Pattern-based Evolution of Software Architectures

Isabelle Côté, Maritta Heisel, and Ina Wentzlaff

University Duisburg-Essen, Faculty of Engineering, Department of Computational and Cognitive Sciences - CoCoS, Working Group Software Engineering, Germany

{isabelle.cote, maritta.heisel, ina.wentzlaff}@uni-duisburg-essen.de

Abstract. We propose a pattern-based software development method comprising analysis (using problem frames) and design (using architectural and design patterns), of which especially evolving systems benefit. Evolution operators guide a pattern-based transformation procedure, including re-engineering tasks for adjusting a given software architecture to meet new system demands. Through application of these operators, relations between analysis and design documents are explored systematically for accomplishing desired software modifications. This allows for reusing development documents to a large extent, even when the application environment and the requirements change.

1 Motivation

Splitting the software life cycle into several, more or less independent development phases is a need to create manageable engineering activities. Patterns introduce a further enhancement, because they provide a concept for reusing software development knowledge. Hence, a vast quantity of patterns specific to and applicable in the different phases of the software life cycle can be found today.

It has been observed that the life-span of software often covers several years, in some cases even decades. During this long lifetime, it is necessary to modify and update existing software to accommodate it to new requirements or a changing environment, where it is deployed in. Modifying existing software systems to adapt them to new or changed requirements is called *software evolution*. Existing software development processes, however, are not designed to incorporate new or changing requirements into an existing system. They usually consider a system that has to be built from scratch. This is a striking fact, as experts see the fraction of maintenance/evolution at 80% of the overall effort for a software project [9]. For this reason, it is necessary to provide systematic support for the evolution task. Ideally, this support can be embedded into an existing development process. This puts new demands on the relation of software development process and pattern usage. Re-engineering techniques and reuse, as well as the traceability between the different development artifacts are crucial in this context.

Each phase of the software life cycle has different objectives. However, even if the engineering activities of the respective software development steps are independent of each other, the resulting artifacts are not. Sudden architectural change by innovation is not desirable. Instead, architectural changes are usually motivated by the need for adding new functionality or reorganizing existing functionality to cope with new

environmental circumstances. Therefore, we investigate the role of application environment and requirements change for architectural evolution. Our approach presented here stresses the early development stages. We introduce evolution operators that guide a systematic software architecture adjustment by establishing profitable pattern relations.

Patterns for software development have become widely accepted, especially during the last decade. They are a means for understanding a given development problem (problem frames [7]), or structuring its solution (as architectural styles [1] or design patterns [5]). Relating these patterns for different phases of the software life cycle [2, 10, 11, 14] in a methodical manner avoids to construct throw-away models. Thus, artifacts of the respective steps that are build on patterns depend on each other, explicitly.

Guiding the transfer of artifacts of the analysis phase to artifacts of software design by a pattern-based transformation procedure takes advantage of this underlying, pattern-based linkage between artifacts and its resulting traceability. Therefore, we extend an existing development process through suitable evolution operators. These operators guide the engineering process in evolving given artifacts. They also assist in the reuse of related development artifacts in successive development phases and provide help with selecting appropriate patterns. Because of its general nature, our pattern-based development method is applicable to a variety of software development problems and assists system evolution.

Section 2 describes our general development method. To illustrate our approach, we present as an example the architectural evolution of a chat system that is introduced in Section 3. Section 4 presents the evolution operators we have defined and attached to the development method of Section 2. In Sections 5 and 6, two evolution scenarios are given, describing the accomplishment of a pattern-based evolution of the original chat software architecture. Finally, Section 7 concludes this work with a brief summary of our contribution and future prospects.

2 A Pattern-Based Software Development Method

Our pattern-based software development procedure is based on Jackson's problem frames approach [7]. It consists of the following four steps:

1. Understand the problem situation

   After investigating the problem domain and identifying its current shortcomings, relevant domain knowledge is collected, and the system mission together with the corresponding requirements are recorded, as shown in Table 1. Domain knowledge (consisting of facts `F` and assumptions `A`) and requirements `R` (describing the chat system in its environment) are collected for detailing the system mission (`SM`).
   The given problem situation is structured by a context diagram, which represents the desired system, shown in Figure 1. It represents the overall problem situation. A context diagram covers the domain knowlege and the requirements of Table 1 by corresponding domains (boxes) and their interactions (*shared phenomena* at labeled interfaces). The *machine domain* (indicated by two vertical stripes) represents the software we are going to build.

