the components shown in the architectural diagrams have to be instantiated in order to obtain a concrete software architecture.

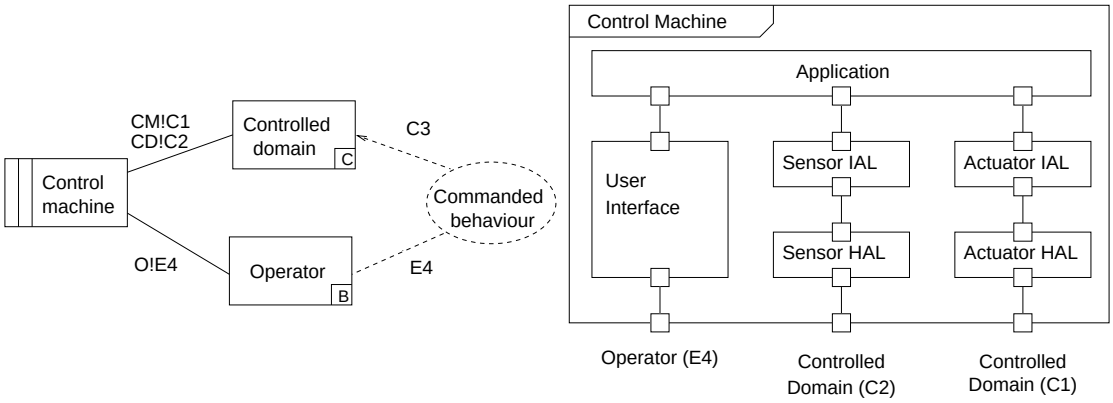


Figure 3: Commanded Behaviour Frame Diagram and Architecture

As can be seen from the frame diagram, the distinguishing feature of the *Commanded Behaviour* frame as compared to the *Required Behaviour* frame is the presence of an operator. That distinction is reflected in the corresponding architecture shown on the right–hand side of Figure 3. The operator commands and the corresponding feedback are handled by a dedicated component *User Interface*. The user interface follows the MVC design pattern [10]. In contrast to the solution discussed in [18] the interface to the *Application* of this component should be the interface to the model, i.e, the *User Interface* comprises the *View* and *Controller* parts of the MVC pattern. With this variation, it can be used in architectures associated with different problem frames. Since

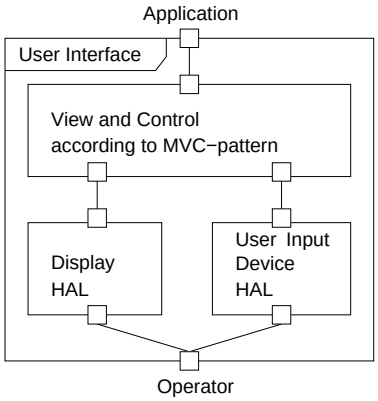


Figure 4: Detailed Architecture for User Interface

each architecture corresponding to a problem frame containing an operator domain will contain a user interface component, we give the structure of such a component in more detail (Figure 4). For reasons of practicality, the user interface component contains not only a sub-component that serves to read user input via some device. In most cases, a sub-component will also be needed that provides some kind of feedback to the user via a display. The physical input arriving at the port at the bottom of the component is transformed into the more abstract phenomena `E4` by the sub-component *View and Control*.

4.3 Information Display

The following problems fit to *Information Display* the problem frame:

‘There is some part of the physical world about whose states and behaviour informa-

tion is continually needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.'

The *Information Display* problem frame offers a structure for applications devoted to the display of real world physical data. The corresponding frame diagram is shown on the left-hand side of Figure 5. The "C" indicates that the `Real World` and `Display` domains are causal. The interface between the `Information machine` and the `Real world` contains only phenomena `C1` that are controlled by the real world. This means that the machine cannot influence the real world. Its purpose is only to display things that happen in the real world.

