Contributions of Pattern Languages to Framework-Based Development in Layered Architectures

Luciano Gerber Karin Becker
Pontifícia Universidade Católica do Rio Grande do Sul – PUC-RS
Faculdade de Informática – Mestrado em Informática
[lgerber, kbecker]@inf.pucrs.br – http://www.inf.pucrs.br/~[lgerber, kbecker]

Abstract

Object-Oriented frameworks are an important technology to promote reuse in software development. However, there are still some problems to face in framework research, such as integration and documentation. In this paper we discuss how pattern languages could help to solve these problems, particularly in framework-based layered software architectures, providing satisfactory solutions to framework integration and improving documentation. We suggest the establishment of standard framework interfaces and adequate use of white and black-box approaches to facilitate framework integration between architectural layers. We also suggest the use of visual graphs with typed links between patterns to facilitate the utilization of a pattern language. These elements are all illustrated by a framework-based layered software architecture to Development of Decision Support Systems, used as a case study.

1. Introduction

Object-oriented (OO) frameworks have been considered an important element to achieve reuse in software development. An *object-oriented framework* [FAY97][GAM94][JOH97] can be described as a "reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact" [JOH97]. It is believed that the use of frameworks brings several benefits to software development, such as reusability and extensibility. However, there are still many open problems in framework-based development [MAT99][FAY97][MAT98]. In this paper, we focus on two of them, namely the integration of individual frameworks to compose applications, and framework documentation

Framework integration is a crucial issue in software development. Nowadays, it is very common to construct applications by reusing and integrating existing frameworks playing different roles in an application, such as interface, database, networking and middleware, and domain-oriented elements. Unfortunately, most frameworks are produced having in mind only individual reuse (extension and customization), not integration with other frameworks. In [MAT99], problems, its causes and possible solutions of framework integration are highlighted. A possible solution to that problem is to predict families of frameworks that one could be connected, establishing standard interfaces to simplify framework integration. We consider that issue in the context of layered software architectures, generalized by the pattern *Layers* [BUS96]. In this specific context, frameworks standard interfaces representing invariant aspects of each layer could be established to make the connection process easier, such that applications could be produced by integrating frameworks of different layers whose instances communicate and collaborate to provide the overall application functionality

Documentation is another key issue in framework-based development. Its primary role is the provision of easy and complete instructions to its user on how to use and adapt a framework. In addition, it is also useful to document a framework internal structure and its main design principles, in such a way that both

abstractors (framework developers) and elaborators (framework users) can learn and possibly reuse some of the design heuristics and techniques applied. Many works address the use of *pattern languages* [APP97][GAM94][BUS96] as an elegant way to document framework construction and utilization, following the growing movement on the software development community in using pattern languages to register developer's knowledge. A *pattern* [APP97][GAM94][BUS96] is an abstract and structured description of a satisfactory solution to a problem that occurs repeatedly in a context, accordingly to some *forces*, and a *pattern language* is a group of interconnected patterns that collaborate to solve a complex problem.

The purpose of this paper is to present and discuss the contributions pattern languages could bring to framework integration-based development in the specific context of layered architectures. First, a pattern language can be used to describe satisfactory solutions to recurrent problems of frameworks integration, contributing a great deal on the easiness for frameworks connection. Second, it serves as a systematic, well-structured and easy-to-use documentation, which can be useful for both abstractors and elaborators.

The paper discusses the properties of pattern languages and how they can be useful to documentation. It also introduces the use of *typed-visual graphs*, composed of equally typed nodes and links, to improve a pattern language readability and easiness of use. Typed-nodes are used to separate and encapsulate solution variants to a same problem in different patterns classified respectively as *solution-oriented* and *problem-oriented patterns*. This distinction helps on the choice of a specific solution, showing the conditions where a given pattern is more adapted to the problem at hand than other available ones, as well as on the evolution of a pattern language. As in many other works [COP95][MAR98][VLI96], we use visual graphs composed of patterns (nodes) and their relationships (edges) to provide an overview of the language, which acts like a map that guides the application of patterns. To make the navigation between patterns easier, we propose the use of *typed links* to denote the different types of semantics that relates patterns. The motivation of the use of typed links is the same as in modeling notations (e.g. UML). Finally, in order not to jeopardize the readability of the resulting pattern language graph, we propose to organize patterns in two types of graphs referred to as *Problem-oriented* and *Solution-oriented*. The former details all the aspects (subproblems) of the overall problem addressed by the language, using problem-oriented patterns. The latter organizes the possible solutions for each identified aspect, and relates problem-oriented patterns to corresponding solution-oriented ones. These properties are illustrated by a case study, which addresses a layered architecture to framework-development of Decision Support Systems (DSS) [BEC93][BEC98].

The rest of this paper is structured as follows. Section 2 addresses the aspects of layered architectures that are relevant to this work. Frameworks, patterns and pattern languages, are discussed in sections 2, 3 and 4, respectively. The case study is described in Section 5, and the identified problems are analyzed in Section 6. Section 7 addresses the contributions of pattern languages in framework-based development in the context of layered architecture. Finally, in the Section 8 we discuss the conclusions and future work.