SM	*A text-message-based communication platform shall be developed, which allows multi-user communication via private I/O-devices.*
R1	Users can phrase text messages, which are shown on their private graphical display.
R2	Users send their phrased text messages to participate in the chat.
R3	Sending text messages changes the chat represented on the users' graphical display.
⋮	...
F1	Users communicate in a local network.
A1	Users follow the course of the chat on their private graphical display.

Table 1. Initial Set of Requirements and Domain Knowledge for a Chat Application

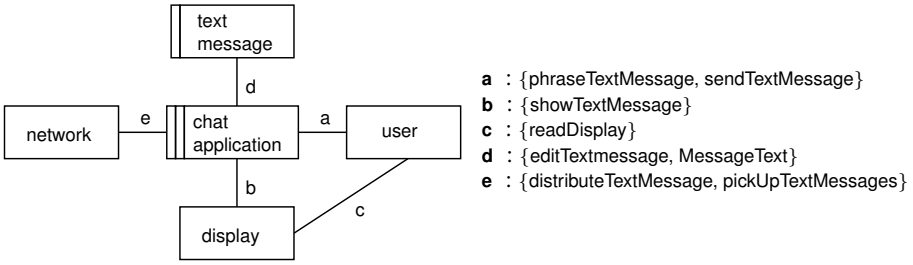

Fig. 1. Context Diagram for a Chat Application

2. Decompose overall problem into simple subproblems

   The requirements guide a knowledge-based decomposition of the overall problem that is represented by a context diagram into several simple problems. A simple problem is represented by a problem diagram, which expresses what the subproblem is about by referring to the involved domains and related shared phenomena. These subproblem representations together with the given domain knowledge suffice to derive a software *specification* describing the interface behavior of the machine. Figures 2 and 3 represent two simple subproblems for requirements R2 and R3. [1]

3. Fit subproblems to (variants of) problem frames

   When using a pattern-based development method, the subproblems are classified by instantiating suitable problem patterns, called problem frames (PF) [7]. These are patterns categorizing software development problems into problem classes during the analysis phase. Thus, each problem frame represents a problem category, which can be linked to patterns of a corresponding solution class. Via analogies, they can be related to patterns of software design, resulting in a smooth transition from requirements engineering documents into design artifacts [11].

[1] Figures 2 and 3 are explained in more detail in Section 3.

4. Instantiate corresponding architectural and design patterns

   Finally, we make use of the *problem/solution-pattern relationship* discussed in the previous Step 3 to derive a software design appropriate for the given problem situation documented in Steps 1 and 2. As an instance of a problem frame, each derived subproblem assigns values to its related architectural styles or design patterns. As a result, we obtain a more or less coarse-grained design (see Figures 4, 6, and 7) for each subproblem. These subproblem solutions can be used as a starting point for additional solution refinement, component deployment, or coding.
   Therefore, using a specific pattern in the analysis phase results in a predetermined choice of patterns in the design phase. If a subproblem fits to a problem frame (as in Figures 2 and 3), related architectural or design patterns (see Figures 4, 6, and 7) can offer a solution structure for it.

In the following, we use design patterns such as *Forwarder-Receiver* [1], which are comparable to architectural styles for developing software architectures. However, composing the overall (architectural) design out of several subproblem solutions [3] is out of scope of this paper. We consider subproblems that contribute to a common solution design in this paper.

3 Example: Developing a Chat System

The starting point of our initial software development project is the system mission in Table 1. As described in Steps 1 and 2 of our development method, domain knowledge and requirements are collected, and a context diagram is set up (see Figure 1). The problem diagrams of Figures 2 and 3 represent two of the subproblems derived for the chat application. They refer to the requirements `R2` and `R3`.

