Modeling Design Patterns for Semi-Automatic Reuse in System Design

Galia Shlezinger

Faculty of Industrial Engineering and Management,
Technion-Israel Institute of Technology, Haifa 32000, Israel

galias@tx.technion.ac.il

Iris Reinhartz-Berger*

Department of Management Information Systems,
University of Haifa, Haifa 31905, Israel

iris@mis.haifa.ac.il

Dov Dori

Faculty of Industrial Engineering and Management,
Technion-Israel Institute of Technology, Haifa 32000, Israel

dori@ie.technion.ac.il

* Corresponding author

Modeling Design Patterns for Semi-Automatic Reuse in System Design

Abstract

Design patterns provide reusable solutions for recurring design problems. As such, they constitute an important tool for improving software quality. However, correct usage of design patterns depends to a large extent on the designer's knowledge, experience, and interpretation. Design patterns often include models that describe the suggested solutions, while other aspects of the patterns, such as the problems they intend to solve and the ways for integrating the solutions into different contexts, are neglected or described informally only in text. Furthermore, design pattern solutions are usually described in an object-oriented fashion that is too close to the implementation, masking the essence of and motivation behind a particular design pattern. We suggest an approach to modeling the different aspects of design patterns and semi-automatically utilizing these models to improve software design. Evaluating our approach on commonly used design patterns and a case study of an automatic application for composing, taking, checking, and grading analysis and design exams, we found that the suggested approach successfully locates the main design problems modeled by the selected design patterns.

1 Introduction

Patterns are types of themes that specify recurring processes, events, or elements. Commonly used in different engineering fields, patterns can be classified into analysis patterns, design patterns, organization patterns, process patterns, and domain-specific patterns. Batra [2] claimed that pattern recognition can be considered as a conceptual modeling technique. He further stated that the main challenge with patterns is "to provide a limited number of patterns at a fairly high level of abstraction and an appropriate level of granularity…" Identifying locations in which patterns might be applied can be a tedious task [9]. Purao et al. [28] described a prototype for automatically generating a conceptual design based on analysis patterns. Their research, which concerns high level analysis and design, aims at producing models automatically from textual requirements. Blomqvist [3] presented an approach for creating ontology patterns semi-automatically by utilizing knowledge organized as patterns from other areas, like data modeling, knowledge reuse, software analysis, and software design. While these approaches are appropriate for high-level conceptual modeling activities, they are less applicable at the

detailed design stage. As the level of design becomes more specific, different parameters, such as code optimization, system limitations, and other non-functional requirements, have to be taken into consideration. These require careful and detailed descriptions of the patterns, their essence, and their exact usages.

In this work, we concentrate on design patterns, which describe reusable solutions for recurring design problems in given contexts [12]. While design patterns may help produce better design and implementation of applications [27, 31], their appropriate use is often hindered due to the inherent ambiguity in the existing ways of description and representation [35]. This may impede effective use of design patterns in particular applications, as designers may not be able to appreciate benefits or predict shortcomings associated with their proper use [7]. Misuse of design patterns typically results from failure to understand the rationale behind the patterns [37] or from difficulties in incorporating the patterns into a specific system design [1].

In view of these observations, we have developed an approach, accompanied by a tool, for clearly modeling design patterns and applying them semi-automatically to system models. In doing so, we utilize knowledge we have gained from modeling and categorizing design patterns. Our approach supports modeling of the different aspects of design patterns, including their problem and solution specifications, their essence, and their correct usage in particular systems. The design pattern models are specified using Object-Process Methodology (OPM) [8], which enables describing structural and behavioral aspects of design patterns at different granularity and abstraction levels. These design pattern models provide the basis for an algorithm for searching design problems in a given system design and suggesting corrections for them in the form of design patterns. Evaluating our approach and the feasibility of its algorithm using a case study research methodology, we found that the suggested approach successfully locates the main design problems modeled by the selected design patterns.

The structure of the rest of the paper is as follows. Section 2 reviews the relevant literature. Section 3 focuses on the suggested design patterns representation method, while the emphasis of Section 4 is on the algorithm for improving a system's design via the use of design patterns. Section 5 reports on the evaluation of the approach using a case study of an automatic application for composing, taking, checking, and grading

analysis and design exams and eight commonly used design patterns. Finally, Section 6 summarizes and proposes possible future research directions.

2 Literature Review: Design Patterns Representation

Design patterns are widely recognized as an important technique for software design and programming. They are usually described using a template, such as the well known one proposed by Gamma et al. [12]. These descriptions are often accompanied by graphical notations. Using UML [22], for example, class diagrams are recruited for modeling the structural aspects of the design patterns, while sequence diagrams are used for specifying their behavioral aspects. Since UML is ill-equipped for precisely representing design patterns [13], different methods and notations have been proposed for representing design patterns. These can be roughly divided into two groups: UML-based methods and proprietary languages.

UML-based methods extend the UML metamodel or define UML profiles in order to represent the different aspects of design patterns. Kim et al. [11, 17], for example, presented RBML, a UML-based meta-language for specifying design patterns. One of the main drawbacks of RBML is that the designer needs to mentally integrate information from three different design patterns specifications: static, interaction, and state machine. Indeed, Kim et al. indicated that their approach is intended for developers of tools that incorporate patterns into UML models. Gueannec et al. [13] proposed specifying design patterns using OCL [36] and meta-level collaboration in UML. Furthermore, they suggested using temporal logic to represent the design pattern's behavioral constraints. This kind of abstraction may be too complex for the average designer. Mak et al. [19] extended UML to describe pattern leitmotifs [10], which are distinguishing structures that define the idea behind design patterns rather than their implementation. Lauder and Kent [18] proposed a three-layered model to describe design patterns in UML. Although this work captures the essence of a design pattern solution, it does not express the pattern's intent or problem description. Pickin and Manjarrés [26] addressed the need to describe information about how and when to use design patterns. They suggested adding a summary of the informal parts of the design pattern description to a formal description of the patterns using a markup language called PCML [25]. While this approach may be

useful for building tools that support design patterns, markup languages are machine-understandable and require development and implementation of dedicated tools. Dong et al. [7] presented an extension to UML that may help identify patterns in the system model, but it does not add information about the pattern itself.

DPML [20] is a proprietary language that enables modeling and reusing design pattern solutions. However, this notation does not support specifying the locations within the system model in which the design patterns should be used. Furthermore, a designer using DPML has to deal with more than one language in order to use design patterns. LePUS [9] is a formal specification language based on a theory of object-oriented design in mathematical logic. The level of abstraction of LePUS is high, its visual representation is quite complex, and the symbolic representation is formal, making it difficult for designers to work with. Dong et al. [6] suggested Object-Z and first order logic for representing design pattern structure and composition, and temporal logic for describing their behavioral aspect. This method is used for design pattern composition and verification rather than for their straightforward implementation in a system model.