2. Layered Software Architectures

Layered software architectures suggest the structuring of applications on several layers addressing different responsibilities, each one in a different abstraction level. Well-known examples of this kind of software architecture are the OSI network 7-layer model [TAN92] (Figure 1**),** the Gebos System [BAU97] and *n*-tiers information system architectures [FOW97].

Typically, layered architectures have two common properties: a) a same concept is provided with different representations,

possibly one for each layer; and b) the various representations of a same concept handle only partial aspects of the concept overall responsibility or functionally, according to the meaning assigned to the layer in the architecture. In terms of behavior, messages are initially sent and handled by an abstraction in the top layer. Then, each layer act like a filter, which receives and responds messages, executing its own responsibilities and delegating messages to the next layer corresponding to the aspects the layer does not handle. Responses to messages are sent in the opposite way, flowing from low to high layers. Therefore, there is always a defined protocol of communication between the layers.

In addition, the abstractions of distinct layers that represent a same concept must be synchronized, such that a change in one abstraction must be reflected in all corresponding ones. This frequently implies the use a notification or event-based mechanism between layers.

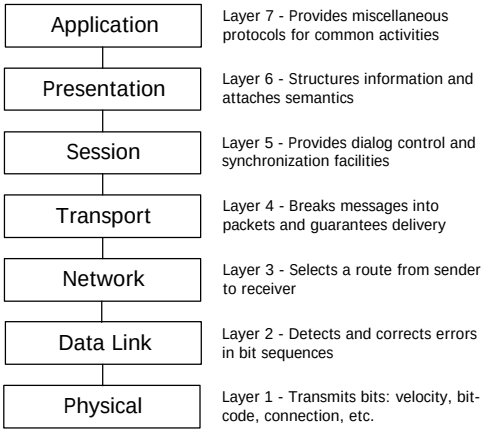


Figure 1 – OSI 7-Layer Model (from [BUS96])

The use of layered software architectures provides some benefits, such as the increase of modularity, separation of distinct responsibilities that evolve separately, reuse of components of each layer in different applications, substitution of components in a specific layer without impacting other layers, among others. Some drawbacks are performance detriment due to state synchronization between layers, and the intensive flow of messages and data between layers when high-level responsibilities are too dependent on low-level layers.

3. Frameworks

OO frameworks represent semi-complete applications or generic sub-systems that can be adapted and integrated for construction of specific applications [MAT98][FAY97]. Framework-based development leads to two different roles in software development, where abstractors create frameworks that will be reused and adapted by elaborators [MAT98][BCK94]. Usually a framework defines its stable features, called *frozen-spots*, which are invariant in all possible framework uses, and leaves open its variable aspects, called *hot-spots*, which represents features that must be customized in each specific application [PRE95].

Using a framework in a specific application involves a customization process that consists in filling the hot-spots and, possibly, extending the framework to add functionality. According to the intended customization process, frameworks are classified as *white-box* and *black-box* [FAY97]. *White-box* frameworks rely on inheritance and overwriting mechanisms, whereas *black-box* ones are adapted by composing objects that conform to hot-spots interfaces.

Framework development is always influenced by two primal contradictory issues: a framework should be reusable (*reusability*), as well as easy to customize (*adaptability*). To predict frozen and hot-spots may increase *adaptability*, allowing different aspects to vary and evolve independently. This in turn may decrease *reusability*, since a developer must carefully anticipate future uses of a framework in order not to limit its reuse spectrum. In addition, white-box frameworks are considered harder to adapt than black-box ones, since the use of inheritance forces the elaborator to understand internal details, and also can lead to an explosion of subclasses. Black-box

frameworks are easier to adapt, since one only needs to know the hot-spots interface to compose objects. On the other hand, they are harder to develop, since one has to design carefully the hot-spots interface.

It is claimed that frameworks bring many benefits to software development, such as improvement of software reusability and extensibility. However, there are some open problems to solve in framework-based development [FAY97][MAT98] [MAT99], two of them addressed in this paper: frameworks integration and documentation.

Framework-based development usually leads to construction of application by reusing and integrating different frameworks playing different roles in a specific application, addressing aspects (e.g. interface, database, middleware, domain-oriented elements). Unfortunately, most frameworks are produced having in mind only in individual reuse (extension and customization), not in integration with other frameworks. Framework integration problems are extensively discussed in [MAT99]. We believe that it is important to predict families of frameworks that can be connected, establishing standard interfaces to ease framework integration, avoiding modification of existing frameworks in order to connect them. Also, the harmonization of forces such as reusability and adaptability, and the adequate use of white-box and black-box approaches, allows the production of more reusable and easy-to-connect frameworks. By reducing our context to layered architectures, the integration of frameworks can be simplified by considering that frameworks belong to specific layers, for which a meaning and overall role and responsibility has been determined by the architecture. It is then possible to capture the particular semantic aspects of each layer and represent them as standard interfaces to facilitate framework connection.

The documentation of a framework must communicate its intent, the problem it addresses, and also show to the elaborator which are the frozen and hot-spots of the framework and how to fill these hot-spots. If elaborators do not clearly understand how to use a framework and the rationale behind design decisions, they may feel discouraged to reuse it. This includes a description of the design decisions, internal aspects of a framework design, and possibly a description of the design patterns used. The documentation of the internal structure and design decisions of a framework are important to abstractors, who may extend an existing framework and reuse the same techniques and heuristics used to create other frameworks in different contexts.