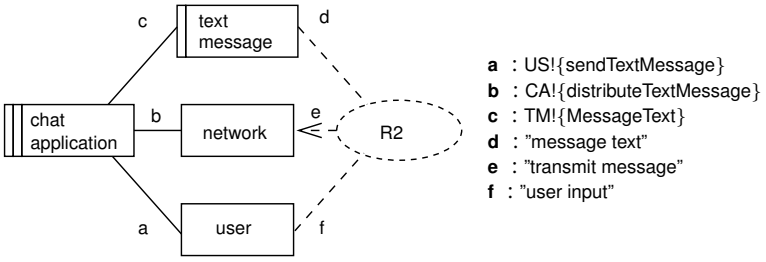

Fig. 2. "Message Forwarder" subproblem (instance of a variant of the *commanded behaviour* PF)

If a problem requirement such as `R2` and `R3` can be fitted to a problem frame, corresponding domains and shared phenomena for describing it in detail become identifiable. Then, the dashed oval contains the corresponding requirements. The dashed lines demonstrate the relationships between these requirements and the different problem domains, see for example Figure 2. An arrowhead pointing at a problem domain denotes a *requirements constraint*, which stipulates the development of a machine controlling the

problem domain as stated in the requirements. Then, the frame diagram supports the derivation of a *specification*, which is a technical description sufficient for developing the desired software. For this, the interfaces at the machine domain are of particular importance. They specify what services the desired software shall provide.

In addition to the interfaces of a context diagram, in problem diagrams an abbrevation of the domain name (like US for user) is given. An exclamation mark at the labeled interfaces indicates which domain *controls* a shared phenomenon or a set of shared phenomena. An example for this is **a**: US!{sendTextMessage} in Figure 2. It means that the user initiates commands for sending chat text messages.

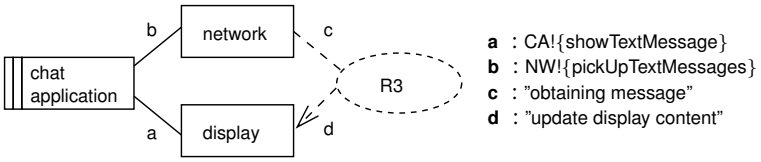

Fig. 3. "Message Receiver" subproblem (instance of *information display* PF)

As recommended by Step 3 of our pattern-based software development method, both subproblems of Figures 2 and 3 are instances of problem frames [7]. In Figure 4, we illustrate how to transform a pattern for software analysis to a pattern for software design in order to accomplish Step 4.

First, we translate the subproblems into a class diagram known from Unified Modeling Language (UML) [13] (see classes chat application and network of Figure 4). This eases linking them to patterns of software design, which in general are represented in UML notation. Thus, in our approach, problem frames and problem diagrams take the role of UML use cases, which are a means for requirements elicitation and problem decomposition. In contrast to use cases, problem frames and problem diagrams refer to their respective requirements explicitly. Furthermore, they represent necessary objects and their interactions in more detail, which facilitates for a more coherent development. And in addition, they support our aim of an integrated pattern-based development procedure. Consequently, problem frames do not replace common development notations such as UML, but they extend them in a profitable way.

The upper part of the class diagram shown in Figure 4 represents the *Forwarder-Receiver* design pattern [1]. The two classes chat application and network are taken from the subproblems in Figures 2 and 3. They are related to the *Forwarder-Receiver* design pattern via analogy:

The software we are going to build, namely the chat application is related to the class Peer in Figure 4. Both have in common that they constitute the core chat system controlled by a user, who can call specific services, such as sendTextMessage. Therefore, we relate them via an inheritance relation. The network domain in Figure 2 takes the role of the class Forwarder in Figure 4, implementing its operations by distributeTextMessage. In the subproblem of Figure 3, network takes the role of a Receiver, which again is expressed by an inheritance relation in Figure 4, where network implements the Receiver operations by pickUpTextMessages. In the following, the domain

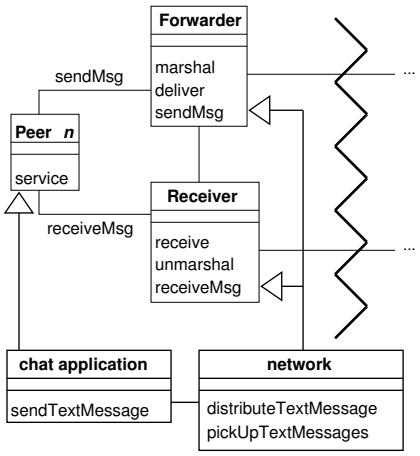

Fig. 4. Relating Problem Diagrams with Design Pattern *Forwarder-Receiver*

text message of Figure 2 and the domain display of Figure 3 are assumed to be part of the chat application or peer. For sake of simplicity, we will not consider them any further, because they have no architectural effects in our example.

Figure 4 shows an initial software design, which is constructed entirely with patterns. Implementing this design results in a peer-to-peer chat system. For our example, we first finish the development process at this point.