In summary, the reviewed UML-based methods are implementation-oriented, and they concentrate on the solutions proposed by the different design patterns. Furthermore, they model behaviors as encapsulated in object classes, complicating the task of reusing design patterns in different contexts that crosscut the system's structure. UML-based methods do not convey the essence of the design pattern as a complete problem-solution pair. Most proprietary languages, on the other hand, are too formal and complex for practical use by average software designers, a factor that discourages their adoption [17].

3 A Layered Design Pattern Framework and Its Implementation

Catering to a number of abstraction levels, the classical OMG framework for metamodeling [23] consists of four layers: the information layer (M0), the model layer (M1), the metamodel layer (M2), and the meta-metamodel layer (M3). Kim et al. [11, 17] defined the notion of model roles at level M2 and explained how they are used for describing design patterns in RBML. Reinhartz-Berger and Strum [30] referred to domain models in the context of the classical four-layered framework: domain models specify the commonality and allowed variability of application families.

Inspired by these ideas, we define three layers of abstraction for representing design patterns: the system layer, the design pattern layer, and the meta-design pattern layer. At the most concrete layer, the system layer, a specific design pattern is implemented within the context of a system model. This implementation adapts the solution part of the selected design pattern model to fit into the specific context of the system. The design pattern itself belongs to the second layer of abstraction, the design pattern layer, where all the design patterns are modeled. The most abstract layer in this framework, the meta-design pattern layer, defines a metamodel of the design pattern concept, which conceptualizes the commonality and possible variability among all the design patterns.

This three-layered framework can be used in conjunction with different modeling languages. We have chosen Object-Process Methodology (OPM) [8] (see Appendix A) as the modeling language in this work due to the following reasons. First, OPM views objects and processes as equally important first-class entities, enabling concurrent specification of both structure and behavior. Second, the refinement-abstraction mechanisms in OPM help maintain an OPM model consistent among the different abstraction levels and provide for comparing models at different abstraction levels. The OPM models of the design patterns are problem-oriented, unlike the solution orientation of UML design pattern models, which has been criticized as being too close in its abstraction level to programming languages [26]. Finally, OPM provides tools for classifying elements based on elements in other models [34]. This capability is important when different layers are involved, as in our proposed design pattern framework. Each OPM entity—object, process, or state—exhibits two associated features: role and multiplicity indicator. Like UML stereotypes, a *role* is a kind of a model entity whose information content and form are the same as those of the basic model entity, but its meaning and usage are different. Roles, which are specified in the upper left corner of the entity frame, are used within a system model in order to associate an entity to its design pattern counterpart. A *multiplicity indicator*, specified in a design pattern model, constrains the number of system entities that can be instantiated from the same design pattern entity in any system that applies this design pattern. The multiplicity indicator appears in the right lower corner of an entity frame and is optionally many (0..n) by

default. Cardinality of links is used for constraining multiplicities of structural and behavioral relationships.

3.1 The Meta-Design and Design Pattern Layers

In the meta-design pattern level, each design pattern is specified by three models: problem, solution, and correction models. A *problem model* represents a recurring design problem that the design pattern aims to solve, while a *solution model* represents the proposed solution for that recurring design problem. A *correction model* helps the designer, or an automated tool, correct the problem in the spirit of the suggested solution. In other words, the correction model is a mapping between the problem model and the solution model of the same design pattern. A single design pattern may have several problem models, but it can have only one solution model, independently of the domain to which the pattern is applied. The result of applying this solution model to a specific system may take different variations, guided by the various correction models associated with the specific design pattern.

As an example, consider the ***observer*** design pattern. This pattern offers a solution to a problem of notifying a set of objects when some change in the state of another object takes place. A one-to-many relationship exists between the "subject" object, whose state is changed, and the "observer" objects that need to be notified. Figure 1 is the observer design pattern specification of Gamma et al. [12, p. 294]. This solution model, expressed in terms of UML class and sequence diagrams, specifies how a change in the state of a (concrete) subject is sent to the different observers, using their *notify* operations. Spanning two abstraction levels, it explicitly describes how the abstract classes **Observer** and **Subject** are respectively inherited by **Concrete Subject** and **Concrete Observer**. Furthermore, informal notes are required in order to convey the full meaning of this design pattern.

In contrast to UML, the observer pattern is expressed in OPM in three separate models. The problem model represents the essence of what the design pattern aims to solve. The model in Figure 2(a), for example, shows that the observer pattern deals with situations in which a change in an object of type **A** triggers the **Changing B** process, which may modify at least two different objects of type **B**. The problem to be solved here is that the

Changing B processes should know all types of **Object B** that can be influenced by a certain change in **Object A**. The most severe manifestation of this problem is a design in which **Object A** and **Object B** are both implemented by the same class.

The design pattern solution model specifies the solution to the recurring design problem. In the observer case, the solution model, depicted in Figure 2(b), includes two processes: **Notify**, in which a change in the **Subject** may affect the **Observer**, and **Change Subscription**, in which an **Observer** is added to or removed from the observer list – a list of potentially affected objects associated with **Subject**. In other words, whenever there is a change in **Subject**, **Notify** announces all its recognized **Observers**. To be included in or removed from the observers list, **Observer** has to trigger the **Change Subscription** process. This process generalizes the methods *Attach* and *Detach* in the UML model of Figure 1.