Recently, pattern languages have been considered as an elegant way to document framework design and utilization. In [JOH92], for instance, a pattern language is used to describe how to use the HotDraw framework for the construction of semantic graphic editors. Some pattern languages have been published to document framework design, such as [BRU97]. A pattern language may provide satisfactory solutions to problems of framework development, as well as to show the “why” of the solutions.

4. Features of Patterns and Pattern Languages

Patterns record people's knowledge and experience. Patterns can help expert developers to document and reuse knowledge, as well to disseminate this knowledge to novice developers. A pattern contains several descriptive elements, and there are many different formats to write patterns in the literature. The essential pattern elements [APP97][MES98] are *name*, that clearly and uniquely identifies a pattern, the description of a recurrent *problem*, a *context* on which the problem occurs repeatedly, a group of *forces* acting over the problem pressuring the solution, and the *solution* itself, which solves the problem by establishing a compromise among the forces. A force in software development could be considered as a criterion software developers use to justify designs (e.g. efficiency, reusability, extensibility). Other elements are important, but optional, such as

solution rationale (how the solution resolves or balances the forces of the problem), *examples* (concrete forms of the abstract pattern description), r*esulting context* (context after applying a pattern), *related patterns*, etc.

A pattern language can be seen as a "super-pattern" [APP97], which is split into several individual and connected patterns that collaborate to solve an overall, complex problem. A pattern language can be used as a "how-to" guide, i.e., step by step, starting from the more generic patterns to more specific ones, using the paths created by the connections established between the patterns. It can also be used as a reference guide, where the user (reader) searches for specific patterns, usually corresponding to specific sub-problems.

The structure of a pattern language could be regarded as a graph, where patterns are nodes and relationships between patterns are edges. The paths formed by the relationships determine the order of application of patterns, which usually flows from more generic (or high-level) patterns to more specific (or low-level) ones. Some pattern languages include a graph providing a general view of the language, acting like a map of use [DYS98], or summaries [MES98].

In the textual description of patterns, the relationships between patterns are expressed in sections like *context*, *related patterns*, *solution* or *resulting context*. For instance, the *context* section is frequently used to express that the pattern has another one(s) as pre-requisite, but other relationships can be specified as well, as pattern refinement. Though one can notice that there are different types of relationships between patterns, there is no standard types acknowledged by the pattern community. Some works try to categorize different types of relationships, like [NOB97][ZIM94]. [NOB97] identifies three primary relationship types:

- **Uses**: a pattern *uses* other ones that complete it and solve problems raised by the former. For instance, *Abstract Factory* uses *Factory Method* to instantiate collections of families of objects.
- **Refines**: a pattern *refines* another one when it addresses a problem that is a specialization of the problem addressed by the refined pattern. This is a kind of inheritance relationship. For instance, the design pattern *Factory Method* [GAM94] refines the metapattern *Unification* [PRE95];
- **Conflicts**: a pattern *conflicts* with another one when both provide exclusive solutions to a same or similar problem. For instance, the design patterns *Prototype* and *Factory Method* [GAM94] conflict because they provide mutually exclusive solutions to the problem of object instantiation.

5. Case Study: A Layered Framework-Based Architecture for DSS Development

To present and discuss the benefits the use of pattern languages could bring to framework-based development in layered architectures, we use a layered architecture for DSS development [BEC93][BEC98] as a case study. It addresses a specific class of DSS known as *model-oriented DSS* [SPR80].

This layered architecture is a guide to DSS developers, which suggests the construction of specific DSS through reuse and integration of frameworks organized in four layers.

Figure 2 presents the four layers in which frameworks are organized. Specific DSS are structured by frameworks adaptation and integration. Each layer provides a proper representation (i.e. according to the layer’s goal) of a model for a decision problem, in different abstraction levels. Layers represent *Semantics* and *Interface* aspects, a common separation in the development of interactive applications.

Two layers constitute the semantics of the architecture, namely *Decision Situation* and *Resolution*. The *Decision Situation Layer* is responsible for representing decision problems using domain-oriented concepts, composed of

decision-maker-oriented models of problems. Decision situation frameworks represent classes of decision problems. For example, the Capital Budgeting class of problems [BRI91] involves decisions about investments whose returns are expected to extend beyond a year. Models for this class of decision problems can be formulated in terms of concepts such as lifetime of the investment project, time-schedules of incomes and expenses, cash flow, profitability criteria, cost of capital, etc. The *Resolution Layer* is responsible for handling a model representation that can be solved (i.e. executed), using algebraic or logical resolution methods. It is composed of resolution-oriented models of decision problems. A Resolution framework could represent a resolution method, such as *linear programming*, *dynamic programming* and *spreadsheets capabilities.*