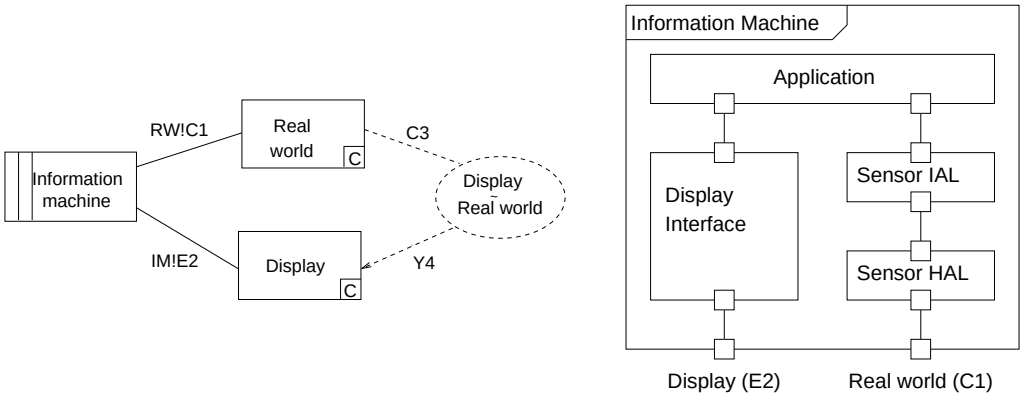


Figure 5: Information Display Frame Diagram and Architecture

Accordingly, the architecture given on the right-hand side of Figure 5 does not contain any components for handling actuators, but only components for handling sensors. There is no operator, but a display is needed. Hence, the architecture contains a display interface.

The *Application* layer of the architecture shown in Figure 5 processes the information yielded by the sensors. If no processing is necessary the application layer can be dropped.

4.4 Commanded Information

The *Commanded Information* problem frame (Figure 6) is derived from the *Simple IS* frame [14]. In [15], the *Commanded Information* frame is presented as a variant of the *Information Display* frame, where an operator is added. The *Commanded Information* frame is very similar to a *Database Query* frame [6], the only difference being that the domain to be displayed need not be causal, but can also be a lexical or a model domain (see [15]).

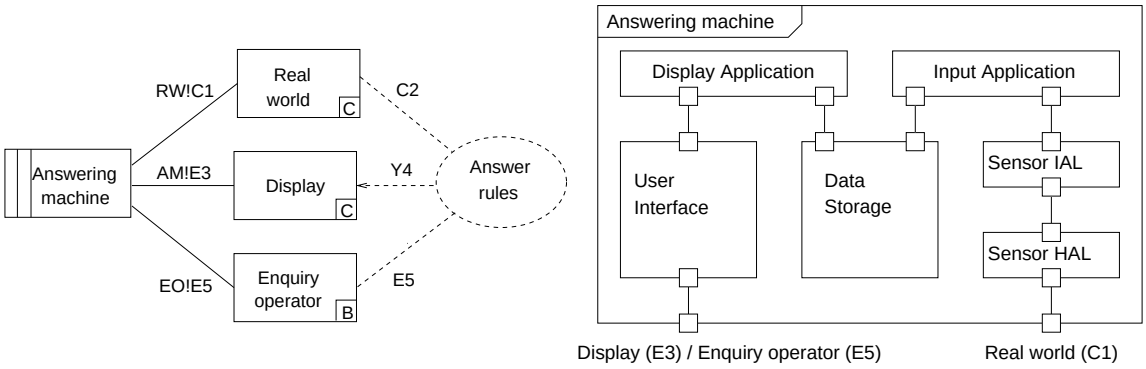


Figure 6: Commanded Information Frame Diagram and Architecture

The architecture we propose for "Answering machines"[4] that solve a *Commanded Information* problem is shown on the right-hand side of Figure 6. To take the presence of an operator into account, the *Display Interface* component of the architecture for the *Information Display* frame is replaced by a *User Interface* component.

Moreover, to cover database applications in addition to the operator-controlled display of physical data, the architecture we propose contains a *Data Storage* component. Of course, this component can be left out if it is not needed to solve the problem. In that case, there would only be one (or even no) application component. Alternatively, for pure database applications, the sensor-handling components of the architecture will not be needed.

4.5 Workpieces

The following problems fit to the *Workpieces* problem frame:

> 'A tool is needed to allow a user to create and edit a certain class of computer processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can act as this tool.'

The "`X`" indicates that the `Workpieces` domain of the frame diagram shown in Figure 7 is a lexical (inert) domain. The *Workpieces* problem frame is very similar to a *Database Update* frame [6].