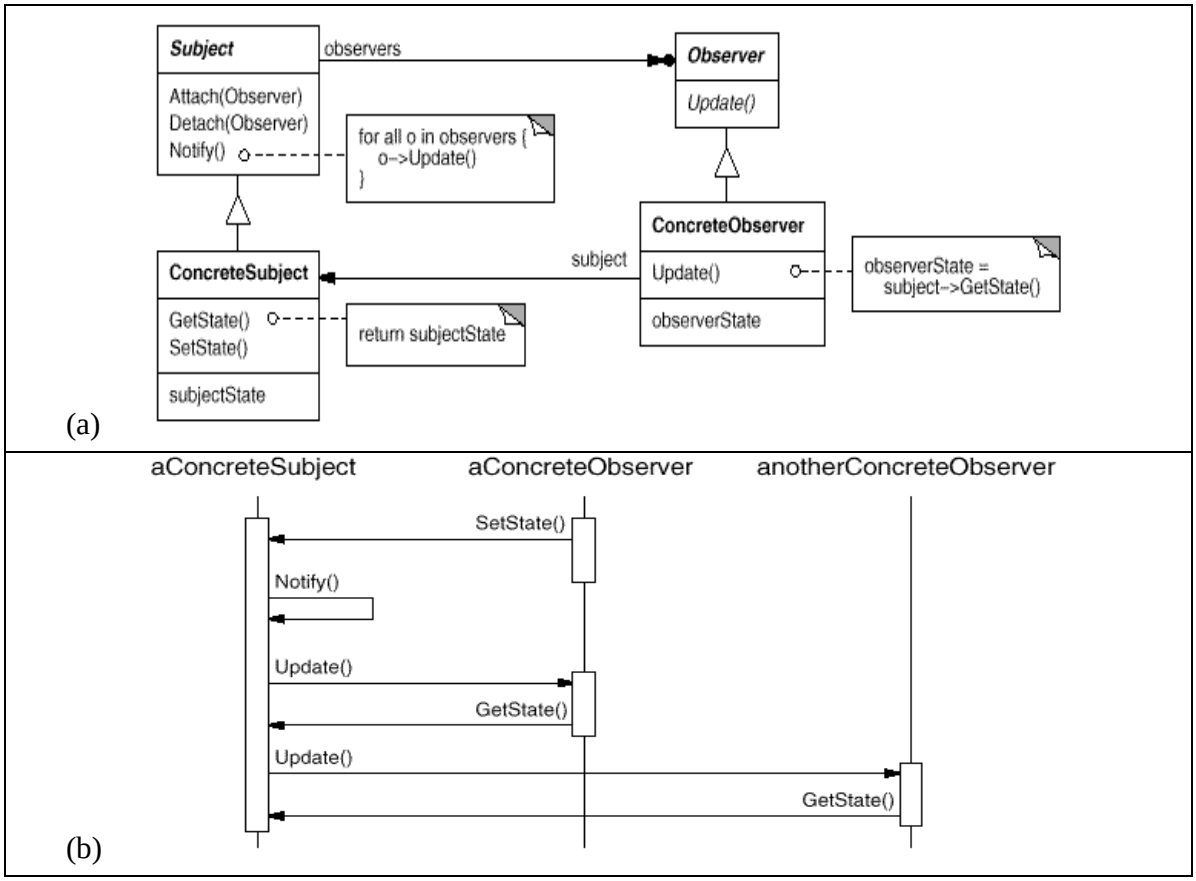


Figure 1. The observer design pattern specification of Gamma's et al. [12, p. 294]:
(a) the class diagram (b) the sequence diagram

Finally, the correction model maps the design pattern solution model onto the problem model. This is done by employing the OPM role mechanism: when applicable, the entity

names in a correction model are taken from the problem model, while their roles are taken from the corresponding solution model. Similarly, the link cardinalities are taken first from the problem model, and only later, new links from the solution model are added. This use of the OPM role mechanism clearly identifies which design patterns were used in the system model and in what way. This solves problems resulting from loss of information regarding the applied design patterns in particular systems [7].

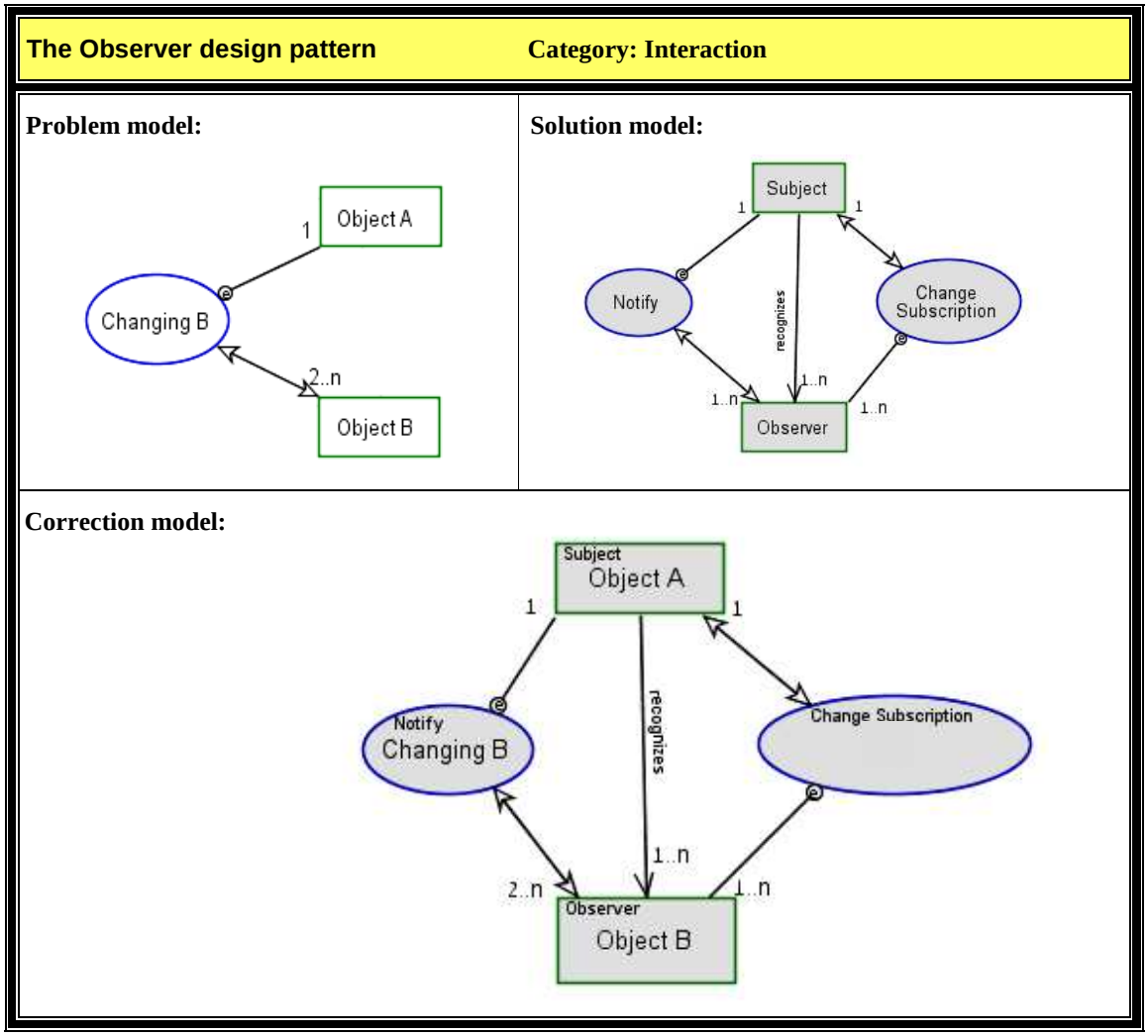


Figure 2. The observer design pattern models: (a) the problem model, (b) the solution model, and (c) the correction model.