4 Considering Evolution through the Development Life Cycle

To support evolution, we define a corresponding evolution step for each step of the method described in Section 2. The work of O'Cinneide and Nixon [8] may seem similar to our approach. However, we do not perform refactoring to introduce design patterns into a given system. In contrast, our system is already composed of design patterns.

Our evolution method consists of several steps each providing evolution operators for the respective development phase. Changes to the original method introduced in Section 2 are indicated in bold face. The evolution operators document what the change is and how it should be carried out. This results in a corresponding set of operators for each step of our development method, e.g the addition or deletion of a domain in the context diagram in Step 1. In the following, we concentrate on illustrating those operators which are applied to our example.

1. Understand the **new** problem situation

   Evolution takes place when a change request is present. This request has different effects, depending heavily on whether or not requirements or domain knowledge are modified. In the following, we refer to new or changed requirements as evolution requirements eR and new or additional domain knowledge as aD ($aD \equiv aF \wedge aA$). The above-mentioned modifications of requirements and/or

domain knowledge make it necessary to gain an understanding of the new circumstances. This may result in a modification of the context diagram, using evolution operators. The operators relevant for this phase are:

eAD – evolution operator *add new domain*:
A new domain has to be added to the context diagram.
The eR and/or the aD introduce a new relevant domain. Relevant means that the domain is necessary to develop the specification for the machine. Usually, this implies that the new domain is directly connected to the machine domain. This domain has to be added to the context diagram. The new phenomena that occur have to be treated with **eAP** (*add new phenomenon*) or **eMP** (*modify existing phenomenon*) (described below).

eMD – evolution operator *modify existing domain*:
A domain contained in the context diagram has to be modified. Possible modifications are for example splitting or merging of domains.
In contrast to **eAD**, the eR and aD do not necessitate a new domain in this case. However, they make it necessary to modify a given domain in the context diagram. This may occur when the eR and/or the aD are extended or changed, resulting in a possible application of **eAP**.

eAP – evolution operator *add new phenomenon*:
A new phenomenon is added to an interface of the context diagram.
Whenever **eAD** is applied, it is also necessary to add new phenomena to the newly created interfaces between the added domain and the domains connected to it. It may also occur that a new phenomenon has to be added to an already existing interface (perhaps as a consequence of applying **eMD**).

eMP – evolution operator *modify existing phenomenon*:
An existing phenomenon has to be modified in the context diagram, e.g. by renaming.
The domains contained in the context diagram suffice to capture the new situation. The shared phenomena, however, have to be changed in order to handle the modified behavior derived from eR and/or aD.

In some cases, it may also occur that neither domains nor shared phenomena are newly introduced. Then, no changes to the context diagram are necessary in Step 1, but the new requirements/domain knowledge may require changes in later steps. A reason for this is that at this stage, only the static aspects of the system and not the dynamic aspects are taken into account. The resulting context diagram now represents the new overall problem situation.

2. Decompose overall problem into simple subproblems, **and adapt existing ones**

It is necessary to investigate the existing subproblems, applying evolution operators as necessary. The eR are the driving force behind this investigation, as they determine whether or not it is necessary to create a new subproblem or to adapt an existing one. Examples of evolution operators for this step are:

eIR – evolution operator *incorporate eR into a given subproblem*:
New domains and associated shared phenomena may be added to an existing

problem diagram. This is possible if the eR references at most the same domains as the given subproblem. Conflicting requirements may occur at this point. However, resolving such conflicts will not be addressed here.

eCS – evolution operator *create new subproblem*:
Either the eR is assigned to a given subproblem, but the resulting subproblem then gets too complex. Hence, it is necessary to split the subproblem into smaller subproblems.
Or the eR cannot be assigned to a given subproblem, and a new subproblem has to be created.

For the next steps, only newly introduced and adapted subproblems have to be taken into further consideration, as only these will undergo changes. The subproblems which have not been addressed in this step can be disregarded for now. They will only become relevant again in later steps, when the solutions of the subproblems are composed to the overall solution.

3. Fit subproblems to (variants of) problem frames and **adjust problem frame instances**

   The operators for this step include:

   eFF – evolution operator *fit to a problem frame*:
   Each newly introduced subproblem is fitted into a problem frame by instantiating it according to the general procedure of Section 2.

   eCF – evolution operator *choose different problem frame*:
   For each adapted subproblem, it is checked whether its underlying problem frame is still valid, or whether another problem frame is now more appropriate. The corresponding problem frame is then instantiated accordingly.