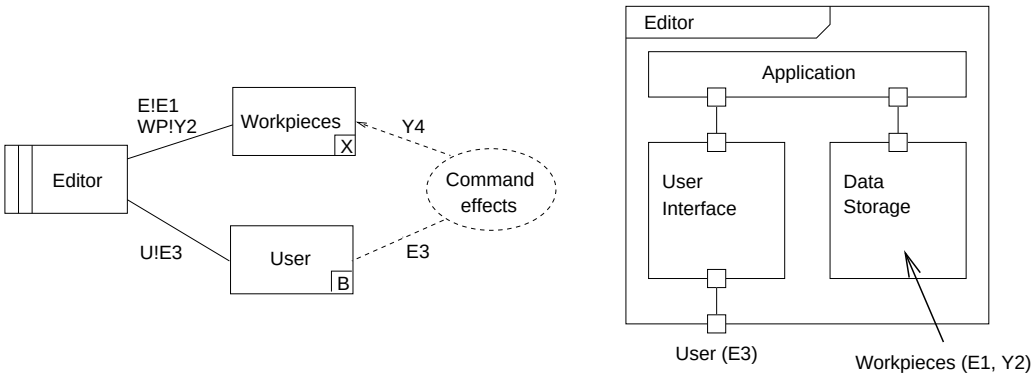


Figure 7: Workpieces Frame Diagram and Architecture

The architecture shown on the right-hand side of Figure 7 contains a user interface component, because the problem frame diagram contains a user. The data storage component of the architecture corresponds to the `Workpieces` domain of the frame diagram. The *Application* component is responsible for manipulating the data storage according to the user commands. Note that there is only one interface with the environment – namely the interface with the user – because the lexical `Workpieces` domain is part of the machine. This holds true also for the input and output domains of the architecture for transformation problems (see Section 4.6). Because our user interface component (see Figure 4) contains not only input but also output facilities, no change in the architecture is necessary if the problem frame is extended with a feedback for changes on workpieces. Such an extention is necessary for realistic problems and user-friendly applications.

Non-functional requirements might state that distributed access to the workpieces must be provided. Such requirements cannot be expressed in problem diagrams. Nevertheless, they may have an influence on the architecture. For this case, we propose a repository architecture (see left-hand side of Figure 8). The repository architecture can be mapped to the layered architecture as shown on the right-hand side of Figure 8 (for one client). Here, remote access to the data storage is possible via a network.

[4]The name "Answering machine" is used by Jackson [15].

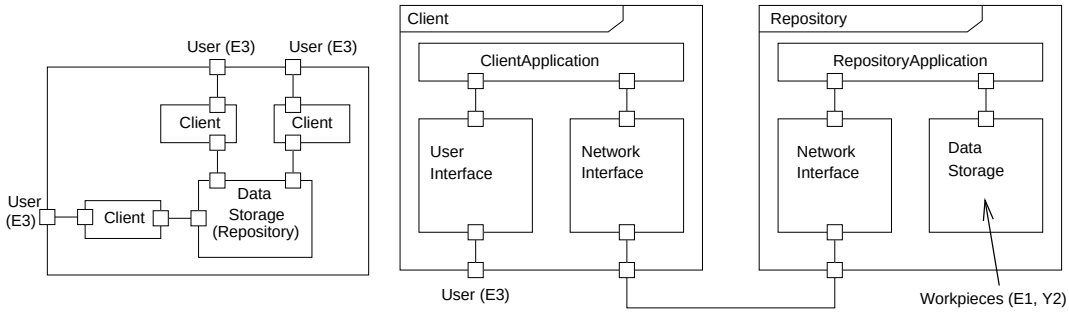


Figure 8: Architecture for Remote Access to Data Storage

4.6 Transformation

The following problems fit to the *Transformation* problem frame:

> 'There are some computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format, and it must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs.'

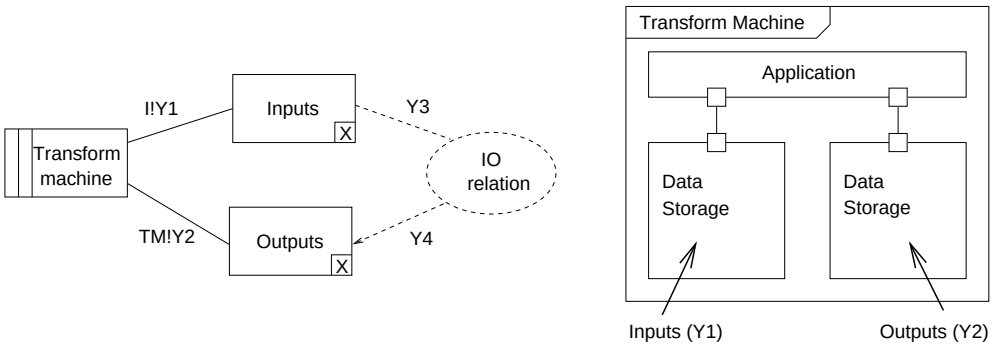


Figure 9: Transformation Frame Diagram and Architecture

Again, the architecture shown in Figure 9 exactly reflects the domains of the frame diagram. Inputs and outputs are stored in data storage components, and the application component is responsible for transforming inputs into outputs. In this architecture there are two data storages. They represent persistent data of perhaps different structure. One is for the inputs (e.g., source code) the other for outputs (e.g., an executable file).