The correction model of the observer design pattern, depicted in Figure 2(c), states that the process **Changing B** plays the role of **Notify**, **Object A** plays the role of **Subject**, and **Object B** plays the role of **Observer**. **Object A** triggers the **Changing B** process, which

informs the relevant **Objects B** whenever **Object A** is modified. In order to actually inform only the "relevant" **Objects B**, a new process, which plays the **Change Subscription** role, maintains the relevant set of **Objects B** that are recognized by **Object A**. This process is not named in the correction model as it is introduced by the solution model and should be named by the designer.

Comparing the UML and OPM solution models, one can notice that while the UML model of the observer design pattern contains two types of diagrams, class and sequence diagrams, the OPM model is represented with one diagram type, which displays both the static and the dynamic aspects of the model. Furthermore, while the UML solution model spans two abstraction levels, explicitly describing how the abstract classes **Observer** and **Subject** are respectively inherited by **Concrete Subject** and **Concrete Observer**, the corresponding OPM model describes only one abstraction level, which models the relations between **Subject**, **Observer**, **Notify**, and **Subscribe**. The correction models are the ones that hold the information on how to integrate design patterns into system models. Note that the essence of the design pattern, as expressed in the OPM problem model, is not described at all in the UML model.

Appendix B provides problem, solution, and correction models of seven other commonly used design patterns which have different purposes [12].

3.2 The System Layer

The system layer is the most concrete one, as it includes design models of different systems and applications. Figure 3 describes an OPM model of a system that displays time in both analog and digital format. It does not apply any design pattern. The main process in this system, **Time Processing**, zooms into three processes: **Counter Updating**, which updates the **Time Counter**, as well as **Digital Display Updating** and **Analog Display Updating**, which change **Digital Time Display** and **Analog Time display**, respectively.

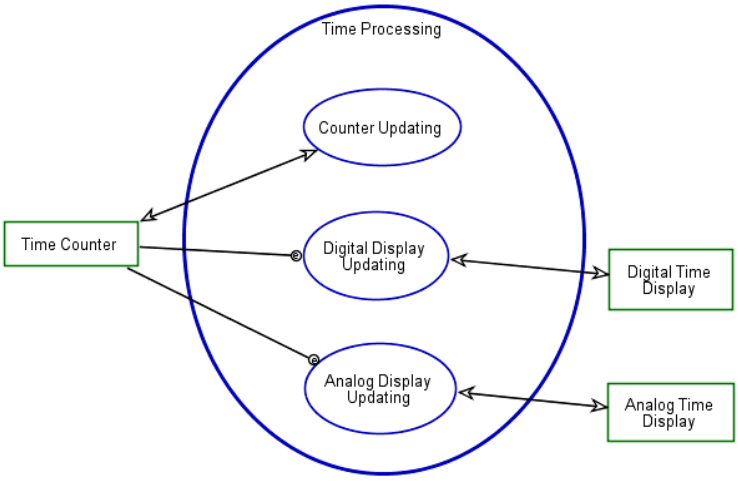


Figure 3. A partial OPM model for a time display system

Examining this model, one can miss the observer problem model at first glance. Nevertheless, zooming out of **Time Processing**, Figure 4(a) specifies the system top level, where the observer problem model is clearly noticeable, as **Time Counter** matches **Object A**, **Digital Time Display** and **Analog Time Display** both match **Object B**, and **Time Processing** matches **Changing B**. Applying the observer correction model, Figure 4(b) specifies the time processing system, which implements the observer design pattern.

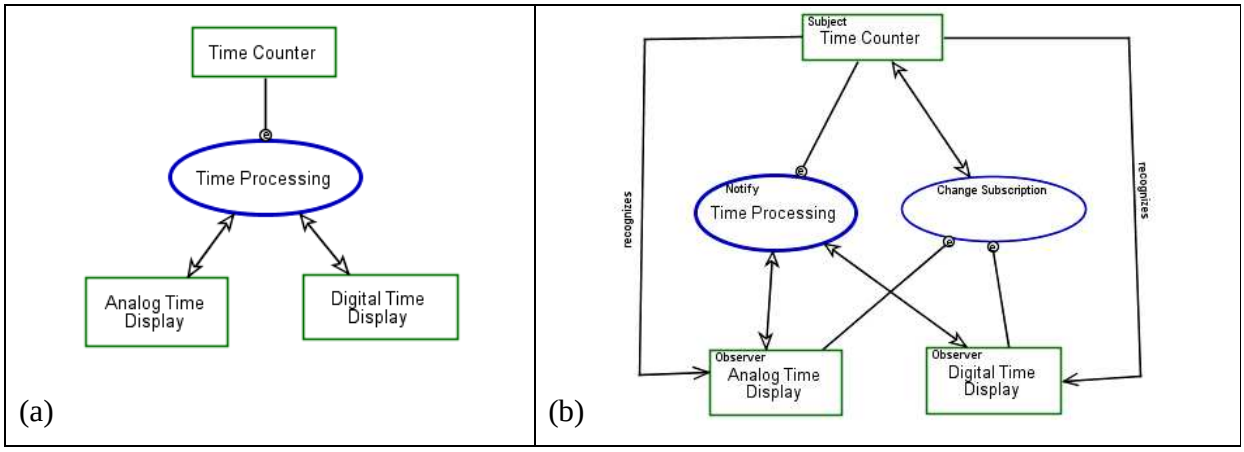


Figure 4. The OPM top level model of the time display system: (a) before applying the observer design pattern and (b) after applying it.

Analyzing the lower, detailed level of the time display system, one can argue about the roles that the different sub-processes play in the context of the observer design pattern. In particular, the processes **Digital Display Updating** and **Analog Display Updating** can be

viewed as two different **Notify** processes, while **Counter Updating** has no role in this context. Hence, it is reasonable for a human designer to implement the observer design pattern in the time display system as specified in Figure 5. Note that this design specifies also that the **Change Subscription** process can be triggered by **Analog Time Display** *or* by **Digital Time Display** (as there may be cases where only one is required for a single subscription).