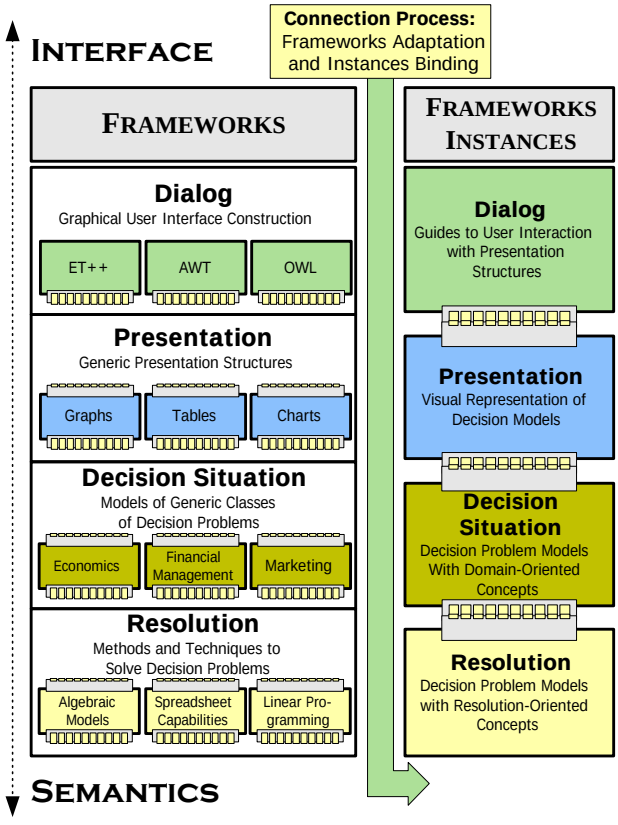


Figure 2 – Layers of the DSS Architecture

Two layers compose the interface, namely *Presentation* and *Dialog*. The *Presentation Layer* provides presentation structures for the manipulation of decision problems, with which decision-makers are familiar. Presentation frameworks represent structures such as *table*, *graph*, *chart*, and *form*, and they are used to provide domain concepts with a visual representation. For instance, a *table* could represent a capital budgeting model, where columns relate to the concept of time and rows represent various concepts like cash flow and profitability criteria.

Dialog is the top layer. It provides elements for user interaction with the DSS, which guide a decision-maker in the modeling and resolution of decision problems, through the interacting with presentation structures. Nowadays it is usually built by customization of graphical components available in GUI frameworks, such as AWT, Swing, MFC, ET++, etc.

The process of specific DSS development is based on the integration of frameworks from different layers. It is necessary to adapt the classes of the reused frameworks, add application-specific behavior, bind and configure framework instances, establishing a communication channel of messages between objects of different layers. Figure 3 shows the dynamic behavior of a specific DSS, according to different scenarios created by different types of messages sent by a DSS user to Dialog objets.

Generally speaking, user interaction falls into three different types of scenarios. First, one can send messages related only to the graphical appearance of presentation elements (e.g. change font size, rezise), and these messages must be forwarded to the presentation layer, which is responsible for handling them. Second, one can send messages related to decision problem modeling (e.g. create a profitability criterion). This type of message must be forwarded to all layers, since they all represent a same decision model in different levels of abstraction. Therefore, each layer responds to them accordingly, and forwards the message to the next layer, until the resolution layer is

reached. Finally, a user can send messages related exclusively to resolution issues (e.g. calculate NPV criterion), and these messages are propagated to the resolution layer, which is responsible for executing the model.

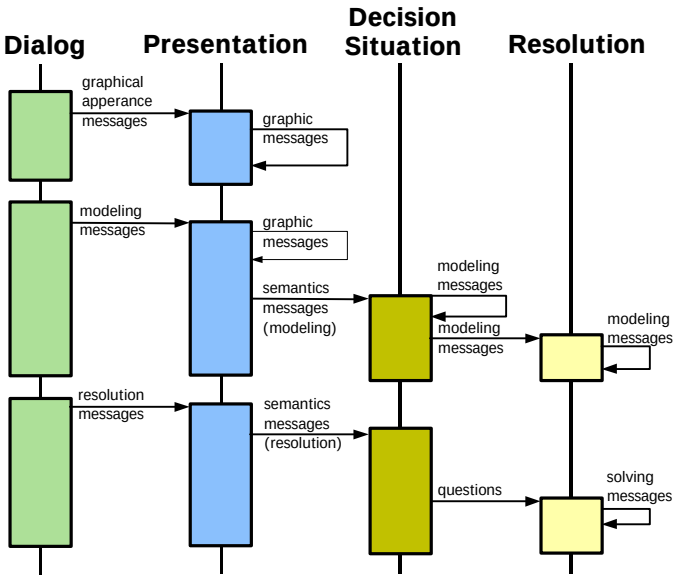


Figure 3 – Collaborations between Framework Instances of Different Layers in a specific DSS

For instance, considering the capital budgeting example, a request to include a profitability criteria such as NPV (Net Present Value) causes the instantiation of a *row* object in a instance of *Table* presentation framework, a NPV object in a instance of *Capital Budgeting* decision situation framework, and a *decision variable* in a instance of *Linear Programming* resolution framework. These connections between these new objects enable messages to flow between layers.

6. Problems Experienced Using The Architecture

We have identified some problems and difficulties on the use of the DSS architecture [GER98][GER99], as summarized below. These are mostly related to lack of documentation to support framework-based development. It is important to highlight that these problems, its causes and possible solutions, though specific to this DSS architecture, can be easily generalized to any layered architecture based on framework integration.