Often, pipe-and-filter architectures are useful for solving transformation problems (another possibility is suggested in [18] where the pipe and filter architectural style is combined with the transformation frame). The pipe-and-filter architecture can be combined with the layered architecture. It can be used to connect parts in the application layer using the *Pipe* that is provided by the layer below. Figure 10 shows how the pipe-and-filter architecture can be integrated into the layered architecture.

In summary, our architectural patterns reflect the problem frames in the following way:

- The interfaces of the architectural patterns correspond exactly to the interfaces of the machine domains as defined in the different frame diagrams. Hence, the architecture refines exactly the machine to build; it neither adds nor leaves out any shared phenomena as compared to the problem description.

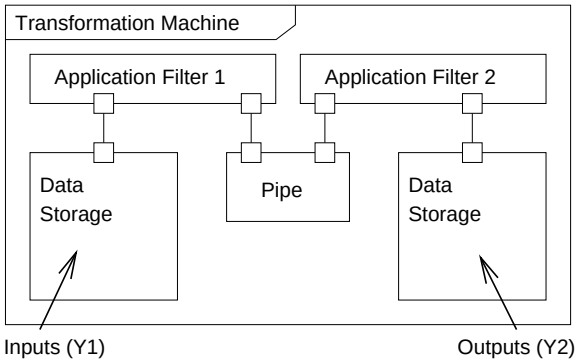


Figure 10: Transformation Architecture with Pipes and Filters

- If the machine has interfaces with causal domains, the corresponding architectural pattern contains components for handling sensors and actuators. This reflects the way in which software can communicate with and influence the physical world.
- If the frame diagram contains a biddable domain (i.e., an operator or user), then the corresponding architectural pattern contains a user interface component.
- If the machine has interfaces with lexical domains, these domains are reflected as parts of the corresponding architectural pattern. This must be the case, because lexical domains can only exist inside the machine.

5 Multi-frame-Problem: Automatic Teller Machine (ATM)

To illustrate how our architectural patterns help solving problems that fit to a given problem frame, we consider an automatic teller machine (ATM). The mission of an ATM is to provide customers with money, provided that they are entitled to withdraw the desired amount. Figure 11 shows the structure of the ATM problem context, where several domains are introduced.

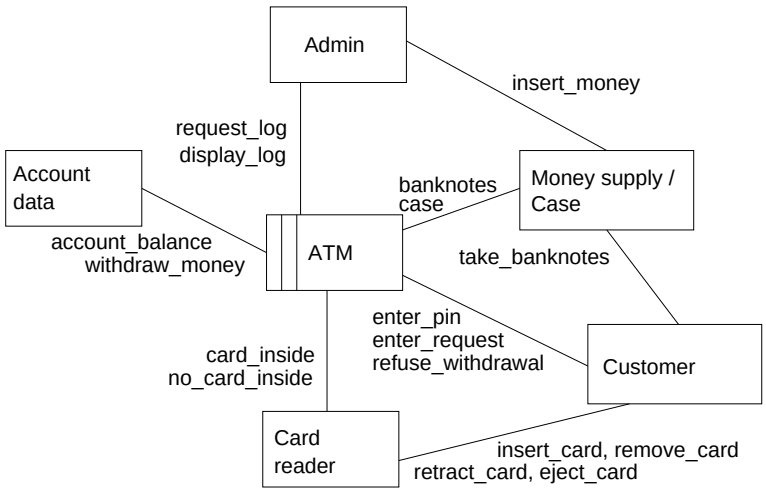


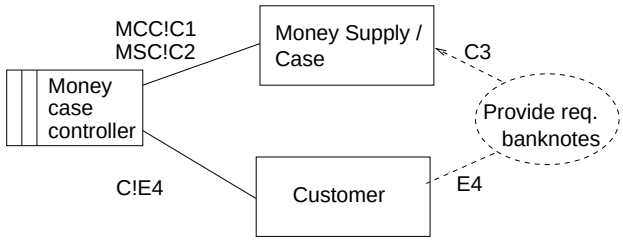
Figure 11: Context Diagram for ATM Problem

The ATM is an example of a multi-frame problem, i.e., it consists of several subproblems that can be fitted to different problem frames. In the following, we will identify the subproblems (namely money-case control, card reader control, log file and update account), fit them to an

appropriate problem frame, and derive the corresponding software architectures according to the patterns given in Section 4. Finally, we will compose the architectures yielded by the different subproblems to obtain an architecture for the whole ATM.