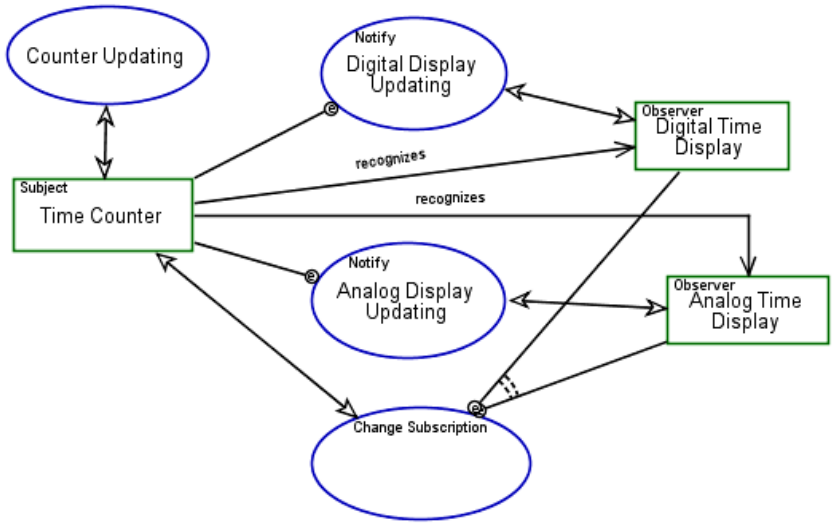


Figure 5. The time display system model after correctly applying the observer design pattern

3.3 A Naturally-Induced Categorization Schema for Design Patterns

One of the important features of our proposed design patterns representation approach is that it induces a natural categorization schema of design patterns. Different categorization schemes have been proposed over the years in order to improve the comprehensibility and correct usage of design patterns, e.g., [14], [5], [12], [16], [21], and [40]. Generally, a design pattern categorization schema should support both understanding of the function of the design pattern and retrieving those patterns that are appropriate for the problem at hand. Buschmann et al. [14, 5] identified five properties that a categorization schema must possess: (1) simplicity and ease of learning, (2) being comprised of only a few classification criteria, (3) reflecting natural properties of patterns for each categorization criterion, (4) providing a roadmap that leads users to several possible patterns, and (5) being extensible in order to provide for new patterns to be easily integrated.

Analyzing the different OPM models of the design patterns presented in [12], we identified common fundamental OPM constructs that appear in several design patterns and help explain their essence. Although the common OPM constructs sometimes appear in the design pattern problem models, the pattern solution models naturally specify the design pattern's intention better than their problem model counterparts. For clarity purposes, these OPM constructs are marked in grey in the models in Figure 2 and Appendix B. The remarkable commonality we have observed induces a design patterns categorization schema, which satisfies the five properties identified by Buschmann et al. [14, 5]. This categorization schema includes four groups, which are summarized in Table 1, but can be extended and refined as new design patterns are introduced.

The first group in our schema is *creational design patterns*. Like the creational group in [12], it is characterized by the OPM construct of *process – result link – object* (see Table 1). This construct purely conveys the idea of a process creating an object.

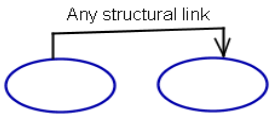
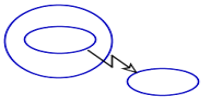
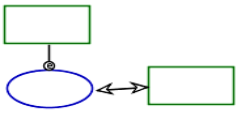
The second group, *structural composition design patterns*, shares in common the abstract characterizing construct of two elements that are connected via a structural relation of some type. The common construct that characterizes structural composition design patterns consists of two processes. However, there may be cases in which the common construct consists of two objects connected by a structural relation. While we did not identify such a construct in any of the design patterns listed in [12], if such a construct is found, this group may be further divided into two sub-groups: static structural composition and dynamic structural composition. The OPM constructs of these groups will be quite similar. However, static structural composition patterns will refer to objects and their structural links, while dynamic structural composition will refer to processes and their structural links.

The patterns in the third group are classified as *wrapper design patterns* since they solve their stated problems by wrapping the original functionality. Such design patterns have already been recognized as wrappers in [12] and [32]. Their common characterizing OPM construct justifies both the existence of this category and its name.

Finally, the fourth group is the *interaction design patterns* group. The patterns in this group focus on the interaction between structural and behavioral aspects of the solution. In OPM terms, these patterns emphasize procedural links, namely effect and event links,

which are responsible for updating objects and for triggering processes, respectively. In the **Observer** design pattern solution model, this construct appears twice: one for notifying the subjects and the other for changing their subscription.

Table 1. Categorization of design patterns according to their OPM models

Design Pattern Category	Design Pattern Examples	Typical OPM Construct
Creational	Factory Method Builder	
Structural composition	Chain of Responsibility Composite Template Method	
Wrapper	Decorator	
Interaction	Observer Command	

4 Automatically Improving Software Design with Design Patterns

Utilizing the knowledge gained from modeling and categorizing different design patterns in our approach, we present in this section an algorithm that scans system design models, expressed in OPM, searches for different embedded problem models, and suggests the application of relevant design pattern solutions using their correction models. The OPM design models, which are the inputs of this algorithm, can be either created manually by designers or reverse-engineered from code. They are represented as object-oriented data structures internal to OPCAT [24], the OPM-based modeling software environment.

The algorithm, called Pattern Candidate Finder, searches for system model portions that are structurally equivalent to design pattern problem models. Here, we use the ability to compare OPM models at different abstraction levels, since usually the system model is concrete, while a design pattern problem model is general and more abstract. When a match is found, the tool suggests that the designer replace the system model portion with

the corresponding design pattern solution model. It further suggests how to do this by presenting the corresponding correction model. The designer can choose whether to accept the suggestion, reject it, or manually apply the design pattern into the system, taking into consideration factors such as the semantics of the system model and the optimization of code. Figure 6 presents the pseudo-code of the main method, Run, of the Pattern Candidate Finder class. This class has two additional methods, Search, which is presented in Figure 7, and Display, which textually displays the design problems that were found and ways to solve them.

```
Run (system model μ) {
        foreach pair of problem model P_i and correction model S_i in the repository {
                Candidates =Search (P_i, μ)          // searches for a candidate part P_i in μ
                foreach M in Candidates {
                        Display (M, S_i)             // S_i is a possible solution for the candidate part
                }
        }
}
```

Figure 6. The Pseudo-code of the *Run* method of the Pattern Candidate Finder class

In the search method, we adapted the structural equivalence algorithm presented in [33] and modified it to support the different layers at which the design pattern and system models reside. We treat an OPM diagram as a directed graph whose nodes are things (objects or processes) and edges are links. The search method finds a path between two nodes that is structurally (but not necessarily semantically) equivalent to a direct link between these things. This equivalence is deduced primarily according to the dominance of OPM links, which is determined using the link semantics and the abstraction order1 [29]. Each link in a design pattern problem model is considered as a query model, which is searched for equivalent paths in the system model, discarding combinations of paths that do not create a valid OPM model.