- **Lack of Standard Interfaces**

The process of framework integration was in practice hard to perform, since an elaborator had many activities to do in order to integrate them. One of the analyzed causes for this problem was the lack of standard interfaces to integrate the frameworks.

- **Overuse of the White-Box Approach**

Regarding difficulties of framework integration, another cause we have identified was an overuse of white-box approach to connect frameworks. As pointed out in section 3, white-box frameworks usually are more reusable but more difficult to adapt than black-box ones.

- **Lack of a Generic and Systematic Guide**

The DSS architecture was sometimes considered difficult to understand, particularly with regard to the underlying integration process. First, the architecture should be described more systematically, where a developer could produce a specific DSS in a step-by-step manner. In the available documentation [BEC98], detailed methods and techniques used to integrate frameworks were described largely in concrete forms within the case studies developed in. In practice, an abstractor had to observe the techniques used, abstract them and finally apply them in his own project.

- **Lack of Rationale for Solutions**

Developers are reluctant to use something they do not fully understand, as discussed in Section 4. It is important to provide the rationale for the design solutions of framework integration, which was seldom made in the DSS architecture.

7. Use of Pattern Languages for Framework-Based Development in Layered Architectures

We believe that the use of a pattern language can bring two contributions to the problem of frameworks integration, in the limited scope of layered architectures. First, it can be used to describe satisfactory solutions to recurrent problems of frameworks integration. Second, a pattern language servers as a systematic, well-structured and easy-to-use documentation, which can be useful for both abstractors and elaborators.

In this section we discuss how some of these problems could be solved with the use of a pattern language. We highlight desired features of such pattern languages and describe the constructs we propose to enhance pattern language writing and use. These issues are illustrated by the case study.

7.1. General Features

The features of pattern languages we believe could help to solve documentation problems described in the previous section are:

- Abstraction of recurrent problems of *framework integration* in layered architectures. In the original architecture, most problems were described in concrete terms within the development of case studies.
- Illustration of problems and solutions addressed by the patterns by a *running example*, which is extensively treated throughout the whole language.
- Separation of *independent sub-problems* composing the generic and complex problem of framework integration, taking a divide-and-conquer approach to solve it.
- Possibility of use as a *step by step guide*, where the user starts with the more generic pattern and applies other ones following the established links between them. This allows the use of a systematic approach to deal with the complex problem of framework integration in a layered architecture.
- Possibility of use as a *direct reference guide*, in which a user can choose a specific pattern that corresponds to some selection criteria, e.g. addressed problem. This kind of use could be facilitate by a pattern language index [MES98], such as those in [GAM94][BUS96].
- *Justification of design decisions* taken to create solutions for frameworks integration problems, describing their rationale and how their harmonize the forces.

7.2. Problem and Solution-Oriented Patterns

Two essential goals must be attained by any solution that addresses problems of framework integration, which were concretely described in the previous section. First, it is important to provide *standard interfaces to integrate frameworks*. This is somehow possible to do in the specific context of layered architectures, since we can identify and represent layer-specific frozen and hot-spots, by capturing the semantics of each layer.

The *adequate use of black and white-box approaches* is also essential, since it results in reusable and easy-to-customize frameworks, balancing in this way the reusability and adaptability forces. The use of white or black-box approaches is influenced by several factors. For instance, when an abstractor does not have a deep knowledge of the problem domain, it could be dangerous to use a black-box approach since it is difficult to predict a great spectrum of framework usage. In some situations, a white-box approach is more suited than a black-box one, and vice-versa.

Therefore, we suggest the provision of solution variants to a same problem of framework integration, reflecting for instance, white and black-box approaches. This allows an abstractor to choose a solution that fits better in a particular context. We propose the encapsulation of solution variants in separated patterns, called *solution-oriented patterns*, which are refinements of a *problem-oriented pattern*. The latter describes a recurrent

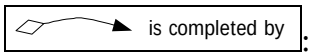
problem of framework integration without committing to any given solution. We believe this separation additionally allows a smooth evolution of a pattern language, since new satisfactory solutions to a same problems adapted to specific contexts can be included any time without causing impact in the pattern language structure. Section 7.4 shows the visual representation assigned to problem and solution-oriented patterns in pattern language graphs.

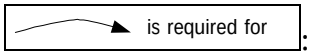
7.3. Typed Graphs and Typed Links

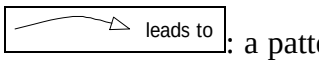
The contributions of visual graphs for pattern languages representation has been highlighted by works such as [COP95][MAR98][VLI96]. Visual graphs are composed of patterns (nodes) and their relationships (edges), and provide an overview of a pattern language, which acts as a map guiding patterns application. It can be used to guide the navigation among patterns, which involves the choice of the appropriate patterns according to the problem at hand, as well as the order in which they should be applied.

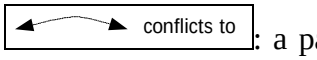
In order not to jeopardize the readability of the resulting pattern language graph, we propose to organize patterns in two types of graphs referred to as *Problem-oriented* and *Solution-oriented*. The former details all the aspects (subproblems) of the overall problem addressed by the language, using problem-oriented patterns. The latter organizes the possible solutions for each identified aspect, and relates problem-oriented patterns to corresponding solution-oriented ones. Figure 5 and Figure 6 presents, respectively, the problem-oriented and solution-oriented graphs for the pattern language created specially to guide framework integration in the case study.