5.1 Money-Case Control Subproblem

The first subproblem of the ATM controller problem concerns the control of the money case. To deliver money to the customer, the ATM has a case where it puts the requested banknotes. This case has a shutter visible to the customer. To prevent another person from taking forgotten money, the ATM will only open the case for a limited amount of time. When that time is over, the ATM retracts the money from the case. To open the case, the ATM starts opening the case and stops when the case is open. The instantiated frame diagram for *Commanded Behaviour* is shown in Figure 12. The customer asks for the desired amount (shared phenomena `E4`), and the money case controller orders the various actions to be done accordingly (shared phenomena `C1`) while receiving information from the money supply case through sensors (shared phenomena `C2`). The corresponding architecture is shown in Figure 13, where the connections between the main application and the user interface as well as the money supply case are given. The phenomena `C1` and `C2` are refined from the more general phenomena `case` and `banknotes` from the context diagram of Figure 11.



C1: {*put_banknote_to_case, start/stop_open case, take_banknotes_from_supply, start/stop_close_case, retract_banknotes_from_case*}

C2: {*case_is_open, case_is_closed, banknotes_removed*}

C3: {*banknotes_in_case*}

E4: {*enter_ request, enter_pin*}

Figure 12: Problem Diagram for Money-Case Control

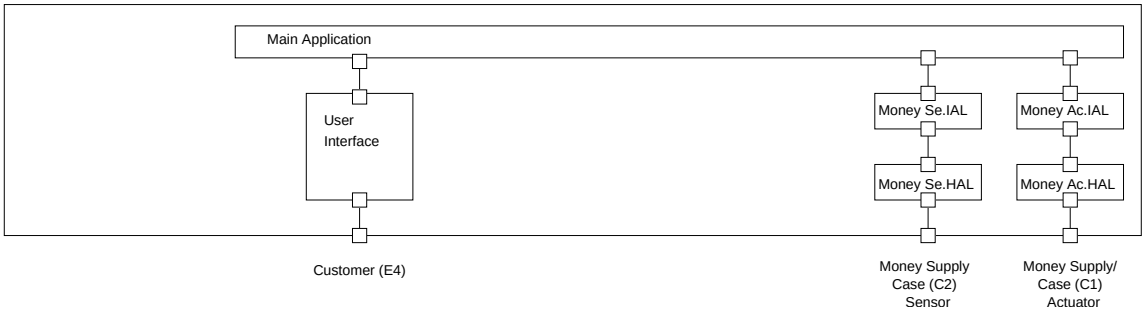
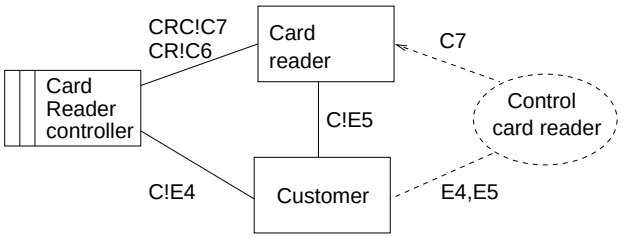


Figure 13: Architecture for the Money-Case Control Subproblem

5.2 Card Reader Control Subproblem

The second subproblem concerns controlling the card reader. Once a user has inserted his/her card, it is under control of the ATM. When the user has successfully withdrawn the money, the ATM ejects the card. In case of an invalid card or three unsuccessful attempts of authentication, the card will be kept by the ATM. After the ATM has ejected the card, the user must remove it within some time. Otherwise it will be retracted again to prevent somebody else from taking it.

With a slight variation, this subproblem also fits to the *Commanded Behaviour* frame as shown in Figure 14. The variation concerns the connection between the `Customer` and the `Card reader` that are connected with the shared phenomena `E5`. The reason for the variation is that on the one hand, the card reader is the domain to be controlled. On the other hand, the card reader acts as a *connection domain* (see [15]) between the customer and the machine: the customer inserts or removes the card (phenomena `E5`), and the card reader informs the machine of these phenomena by transforming them into phenomena `C6`. However, this subproblem is still a *Commanded Behaviour* problem, because the additional connection between customer and card reader does not concern the interfaces of the machine.



C6: {*card_inside, no_card_inside*} E5: {*insert_card, remove_card*}
C7: {*eject_card, retract_card*}

Figure 14: Problem diagram for Card Reader Control

The instantiated architectural pattern in Figure 15 covers all interfaces relevant for this subproblem. The phenomena `E4` are the same as in the previous subproblem (Section 5.1).