[1] The abstraction order defines for each two procedural links a third procedural link which replaces the two when a thing (process or object) is abstracted (folded, out-zoomed, or state-suppressed).

```
Search (problem model P, system model μ) {
    Matches = new MatchGroup()
    linksInP = P.Get_Links ();                    //creates a group of all the links in the problem model P
    foreach l in linksInP {
          sourcesInSystemModel = μ.Get_Same_Type_System_Model_Entities(l.getSource());
          //finds all entities in the system model that are of the same type (object, process, or
          //state) as the link source
          destinationsInSystemModel = μ.Get_Same_Type_System_Model_Entities(l.getDestination());
          //finds all entities in the system model that are of the same type (object, process, or
          //state) as the link destination

          foreach source in sourcesInSystemModel {
                foreach destination in destinationsInSystemModel {
                      l.paths.Add(StructuralEquivalence(l, source, destination, μ));
           }
    }
    Matches = BuildMatches(linksInP); // builds valid OPM models which match the problem model P
                                          //by choosing one equivalent path for each link in P.
    return Matches;
}
```

Figure 7. Pseudo-code of the search procedure of the Pattern Candidate Finder algorithm

Due to the refinement/abstraction mechanisms of OPM that preserve consistency between the different diagrams of the same model, a particular design problem may appear at various abstraction levels. In order to report each design problem only once, a post processing stage is added to the Pattern Candidate Finder algorithm. At this stage, a candidate is discarded if one of its elements is not at its lowest level of abstraction and the associated design problem model does not explicitly contain different abstraction levels. As an example to this rule, consider Figure 4(a) and Figure 3. **Time Processing** from Figure 4(a) is refined (in-zoomed) in Figure 3. The observer design problem appears in these two diagrams twice, as presented in Table 2. However, the first case is an abstraction (zooming-out) of the second one. Thus, we wish that the algorithm reports only one design problem that correspond to the observer design pattern in this case. However, if the design problem model itself requires different abstraction levels, such as

in the case of the Command pattern problem model, then candidates that include elements which are not at the lowest level of abstraction and occur in the structure of the design problem model will be reported.

Table 2. The appearance of the observer design patterns in Figure 3 and Figure 4(a)

Case #	Changing B	Object A	Object B
Figure 4(a)	Time Processing	Time Counter	Digital Time Display, Analog Time Display
Figure 3	Time Processing (through Digital Display Updating and Analog Display Updating)	Time Counter	Digital Time Display, Analog Time Display

The complexity of the structural equivalence algorithm is $O(n_\mu)$ [33] and the complexity of the entire Pattern Candidate Finder algorithm is $O(n_\mu l_P)$, where l_P is the overall number of links in all the problem models in the algorithm's repository.

5 Evaluation of the Pattern Candidate Finder Algorithm

In order to evaluate the suggested Pattern Candidate Finder algorithm, we applied a case study research methodology [39]. This type of research methodology is adequate where (1) the research attempts to answer "how" or "why" questions, (2) the investigator has a little or no possibility to control the events, and (3) the research concerns a contemporary phenomenon in a real-life context. Our research satisfies these three conditions: (1) the research question is whether and how the suggested algorithm might help improve system designs, (2) system design is a mental process, over which we have virtually no control, and (3) system design is a common contemporary phenomenon in a real-life context of organizations such as software companies.

Following the case study research methodology, we run the suggested algorithm on several system designs, the largest of which is ExamPal. ExamPal is an automatic application for composing, taking, checking, and grading analysis and design exams. Its design model consists of 11 Object-Process Diagrams (OPDs) with a maximum of five detail (in-zooming) levels. The nature of this system leads us to believe that design patterns would be applicable. However, we did not instruct the creators of this system model how to design the system. Neither did we deliberately insert design problems to

the system model. Since the system is not concerned with distribution and reusability, we thought that wrapper or interaction design patterns were not relevant.

The system was modeled in OPM by a group of graduate and experienced undergraduate students who took an advanced information systems engineering course, called "Methodologies for Information Systems Development" at the Technion. All students in this course took several analysis and design courses, carried out annual information systems projects, and some of them have already worked as software designers. Thus, this population is comparable to industry junior designers. However, none of the students was experienced with design patterns. As part of their final grades, the students were required to develop a system for composing, taking, checking, and grading analysis and design exams, including specifying its requirements and analyzing them, designing feasible solutions that satisfy the requirements, implementing their design, and testing the system they had developed.

The eight design patterns used for running the Pattern Candidate Finder algorithm were the Factory Method, Builder, Decorator, Composite, Observer, Command, Chain of Responsibility, and Template Method. These patterns were selected as representatives of all the categories defined in [12] and are considered commonly used patterns [14, 15]. The models of these patterns are given in Appendix B, while the system model of ExamPal is given in Appendix C.

To evaluate the algorithm results, two experienced software designers, who are not the authors of this paper, were requested to specify which of eight design patterns they would use to improve the ExamPal design model. These experienced designers were further asked to indicate the exact places in which these design patterns should be applied. Comparing the results obtained from the experienced software designers with those proposed by the algorithm, we classified the pattern candidates into three groups: (1) hits, i.e., cases which the algorithm discovered design pattern candidates that were also recommended by the experts, (2) false positives, i.e., cases where the algorithm discovered design pattern candidates that were not recommended by the experts and should not be applied according to them, and (3) false negatives, i.e., cases which were recommended by the experts but were not found by the Pattern Candidate Finder

algorithm. When contradictions between the design experts were discovered, the candidate was considered as not applicable in that context.

For the Builder, Factory Method, Composite, Decorator, Chain of Responsibility, and Template Method design patterns, we found no false positives, no false negatives, and no hits. These findings are in line with our expectations of the design model, since the system does not possess characteristics that suggest using these patterns: it does not contain a general framework user interface specification, nor does it models a change in an existing system or deal with composite objects. **Error! Reference source not found.** summarizes the results we got for the other two design patterns, namely Observer and Command.

Table 3. Design experts vs. the suggested Pattern Candidate Finder algorithm

Design Pattern	Hits	False Positives	False Negatives
Observer	1	2	0
Command	4	4	2

Regarding the Observer design pattern, one hit and two false positives were found. No false negatives were found for this design pattern. The found false positives do indeed match the Observer problem model, but are not necessarily appropriate for implementing the Observer design pattern. This finding indicates the need for semantics-based techniques in addition to the structural (syntactic) ones, raising potential for future research.

Analyzing the Command design pattern, the algorithm originally found eight candidates, of which four were hits. Two design pattern candidates were missed by the algorithm in the ExamPal system. Checking the reasons for missing these candidates by the algorithm, we concluded that this may indicate that the Command problem model is not refined enough and requires further detailing.

To summarize, the suggested algorithm found the main places where design patterns have to be applied, however it also raised some false positives and in the Command case – two false negatives were detected as well. Thus, we concluded that the algorithm

accuracy and effectiveness in terms of the manual operations required from the designers depend on the following factors:

(1) The application of the structural equivalence algorithm in general and the selection rules in particular,

(2) The generality of the design pattern problem models, and

(3) The completeness or detail level of the system models.