4. **Modify** and instantiate corresponding architectural and design patterns

   Comparable to evolution Step 2 driven by eR, this step is guided by aD. Evolution operators applicable in this step are:

   eAA – evolution operator *adjust given architecture*:
   Adapted subproblems, which still fit into already instantiated problem frames, can usually be incorporated into the given architecture without difficulties.

   eCA – evolution operator *choose different architectural style or design pattern*:
   New subproblems or subproblems that fit to different problem frames than before lead to a new investigation of the solution.
   This investigation may result in a (re-)assignment of existing subproblems to new architectural styles or patterns.
   Furthermore, aD can cause a change in the *problem/solution-pattern relation*, resulting in a reallocation of subproblem elements to corresponding parts of solution patterns via new analogies. For instance, this fact distinguishes evolution scenario I from evolution scenario II (see Sections 5 and 6).

In the subsequent sections, we illustrate the usage of these evolution operators by two evolution scenarios for our chat application.

SM	*A text-message-based communication platform shall be developed, which allows multi-user communication via private I/O-devices.*
R1	Users can phrase text messages, which are shown on their private graphical displays.
R2	Users send their phrased text messages to participate in the chat.
R3	Sending text messages changes the chat represented on the users' graphical displays.
eR5	**Users want to chat via long distances. Therefore it is necessary to pass the data from the local network to the wide access network (and vice-versa).**
aF1	Users communicate in a local network**, or via a wide area access network**.
A1	Users follow the course of the chat on their private graphical display.

Table 2. Changed Requirements and Domain Knowledge for Evolution Scenario I

5 Evolution Scenario I

The starting point for this first software evolution scenario is a chat system as described in Section 3 for local communication (cf. F1 in Table 1), for example via a bluetooth device as an implementation of the network domain.

A limitation of such a chat application is that users are restricted to the range of their bluetooth devices. This limitation has to be removed now. An extended fact **a**F1 about the application domain is introduced, see Table 2: Users communicate [...] **via a wide area access network**.

However, there is also a *constraint* restricting the evolution procedure: *The structure of the original application should be maintained*. This constraint stresses the maximal possible reuse of artifacts of the existing system. Therefore, it will be necessary to maintain the existing architecture in its original form as far as possible. We now follow the procedure described in Section 4 for evolving the given chat system:

1. Understand the new problem situation

   Analyzing the change of domain knowledge results in an additional requirement, which is added to Table 2 as **eR5**.
   It is not necessary to add a new domain into the context diagram. The set of existing phenomena suffices, as well. Therefore, it is not necessary to make any changes to the existing context diagram. It still looks as shown in Figure 1.

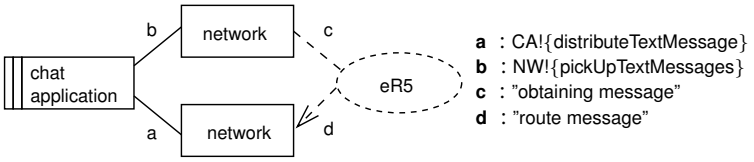

Fig. 5. "Message Dispatcher" subproblem (instance of a variant of the *transformation* PF)

2. Decompose overall problem into simple subproblems and adapt existing ones

   We apply the operator **eCS** (*create new subproblem*) of Section 4 and create a new subproblem. The formerly used network is not able to provide a wide area access.

Taking the above constraint into consideration, as well, we obtain the following new subproblem represented in Figure 5. We need to transfer the data from our bluetooth network to another network, which will deal with passing the data to or receiving the data from the wide area access network. The other subproblems remain unchanged.

3. Fit subproblems to problem frames and adjust problem frame instances

   As we have created a new subproblem, the evolution operator **eFF** (*fit to a problem frame*) described in Section 4 has to be applied. Accordingly, the new subproblem becomes an instance of a transformation problem frame variant.