To make navigation between patterns easier, we additionally propose the use of *typed links* to denote the semantically different types of relationships between patterns. The motivation for the use of typed links is the same as in modeling notations (e.g UML). Our work extends and adapts the typed links proposed in [NOB97]. Relationships are classified into five categories, for which examples are presented in Figure 5 and Figure 6.

- 

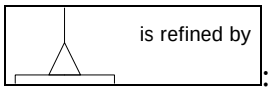
: a pattern *is completed by* another one when it acts as a "traffic guard", splitting a generic problem into a group of sub-problems addressed by the patterns that complete it, and leading the user to these patterns. This type of relationship is represented using the notation used in UML class diagrams for the *aggregation* relationship type. For instance, *Connection of Presentation and Decision Situation* is completed by *Instances Binding*, *Graphical Appearance* and *Semantics Behavior*.
- 

: a pattern *is required for* another pattern when it has to be used necessarily before using the other pattern. For instance, *Instance Binding* must be applied before *Graphical Appearance* and *Semantics Behavior*. This type of relationship is represented using the notation used in UML class diagrams for the *association* relationship type.
- 

: a pattern *leads to* another one when it leaves an unsolved problem or when the solution applied generates a problem that can be solved by the other pattern. It represents a weaker pattern relationship than *is required for* and *is completed by*. For instance, *Graphical Appearance* suggests the use of *Semantics Behavior* to deal with the unsolved problem of incorporating semantics behavior in presentation objects. The notation is similar to *is required for*, but with an empty arrow.
- 

: a pattern *conflicts with* another pattern when it represents a variant solution that cannot be used in conjunction with another one. We represent this type with a bi-directional dotted arrow. For instance, in Figure 6, *Instances Binding WB*

and *Instance Binding BB* conflicts because they provide alternative solutions to the same problem of instances binding, based respectively on white and black-box approaches.

- 

: a pattern *is refined by* another one when the latter addresses a problem that is a specialization of the problem addressed by the former. This specialization can be of various natures, in order to deal with a more specific version of the shared problem, context and/or forces. We represent this type with the notation used in UML class diagrams for the *specialization* relationship type. For instance, *Instances Binding WB* and *Instance Binding BB* refine *Instances Binding*, because they are used exclusively in more specific contexts, taking into consideration different resulting contexts.
- *uses*: this relationship is not represented visually in graphs, due to its low granularity and its irrelevance to the domain at hand. A pattern *uses* another pattern when its solution applies a domain-independent design pattern, available in catalogs such as [GAM94]. For instance, *Notification of Semantics State Changes* applies the design pattern *Event Notification* [RIH96] to create an observer/subject [GAM94] mechanism to maintain consistency between related presentation and decision objects.

7.4. A Pattern Language Fragment for the DSS Architecture

The issue of developing a pattern language for the layered DSS Architecture case study was addressed in [GER99], where a language portion was designed for the restricted problem of integrating presentation and decision situation layers. The language involves concerns related to both abstractors and elaborators point of view, i.e., how to develop frameworks thinking in integration and how to integrate them in specific DSS applications.

Recall from Section 5 that generic presentation frameworks can be reused to represent different decision situation models, and that a same decision problem can be given different visual representations. From a decision-maker point of view, a presentation structure is a graphical representation designed to provide a more intuitive representation of a decision model. For example, Figure 4 represents a budget model as a tree of budgetary items. Budgetary itens are described by a label and nature (investment (V), income (I), expense (E) and balance (B)).

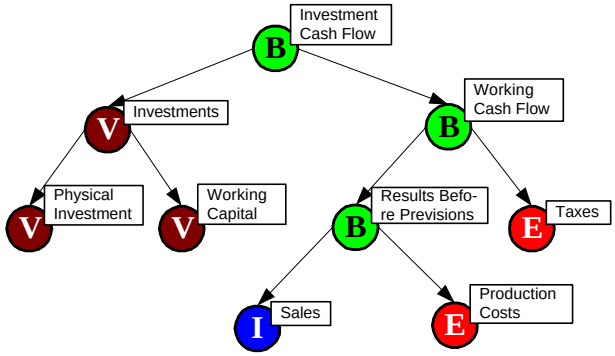


Figure 4 – Instance of Presentation Framework Graph Representing a Budget Item Hierarchy

The language uses as a running example the presentation frameworks *Graph* and *IndentTree*, and the decision situation framework *Capital Budgeting*. *IndentTree* and *Graph* represent generic presentation structures as the names suggest, and the *Capital Budgeting* framework captures a generic decision model to capital budgeting problems. These presentation structures are used to manipulate and visualize the hierarchical structure of the budget items composing a capital budgeting model. Figure 4 shows a possible graphical appearance of a budgeting model instance graphically represented by the adapted classes of the *Graph* framework.