6 Summary

We have introduced a three-layered framework for modeling design patterns. The layers are the meta-design pattern layer, the design pattern layer, and the system layer. The meta-design pattern layer includes specifications of commonalities among design patterns, allowing for variability among the different design patterns. This layer includes the different templates for documenting design patterns. We focused on a template that defines three model types for each design pattern: problem models, solution models, and correction models. The correction models map problem models to solution models. The design pattern layer hosts design pattern models in a complete and coherent way, which helps understand and use these patterns correctly in software system models. Finally, the concrete system layer includes specific system models that implement the various design patterns. The elements in the system layer are mapped to the roles they play in the more abstract representation of the design pattern solution models in the design pattern layer above it.

Evaluating the approach on eight commonly used design patterns and several case studies, we found that the suggested approach and algorithm successfully locates the main design problems modeled by the selected design patterns. However, the algorithm detects a manageable amount of false positives, requiring a reasonable amount of human designers' involvement for deciding whether a design pattern suggested by the algorithm should be applied or rejected. This outcome is in line with Yacoub and Ammar [38] who argue that "patterns are mental building blocks that are more related to human understanding than to automatic usage." The combination of the proposed design pattern models and the Pattern Candidate Finder algorithm may help users understand design problems and consequences of their solutions.

The contribution of the work is twofold. First, the suggested representation approach offers a comprehensive design pattern categorization schema, which includes creational design patterns, structural compositions, wrappers, and interactions. This categorization schema is intuitive and extensible, as it relies on common OPM constructs that are identified in the patterns' models. Furthermore, establishing the categorization on a simple visual feature of the patterns makes the schema comprehensible and usable as a design pattern catalog reference. Second, our approach makes the development of a design improver feasible, as it enables extracting the knowledge gained from modeling and analyzing design patterns to find specific location in a system model where design patterns can be embedded to improve the design. This mechanism may be helpful not only for novice designers, but also for experts who are not familiar with the ever-growing vocabulary of design patterns.

Possible extensions of our work include studying design patterns composition [6] and improving the Pattern Candidate Finder algorithm by incorporating semantic techniques or learning mechanisms, as presented in [28]. Additional research should further empirically test the value of our approach in terms of its cost-effectiveness by comparing standard designs to those done with our approach of enhancing the design with semi-automatically suggested design patterns.

References

1. Abdul Jalil, M., Azman Mohd Noah, S. "The difficulties of Using Design Patterns among Novices: An Exploratory Study". Proceedings of the 2007 International Conference on Computational Science and its Applications (ICCSA 2007), pp 97-103, 2007.
2. Batra, D. "Conceptual Data Modeling Patterns: Representation and Validation". Journal of Database Management (JDM) 16(2), pp. 84-106, 2005.
3. Blomqvist, E. "Fully Automatic Construction of Enterprise Ontologies Using Design Patterns: Initial Method and First Experiences". Proceedings of OTM 2005 Conferences, Ontologies, DataBases, and Applications of Semantics (ODBASE), Lectures Notes in Computer Science 3761, pp. 1314-1329, 2005.
4. Buschmann, F., Meunier, R. "A System of Patterns". In: Coplien, J. O. and Schmidt, D. C. (eds.): Pattern Language for Program Design, Addison-Wesley, pp. 325-343, 1995.

5. Buschmann, F., Meunier R., Rohnert, H., Sommerland, P., and Stal, M. Pattern-Oriented Software Architecture: a System of Patterns. Wiley (1996).
6. Dong, J., Alencar, P.S.C., Cowan, D.D., Yang, S. “Composing pattern-based components and verifying correctness”. The Journal of Systems and Software 80 (11), pp. 1755-1769, 2007.
7. Dong, J., Yang, S., Zhang, K. “Visualizing Design Patterns in their Applications and Composition”. IEEE Transactions on Software Engineering 33 (7), pp. 433-453, 2007.
8. Dori, D. Object-Process Methodology – A Holistic System Paradigm. Springer, 2002.
9. Eden, A.H. “LePUS: A Visual Formalism for Object-Oriented Architectures.” Proceedings of the 6th World Conference Integrated Design and Process Technology (IDPT'2002), 2002.
10. Eden, A.H. "Precise Specification of Design Patterns and Tool Support in Their Application". PhD thesis, University of Tel Aviv, 1999.
11. France, R.B., Kim, D.-K., Ghosh, S., and Song, E. “A UML-Based Pattern Specification Technique”, IEEE Transactions on Software Engineering, Vol. 30, No. 3, pp. 193-206, 2004.
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
13. Guennec, A.L., Sunye, G., Jezequel, J-M. "Precise Modeling of Design Patterns". Proceedings of the 3rd International Conference on UML (UML'2000), Lectures Notes in Computer Science (LNCS) 1939, Springer Verlag, pp. 482-496, 2000.
14. Hahsler, M. “A Quantitative Study of the Application of Design Patterns in Java”. Working Papers on Information Processing and Information Management Nr. 01/2003, Institute of Information Processing and Information Management, Wien, Wirtschaftsuniv, 2003.
15. Hahsler, M. “A Quantitative Study of the Adoption of Design Patterns by Open Source Software Developers”. Chapter in S. Koch (Ed.): Free/Open Source Software Development, Idea Group, Inc. 2004.
16. Helm, R. “Patterns in Practice", Proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (OOPSLA'2005), p. 337-341, 2005.
17. Kim. D. K., France R. B., and Ghosh, S. “A UML-based language for specifying domain-specific patterns”, Journal of Visual Languages and Computing, Vol. 15, No. 3-4, pp. 265-289, 2004.
18. Lauder, A., Kent, S. "Precise Visual Specification of Design Patterns". The 12th European Conference on Object-Oriented Programming (ECOOP'98), Lecture Notes in Computer Science 1445, pp. 114-136, 1998.