4. Modify and instantiate corresponding architectural and design patterns

   By having a closer look at the new problem diagram in Figure 5, we see that what is performed by this subproblem can be characterized as a kind of dispatching. Now the evolution operators **eAA** (*adjust given architecture*) and **eCA** (*choose different architectural style or design pattern*) have to be considered. The operator **eAA** preserves the *Forwarder-Receiver* architecture for the subproblems in Figures 2 and 3. These problem diagrams stay untouched, and so does their corresponding architecture. To the newly created subproblem, we apply the operator **eCA**. As we know that we need a dispatcher, this leads us to the pattern of *Client-Dispatcher-Server* [1], because a dispatcher is responsible for establishing a (wide area) connection between two parties. Another alternative could be the design pattern *Proxy*. However, we choose the first pattern, namely *Client-Dispatcher-Server*, to illustrate the evolution in this scenario. To satisfy the accompanying evolution constraint to reuse as many development artifacts as possible, we attach the *Client-Dispatcher-Server* to the already applied *Forwarder-Receiver* pattern, resulting in a hybrid design pattern. Here, we follow in general the *pattern-oriented analysis and design* (POAD) approach [15]. As shown in Figure 6, the new solution pattern for the added subproblem in Figure 5 can therefore simply be "plugged together" on the conceptual level with the existing one in Figure 4.
   The connection between the two solution patterns namely *Forwarder-Receiver* and *Client-Dispatcher-Server* is realized through dependencies. We want to maintain the original patterns as much as possible. The class Forwarder and Client share the responsibility for sending some content or requests. Therefore, we can reuse Forwarder for implementing sendRequest of class Client. The same holds for the class Receiver, which is responsible for realizing the Server operation receiveRequest.
   The chat application and network class derived from our subproblem descriptions are related to the combined solution patterns via generalization/specialization relations. Where chat application takes the role of a peer that provides services such as sendTextMessage, and network takes the role of forwarder and receiver for handling the reception of messages by pickUpTextMessage and their delivery through distributeTextMessage. In its role as a Dispatcher, chat application uses its reference to network for controlling its message handeling, respectively.
   Interesting is the role of the network, because it is part of all three subproblems. For the subproblems in Figures 2 and 3, the network is responsible for providing the Forwarder/Receiver functionality. For the subproblem in Figure 5, it implements Dispatcher operations. For the *Dispatcher* the network connects the different chat

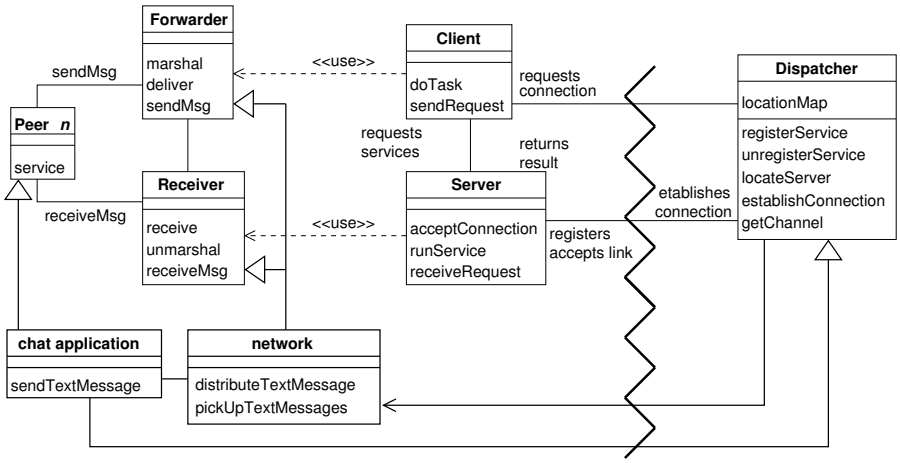

Fig. 6. Evolved Class Diagram of the Chat Application (Hybrid Style)

peers via long distances, whereas each peer is a *Client* as well as a *Server* communicating via a *Forwarder-Receiver* mechanism.

It is clearly visible that in this first evolution scenario the modification of **a**F1 and the addition of **eR5** resulted in the creation of a new subproblem. Because of this new subproblem, it was necessary to make a new design decision. Considering the given constraint, the decision leads us to an extention of the existing *Forwarder-Receiver* architecture to a hybrid *Forwarder-Receiver/Client-Dispatcher-Server* style.