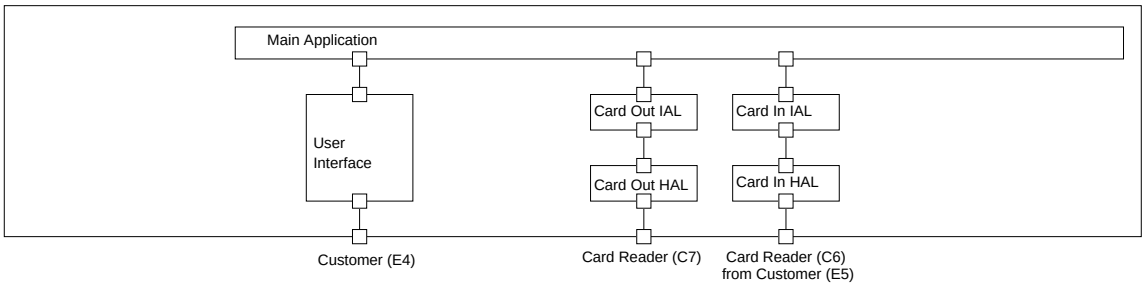
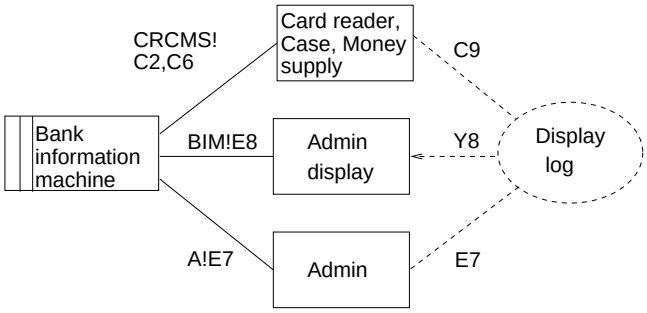


Figure 15: Architecture for the Card Reader Control Subproblem

5.3 Log File Subproblem

The third subproblem of the ATM controller problem concerns the log file of the ATM. The bank needs to know for example when the customer does not remove his/her money and when the ATM retracts a card or the money. These actions should be logged by the ATM, and some bank employee (called *Admin*) should be able to read selected parts of the log. The log file subproblem covers actions and domains that fit to the *Commanded Information* frame. The instantiated problem frame is shown in Figure 16, and the instantiated architecture is shown in Figure 17. The phenomena `C2` and `C6` are the same as in the subproblems of Sections 5.1 and 5.2.



E7: {*request_log*} Y8: {*log*}
E8: {*display_log*} C9: {*ATM_actions*}

Figure 16: Problem Diagram for Log File

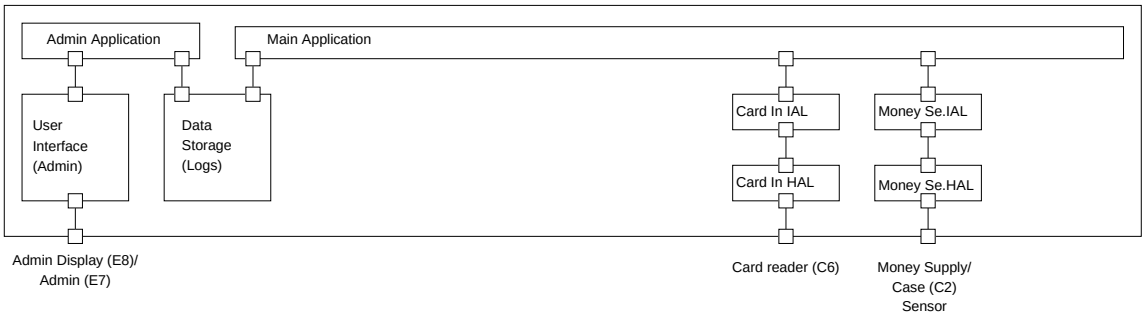
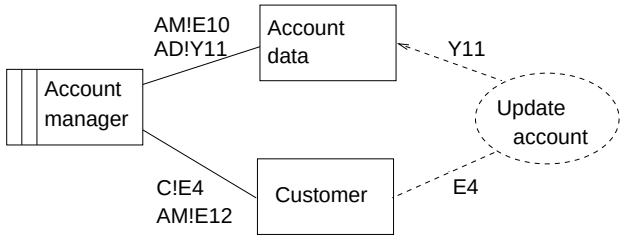


Figure 17: Architecture for the Log File Subproblem

5.4 Update Account Subproblem

The last subproblem is an instance of a variant of the *Workpieces* frame and concerns the account of the customer. To withdraw money, it must be checked whether the requested amount of money exceeds the limit associated with the account. Moreover, the account data have to be updated in case of a successful withdrawal. Figure 18 shows the problem diagram. The phenomena `C4` are the same as in the subproblem of Section 5.1.