Figure 5 shows the problem-oriented graph for this language fragment. The initial pattern *Connection of Presentation and Decision Situation* introduces the complex problem addressed by the language. It is split into three more specific problems, addressed by the patterns *Instances Binding*, *Graphical Appearance* and *Semantics Behavior*. In a specific application, presentation and decision

situation objects need to communicate, as suggested by the interaction scenarios depicted in Figure 3, and therefore they must be adapted in the connection process in order to allow the binding of instances of both layers, an issue covered by *Instances Binding* pattern. Considering the example of Figure 4, any graph node must know the budgetary item it visually represents. In addition, presentation objects must offer an adapted graphical appearance, according to the particularities of the decision objects they will represent. In our example, the chosen appearance is a circle with the item nature (e.g. Income), with the item label displayed next to the circle (e.g. Sales). *Graphical Appearance* addresses all the details of this problem. Finally, presentation objects must incorporate the behavior of related decision situation objects, i.e., semantics messages sent by the dialog layer must be forwarded to decision objects, and possible changes in decision objects state must be propagated to the corresponding presentation objects. *Semantics Behavior* deals with this issue. The application of this latter pattern, on the other hand, involves dealing with more problems, captured by other patterns, and so forth.

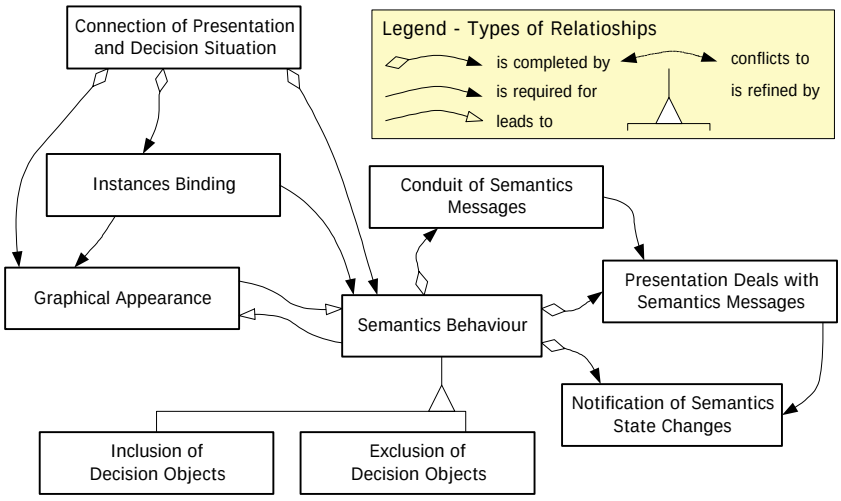


Figure 5 – Problem-Oriented Graph

Part of the solution graph developed for this language fragment is depicted in Figure 6. In our specific example, solution-oriented patterns reflect white and black-box approaches to solve a same framework integration problem (see section 7.2).

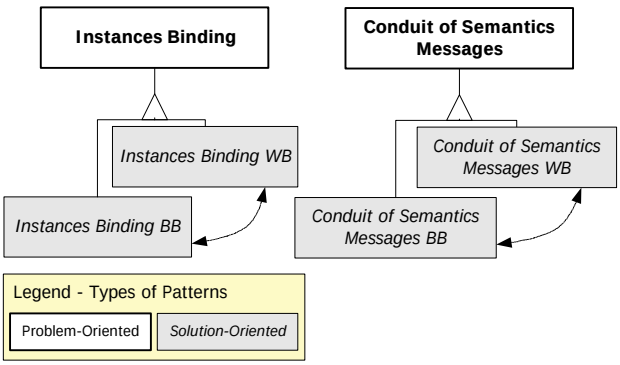


Figure 6 – Solution-Oriented Graph

8. Conclusion and Future Work

In this paper, we have discussed some problems of framework integration, particularly in layered software architectures, and how pattern languages could be used to solve these problems, using a DSS layered architecture as a case study.

We purposed the separation and encapsulation of solutions variants to a same problem in problem and solution-oriented patterns. In our case study, it allowed the adequate use of white and black-box approaches to deal with framework integration problems, harmonizing the reusability and adaptability forces, leaving to the abstractor to choose a more satisfactory solution to a given context.

As several pattern languages in the literature, we suggested the use of visual graphs to facilitate the use of pattern languages as step-by-step guides to deal with the complex problem of framework integration between layers. Particularly, we purposed the use of typed links to denote semantically distinct kinds of relationships between patterns, each one with a special appearance in pattern language graphs.

The pattern language fragment developed for the case study using our approach is in an embryonic state. Abstractors and elaborator are

starting to apply it in constructing specific DSS, validating and verifying the feasibility of our propositions.

We intent, as future work, to develop a pattern language to all layers of the architecture used as a case study. Moreover, we believe we could generalize the solutions provided for problems of framework integration of the case study to a great spectrum of layered software architectures based on framework integration.

9. Bibliography

[APP97] APPLETON, Brad. **Patterns and Software: Essential Concepts and Terminology**. Available in the WWW at http://www.enteract.com/~bradapp/. June 1997.

[BAU97] BÄUMMER, Dirk et al. Framework Development for Large Systems. **Communications of the ACM**, New York, v. 40, n. 10, pp. 52-59, Oct. 1997.

[BCK94] BECK, Kent. Pattern and Software Development. **Dr. Dobb's Journal**, San Francisco, v. 19, n. 2, pp. 18-21, Feb. 1994.