6 Evolution Scenario II

The second evolution scenario is based on the results of Section 5. With the current chat application it is possible to communicate with other users via bluetooth or a network providing wide area access. The devices used so far are general purpose computers. Mobile devices such as portable phones or personal digital assistants (PDA) are not supported yet. This describes the limitation that we remove in this second evolution scenario: The usage of portable devices should be possible, as well. Once more the evolution steps described in Section 4 are applied:

1. Understand the new problem situation

   New domain knowledge is added, described by the additional fact **aF2** in Table 3: **The devices used are general purpose computers as well as portable phones and PDAs**. This new fact describes hardware constraints referring to the machine domain. No additional requirements are necessary. Therefore, the context diagram remains unchanged, even though new domain knowledge has been introduced. The reason is that **aF2** influences internal characteristics of the machine, and not its behavior. These characteristics, however, cannot be described by a context diagram.

SM	*A text-message-based communication platform shall be developed, which allows multi-user communication via private I/O-devices.*
R1	Users can phrase text messages, which are shown on their private graphical displays.
R2	Users send their phrased text messages to participate in the chat.
R3	Sending text messages changes the chat represented on the users' graphical displays.
eR5	Users want to chat via long distances. Therefore it is necessary to pass the data from the local network to the wide access network (and vice-versa).
aF1	Users communicate in a local network, or via a wide area access network.
aF2	**The devices used are general purpose computers as well as portable phones and personal digital assistants (PDAs).**
A1	Users follow the course of the chat on their private graphical display.

Table 3. Changed Requirements and Domain Knowledge for Evolution Scenario II

2. Decompose overall problem into simple subproblems and adapt existing ones

   The distribution of the overall problem situation into subproblems stays unchanged, because the requirements do not change.

3. Fit subproblems to problem frames, and adjust problem frames instances

   No changes have to be performed in this step, because no changes were performed in the previous step.

4. Modify and instantiate corresponding architectural and design patterns

   This step is the nontrivial one in this scenario, because here the effect of the newly introduced domain knowledge becomes visible. Fact **aF2** influences the decision which pattern to select in the solution space. Mobile devices such as PDAs or mobile phones do not possess the same resources as general purpose computers do. The new fact thus imposes a constraint on the selection of design patterns and architectural styles. Here, it leads to a re-design of the present architecture by means of the evolution operator **eCA**.

   The new domain knowledge enforces a reorganization of the given subproblems, resulting in a different choice of architectural style or design pattern out of the related solution class. For our example, the three subproblems are matched with the *Client-Dispatcher-Server* pattern only (cf. Figure 7).

   The former forwarder and receiver components are merged with the client and server components. The chat application now consists of two parts namely, a server and a client part. The functionalities formerly represented by the peer are now partially realized by the server and client classes, respectively.

In the first evolution scenario, we had to deal with a constraint. This constraint is still present in the second scenario, however, in a weakened form: all the subproblems not involved in the data transmission/reception remain unchanged, and the development documents related to them can be reused as is. In this second scenario, adding domain knowledge in form of **aF2** results in a complete restructuring of the architecture. The reason is that the new domain knowledge leads to a different matching of subproblems to architectural patterns, resulting in a new, more appropriate and simpler architecture.

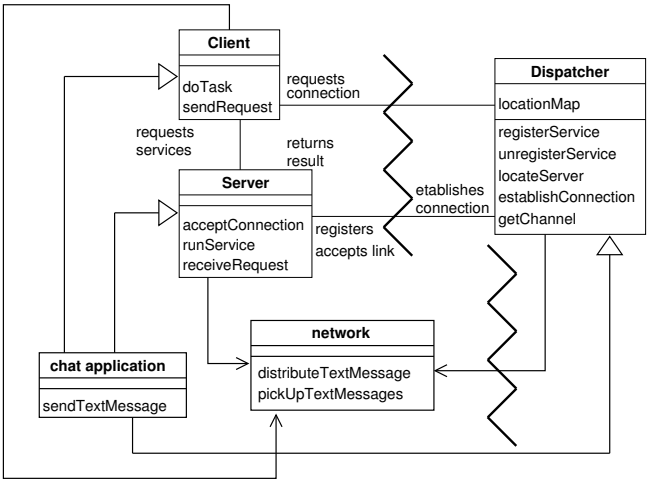

Fig. 7. Resulting architectural design based on *Client-Dispatcher-Server*

7 Conclusion and Future Work

We have introduced a pattern-based development method, incorporating evolution in each step, and driven by evolution operators. With this method, it is possible to perform software evolution systematically whenever new requirements or changes in the application environment occur.