The variation of the *Workpieces* frame concerns the phenomenon *refuse_withdrawal* (`E12`), which is used by the `Account manager` machine to provide feedback to the customer. In the *Workpieces* frame, no customer feedback is provided (see Figure 7). For real applications, however, feedback is needed. The *Update Information System* frame of [6] contains an extra domain for feedback output. Since users are reflected in our architectural patterns by *User Interface* components that contain a display interface, our architectural pattern covers the frame variation.



E10: {*withdraw_money*} E12: {*refuse_withdrawal*}
Y11: {*account_balance*}

Figure 18: Problem Diagram for Update Account

Figure 19 shows a simple architecture for the update-account subproblem. In real ATMs, the account data would be stored in a distributed data storage. Hence, the database may be replaced by a network interface and a repository as shown in Figure 8.

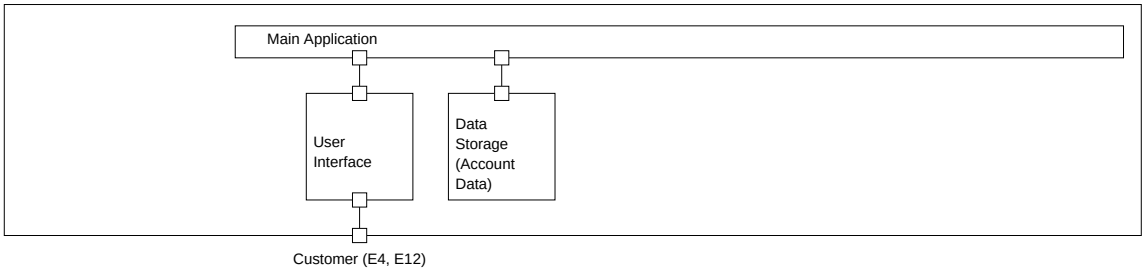


Figure 19: Architecture for the Update Account Subproblem

5.5 Composed Architecture

We now must compose the architectures developed for the subproblems to obtain an architecture for the whole ATM. For doing this, we must find the parts occurring in different subproblem architectures, that must be identified in the composed architecture. In particular, this concerns the sensor and actuator handling parts. Parts that occur in different subproblem architectures, but handle the same phenomena (for example, `C1` or `E4` in the ATM problem) should occur only once in the composed architecture.

In contrast to [20], where a bus system is the central part and several hardware components access this bus system to exchange information, we need to find a composition for a single microprocessor system. Therefore the application layer must be able to handle the full functionality as given by the application components of the subproblems. Ideally, the functionalities of the subproblems should be implemented in different components (parts) of the overall application layer. However, giving methods for designing the overall application layer is subject to future work.

The correctness of the composed solution must be shown in a separate step. For all components, their exact specifications must be set up, and it must be shown that the components work together in such a way that they fulfil the specifications of all machines corresponding to the different subproblems. These issues cannot be expressed at the level of architectural patterns.

The composed architecture for the ATM is shown in Figure 20. This example shows that our patterns yield appropriate architectures for subproblems fitting to problem frames, and that these architectures can be combined in a modular way to obtain an architecture of the overall system.

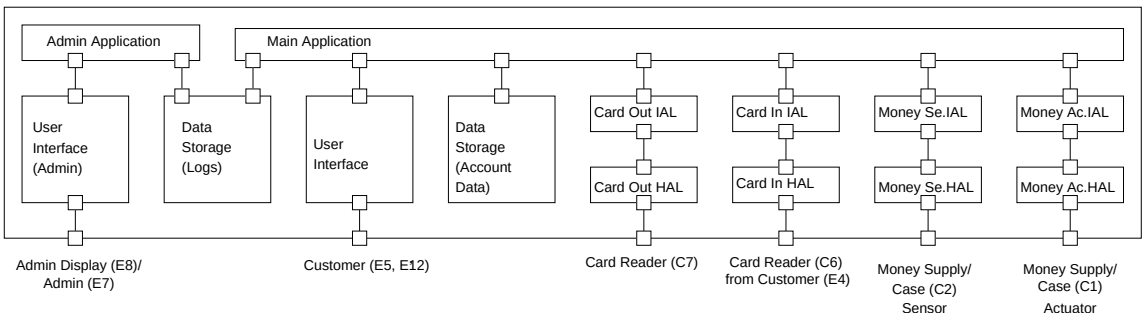


Figure 20: Composed Architecture

6 Conclusions

In the present paper, for each basic problem frame (together with the commanded information variant), an associated architecture is defined that is expressed using composite structure diagrams of UML 2.0. The contributions of our approach are the following:

- We provide a way to start describing, through a given architectural pattern, solution structures associated with basic problem frames.
- To this end, we designed new architectural patterns specifically elaborated to reflect the characteristics of the various problem frame domains and their intended refinement when designing the solution.
- The components of problem frames are reflected in the architectures, and a clear correspondence between domains of frame diagrams and components of architectures is established.
- The provided architectural patterns as well as the problem frames are quite generic and cover a wide range of problems; both can be refined to accommodate more specific kinds of problems.
- The architectural patterns support the recomposition of solutions developed for different subproblems of multi-frame problems in a systematic way.