[BEC93] BECKER, Karin. **Reusable *Frameworks* for Decision Support Systems Development**. Namur, 1993. Thesis (Doctor Degree in Computer Science) – Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix.

[BEC98] BECKER, Karin; BODART, François. An Object-Oriented Framework-Based Architecture for Decision Support Systems. **CLEI Electronic Journal**, Santiago - Chile, v. 1, n. 1, pp. 1-20, June 1998.

[BRI91] BRIGHAM, E.F. & GAPENSKI, L.C. **Financial Management: Theory and Practice of Managerial Finance**. 6. ed. [S.l.] : Dryden Press, 1991.

[BRU97] BRUGALI, Davide; MENGA, Giuseppe; AARSTEN, Amund. The Framework Life Span. **Communications of the ACM**, New York, v. 40, n. 10, pp. 65-68, Oct. 1997.

[BUS96] BUSCHMANN, Frank et al. **Pattern-Oriented Software Architecture – A System of Patterns**. Chichester : John Willey and Sons, 1996.

[COP95] COPLIEN, James; SCHMIDT, Douglas. **Patterns Languages of Program Design 1**. Reading, MA : Addison-Wesley, 1995.

[DYS98] DYSON, Paul; ANDERSON, Bruce. State Patterns. In: MARTIN, Robert; RIEHLE, Dirk; BUSCHMANN, Frank (eds.). **Pattern Languages of Program Design 3**. Reading, MA : Addison-Wesley, 1998. pp. 125 a 142.

[FAY97] FAYAD, Mohamed; Schmidt, Douglas. Object-Oriented Application Frameworks. **Communications of the ACM**, New York, v. 40, n. 10, pp. 32-38, Oct. 1997.

[FOW97] FOWLER, Martin. **Analysis Patterns – Reusable Object Models**. Reading,MA : Addison Wesley, 1997.

[GAM94] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading, MA : Addison-Wesley, 1994.

[GER98] GERBER, Luciano; BECKER, Karin. Using Design Patterns to Evaluate an OO Architecture for DSS Development**.** In: Conferencia Latinoamericana de Informática – CLEI (14. : 1998 : Quito). **Proceedings...** Quito: ACTAS, 1998. pp. 273-288.

[GER99] GERBER, Luciano. **A Pattern Language to Development of Decision Support Systems based on Frameworks**. Porto Alegre, Brazil, 1999. Thesis (Master Degree) – Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul.

[JOH92] JOHNSON, Ralph. Documenting Frameworks using Patterns. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS (7.: 1992 : Vancouver). **Proceedings...** New York : ACM SIGPLAN Notices, 1992. pp. 63-76.

[JOH97] JOHNSON, Ralph. Frameworks = (Components + Patterns). **Communications of the ACM**, New York, v. 40, n. 10, pp. 39-43, Oct. 1997.

[MAR98] MARTIN, Robert; RIEHLE, Dirk; BUSCHMANN, Frank. **Pattern Languages of Program Design 3**. Reading, MA : Addison-Wesley, 1998.

[MAT98] MATTSON, Michael. **Object-Oriented Frameworks: A Survey of Methodological Issues**. Ronneby, Sweden, 1996. Thesis (Doctor Degree) – Department of Computer Science and Business Administration, University College of Karlskrona/Ronneby.

[MAT99] MATTSON, Michael; BOSCH, Jan; FAYAD, Mohamed E.. Framework Integration: Problems, Causes, Solutions. **Communications of the ACM**, New York, v. 42, n. 10, pp. 81-87, Oct. 1999.

[MES98] MESZAROS Gerard; DOBLE, Jim. A Pattern Language for Pattern Writing. In: MARTIN, Robert; RIEHLE, Dirk; BUSCHMANN, Frank (eds.). **Pattern Languages of Program Design 3**. Reading, MA : Addison-Wesley, 1998. pp. 529-574.

[NOB97] NOBLE, James. **Classifying Relationships Between Design Patterns**. Available in the WWW at http://www.mri.mq.edu.au/~kjx/. April 1998.

[PRE95] PREE, Wolfgang. **Design Patterns for Object-Oriented Software Development**. Reading, MA : Addison-Wesley, 1995.

[RIH96] RIEHLE, Dirk. The Event Notification Pattern – Integrating Implicit Notification with Object-Orientation. **Theory and Practice of Object Systems**, New York, v. 2, n. 1, p. 43-52, 1996.

[SPR80] SPRAGUE, R.H.. A Framework for Research on DSS. In: FICK, G.; SPRAGUE, R.H. (eds.), **Decision Support Systems: issues and challenges**. [S.l.] : Pergamon Press, 1980. pp. 5-22.

[TAN92] TANEMBAUM, Andrew. **Modern Operating Systems**. Englewood Cliffs, New Jersey : Prentice-Hall, 1992.

[VLI96] VLISSIDES, John et al.. **Patterns Languages of Program Design vol. 2**. Addison-Wesley, Reading, MA, 1996.

[ZIM94] ZIMMER, Walter. Relationships Between Design Patterns. In: COPLIEN, James; SCHIMDT, Douglas (eds.). **Pattern Languages Of Program Design 1**. Reading, MA : Addison-Wesley, 1994. pp. 345-364.