Through a chat application example, we have shown the usage of our method. We illustrated how a system evolved from a straight peer-to-peer architecture via a hybrid architecture to a client-dispatcher-server system. This evolution is achieved by applying evolution operators. They allow for identifiying those development documents, which have to undergo change. Thus, they provide guidance for performing the necessary modifications in phases to come.

This approach enables us to perform a systematic rework on the affected development documents by means of patterns. We have shown that it is possible to link the artifacts of the analysis phase to the artifacts of the design phase. Therefore, changes in the analysis help to perform an analogous procedure in the design phase.

Our method does not intend to replace existing methods or notations. It rather extends them by a goal-oriented, pattern-based approach resulting in coherent and precise specifications. The method does not postulate a dogmatic approach, always resulting in exactly one unique solution. We intend to constrain the possible solutions (design space) to provide a small, most promissing set of patterns useful for solving the problem.

Also non-functional requirements can be treated with our approach: If these quality aspects are specified in the problem frames (such as *HCI*Frames [14] or using Security Problem Frames [6]), they will lead to solutions that address these non-functional issues, see evolution scenario one in Section 5. Additionally, non-functional aspects

that are manifested in domain knowledge can be covered, see evolution scenario two in Section 6.

In summary, the advantages of our approach are the following:

- Our method is pattern-to-pattern, integrating evolution. Hence, it has all assets that come with the use of patterns, in particular, reuse of established analysis and design knowledge.
- Our evolution operators provide guidance concerning pattern selection and transformation. They help to adapt the development problem and to find new solutions if necessary.
- Non-functional requirements can be treated, as well.

Currently, we are working on a formalization of the problem frames. For that purpose, we are building a formal metamodel using the formal specification language Object-Z [12]. The graphical representation of the formalization is given through UML class diagrams [13]. The metamodel is equipped with integrity conditions to ensure the validity of a frame with respect to the metamodel. It is planned to create an Eclipse [4] plug-in for this metamodel. The plug-in will work as an editor to create valid frame diagrams in the context of our metamodel. The metamodel will also serve as a basis for investigating which of the evolution operators can be formalized and incorporated into the metamodel. In a further step, the plug-in will be extended to allow for an automation of the identified and formalized evolution operators presented here.

In the future, we also plan to analyze the effects of domain knowledge within the evolution process in more depth. Additionally, we plan to examine the impact of the change in the architectural structures to later documents of the software life cycle, especially considering the source code. Furthermore, we intend to put stronger emphasis on distinguishing internal and external quality aspects and how they are successfully covered by our method. It is also planed to have a detailed look at the decompositions and compositions of the subproblems with respect to the architecture, which would also contribute to investigate the scalability of problem frames. We further will investigate which problem frames should be related to which architectural design patterns or architectural styles for completing our method.

References

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

[2] C. Choppy, D. Hatebur, and M. Heisel. Architectural Patterns for Problem Frames. *IEE Proceedings - Software*, 152(4):198–208, 2005.

[3] C. Choppy, D. Hatebur, and M. Heisel. Component composition through architectural patterns for problem frames. In *Proc. XIII Asia Pacific Software Engineering Conference*, pages 27–34. IEEE Computer Society, 2006.

[4] T. E. Foundation. Eclipse - an open development platform, 2007. `http://www.eclipse.org`.

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

[6] D. Hatebur, M. Heisel, and H. Schmidt. Security engineering using problem frames. In *Proc. of the Int. Conference on Emerging Trends in Information and Communication Security (ETRICS)*, volume 3995/2006, pages 238–253. Springer Berlin / Heidelberg, 2006.

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

[8] M. O'Cinneide and P. Nixon. Automated Software Evolution Towards Design Patterns, 2001. `http://citeseer.ist.psu.edu/671812.html`.

[9] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 2001.

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

[11] H. Schmidt and I. Wentzlaff. Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. In *Proceedings of the 3rd European Workshop on Software Architectures (EWSA'06)*, Nantes, France, 2006. Springer.

[12] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[13] UML Revision Task Force. *OMG Unified Modeling Language: Superstructure*, 2007. `http://www.omg.org`.

[14] I. Wentzlaff and M. Specker. Pattern-based Development of User-Friendly Web Applications. In *Workshop Proceedings of the 6th International Conference on Web Engineering (ICWE)*, New York, NY, USA, 2006. ACM Press.

[15] S. M. Yacoub and H. H. Ammar. *Pattern-oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley, 2003.