While in [11, 18] a related approach is taken to extend problem frames with architectural concepts, so as to introduce more knowledge and information into problem frames, our approach is to propose architectural patterns dedicated to the basic problem frames. It can be used when moving to the design phase (or even in the detailed specification phase) of the software development as advocated in [5, 6]. We see the two approaches as complementary: while [11, 18] provide a way to incorporate/reuse domain knowledge in the problem description, we focus on proposing a way to move from the problem description to the solution description.

Although the work presented here is independent of any formal specification language, if desired, it would be possible to accompany the architectural descriptions with a formal specification development along the ideas of [5, 6, 7] (see Section 3).

In the future, we intend to extend this work in several directions. We would like to describe the behavioural aspects associated with the architectural patterns, and to specify in more detail the communication between the different architecture components, for instance using communication patterns. As we rely on a problem decomposition, we need to investigate in more detail how the composition of architectures can be achieved and how the subproblem applications can be incorporated in the overall application.

We also plan to consider frame concerns, since they are an important part of problem frames as an analytical tool. In particular, frame concerns (and their derived correctness arguments) relate to a rich traceability of requirements through to a solution (see for example the Praxis REVEAL approach [12]). Hence, it is important to know how the proposed solution architectures contribute to their discharge.

Moreover, since our approach aims at a guided and integrated use of several techniques and several patterns, we would like to explore how to integrate the use of design patterns in this development.

Acknowledgements. The authors would like to thank the anonymous referees for their helpful comments.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] R. Bharadwaj and C. Heitmeyer. Hardware/Software Co-Design and Co-Validation using the SCR Method. In *Proceedings IEEE International High-Level Design Validation and Test Workshop (HLDV 99)*, 1999.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[4] J. Cheesman and J. Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[5] C. Choppy and M. Heisel. Use of patterns in formal development: Systematic transition from problems to architectural designs. In M. Wirsing, R. Hennicker, and D. Pattinson, editors, *Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers*, LNCS 2755, pages 205–220. Springer Verlag, 2003.

[6] C. Choppy and M. Heisel. Une approche à base de "patrons" pour la spécification et le développement de systèmes d'information. In *Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2004*, pages 61–76, 2004.

[7] C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th WADT, Selected Papers*, LNCS 1827, pages 104–123. Springer Verlag, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

[8] M. E. Fayad and R. E. Johnson. *Domain-Specific Application Frameworks*. John Wiley and Sons, 1999.

[9] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[11] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating Software Requirements and Architectures using Problem Frames. In *Proceedings of IEEE International Requirements Engineering Conference (RE'02)*, Essen, Germany, 9-13 September 2002.

[12] J. Hammond, R. Rawlings, and A. Hall. Will it work? In *Proceedings of 5th IEEE International Requirements Engineering Conference (RE'01)*, Toronto, Ont., Canada, August 2001.

[13] M. Heisel and D. Hatebur. A model-based development process for embedded systems. In T. Klein, B. Rumpe, and B. Schätz, editors, *Proc. Workshop on Model-Based Development of Embedded Systems*, number TUBS-SSE-2005-01. Technical University of Braunschweig, 2005.

[14] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[15] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[16] L. Lavazza and V. D. Bianco. A UML-Based Approach for Representing Problem Frames. In K.Cox, J. Hall, and L. Rapanotti, editors, *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. IEE Press, 2004.

[17] D. L. Parnas and J. Madey. Functional documents for computer systems. In *Science of Computer programming*, volume 25, pages 41–61, 1995.

[18] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture Driven Problem Decomposition. In *Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04)*, Kyoto, Japan, September 2004.

[19] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[20] C. P. Shelton and P. Koopman. Developing a Software Architecture for Graceful Degradation in an Elevator Control System. In *Proc. Workshop on Reliability in Embedded Systems*, 2001.

[21] A. Sutcliffe. *The Domain Theory, Patterns for Knowledge and Software Reuse*. Addison Wesley, 2002.

[22] A. Sutcliffe and N. Maiden. The Domain Theory for Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, 1998.

[23] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. TAN a 92:1 2.Ex.

[24] UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.