19. Mak, J.K.H., Choy, C.S.T., Lun, D.P.K. "Precise Modeling of Design Patterns in UML". Proceedings of the 26th International Conference on Software Engineering (ICSE'04), pp. 252-261, 2004.
20. Mapelsden, D., Hosking, J., Grundy, J. "Design Pattern Modeling and Instantiation using DPML". Proceeding of TOOLS Pacific 2002, Sydney, Australia. Conference in Research and Practice in Information Technology, 10. Noble, J. and Potter, J., Eds., ACS.
21. Noble, J. "Classifying relationships between Object-Oriented Design Patterns". Proceedings of the Australian Software Engineering Conference, pp. 98-108, 1998.
22. Object Management Group. UML 2.0 Superstructure FTF convenience document. http://www.omg.org/docs/ptc/04-10-02.zip
23. OMG, "Meta Object Facility (MOF™)". version 1.4, 2003, http://www.omg.org/docs/formal/02-04-03.pdf
24. OPCAT inc. OPCAT web site. http://www.objectprocess.org/
25. Object Venture Inc., Pattern and Component Markup Language, 2002, http://www.objectventure.com/pcml.html
26. Pickin, S., Manjarrés, A. "Describing AI Analysis Patterns with UML". Proceedings of the 3rd International Conference on UML, Lecture Notes in Computer Science 1939, pp. 466-481, 2000.
27. Prechelt, L., Unger, B., Philipssen, M., Tichy, W. "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance". IEEE Transactions of Software Engineering, pp. 595-606, June 2002.
28. Purao, S., Storey, V.C., and Han, T. "Improving Analysis Pattern Reuse in Conceptual Design: Augmenting Automated Processes with Supervised Learning". Information Systems Research 14 (3), pp. 244-268, 2003.
29. Reinhartz-Berger, I. and Dori, D. "A Reflective Metamodel of Object-Process Methodology: The System Modeling Building Blocks". In P. Green and M. Rosemann (Eds.): Business Systems Analysis with Ontologies. Idea Group, pp. 130-173, 2005.
30. Reinhartz-Berger, I. and Sturm, A. "Enhancing UML Models: A Domain Analysis Approach", Journal on Database Management (JDM) 19 (1), special issue on UML Topics, pp. 74-94, 2007.
31. Schmidt, D. "Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software". Communications of the ACM, Vol. 38, No. 10, October, 1995.
32. Shalloway, A., Trott, J. Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley, 2001.

33. Soffer, P. "Structural Equivalence in Model-Based Reuse: Overcoming Differences in Abstract Level". Journal of database management 16(3), pp. 21-39, 2005.
34. Sturm, A., Dori, D., Shehory, O. “Domain Modeling with Object-Process Methodology.” Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS'2006), pp. 144-151, 2006.
35. Taibi, T. and Ngo, D.C.L. "Formal Specification of Design Patterns – A Balanced Approach". Journal of Object Technology 2(4), 2003: 127-140.
36. Warmer, J. and Kleppe, A. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1998.
37. Wendorff, P. “Assessment of design patterns during software reengineering: Lessons learned from a large commercial project.” Proceedings of the 5th Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, pp. 77 – 84, 2001.
38. Yacoub, S. and Ammar, H. Pattern-oriented analysis and design: composing patterns to design software systems. Addison-Wesley, Boston, 2003.
39. Yin, R.K, “Case Study Reaserch. Design and Methods”, 1994, Second edition, Thousand Oaks: Sage.
40. Zimmer, W. “Relationships between Design Patterns”. In: Coplien, J. O. and Schmidt, D. C. (eds.): Pattern Language for Program Design, Addison-Wesely, pp. 345-364, 1995.

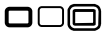
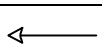
Appendix A: A Quick Guide to the Syntax and Semantics of OPM

Object-Process Methodology (OPM) [8, 29] is a holistic approach to the modeling, study, development, and evolution of systems. Structure and behavior coexist in the same OPM model to enhance the comprehension of the system as a whole. The OPM elements, which are summarized along with their symbols and meanings in Table 4, are entities and links. Entities generalize objects, processes, and states. Objects and processes, the two basic building blocks of an OPM system model, are first-order classes of things. Objects are things that exist, while processes are things that transform objects by creating or destroying them or by changing their states. Links, which connect entities, are structural or procedural. Structural links express static, structural relations between pairs of objects or processes. Aggregation, generalization, characterization, and instantiation are the four fundamental structural links in OPM. General structural relations can take on any semantics, which is expressed textually by their user-defined tags.

The behavior of a system is manifested in three major ways: (1) processes can transform (generate, consume, or change the state of) objects, (2) objects can enable processes without being transformed by them, and (3) objects can trigger events that invoke processes. These ways are specified using different types of OPM procedural links (see Table 4).

The complexity of an OPM model is controlled through three refinement/abstraction processes: *in-zooming/out-zooming*, in which the entity being refined is shown enclosing its constituent elements; *unfolding/folding*, in which the entity being refined is shown as the root of a directed graph; and *state expressing/suppressing*, which allows for showing or hiding the possible states of an object. These mechanisms enable the user to recursively specify and refine the system under development to any desired level of detail without losing legibility and comprehension of the complete system. Furthermore, these mechanisms include consistency rules among the different granularity and abstraction levels, so that a low-level OPM specification is consistent with the higher level specifications [8].

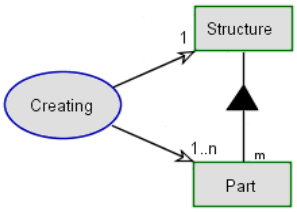
Table 4. OPM concepts, symbols, and meaning

Concept Name	Symbol	Concept Meaning
Informatical, systemic object		A piece of information
Informatical, environmental object		A piece of information which is external to the system
Process		A pattern of transformation that objects undergo
Initial/Regular/Final state		An initial/regular/final situation at which an object can exist for a period of time
Exhibition-Characterization		A fundamental structural relation representing that an element exhibits a thing (object/ process)
Aggregation-Participation		A fundamental structural relation representing that a thing (object/process) consists of one or more things
Generalization-Specialization		A fundamental structural relation representing that a thing is a subclass (refinement) of another thing (object/ process)
Classification-Instantiation		A fundamental structural relation representing that a thing (object/process) consists of one or more things
General structural link		A bidirectional or unidirectional association between things that holds for a period of time, possibly with a tag denoting the association semantics
Enabling event link		A link denoting an event (such as data change or an external event) which triggers (tries to activate) a process. Even if activated, the process does not change the triggering object.
Consumption event link		A link denoting an event which triggers (tries to activate) a process. If activated, the process consumes the triggering object.
Condition link		A link denoting a condition required for a process execution, which is checked when the process is triggered. If the condition does not hold, the next process (if any) tries to execute.
Agent link		A link denoting that a human agent (actor) is required for triggering a process execution
Instrument link		A link denoting that a process uses an object without changing it. If the object is not available (possibly in a specific state), the process waits for its availability.
Effect link		A link denoting that a process changes an object
Consumption/Result link		A link denoting that a process consumes/yields an object
Invocation link		A link denoting that a process triggers (invokes) another process when it ends
XOR relation		A connection between structural or procedural links denoting that exactly one of the links is applicable (i.e., active in a single instance of the object or the process)
OR relation		A connection between structural or procedural links denoting that at least one of the links is applicable (i.e., active in a single instance of the object or the process)

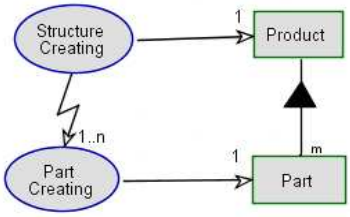
Appendix B: OPM models of Design Patterns

The Builder design pattern **Category: Creational**

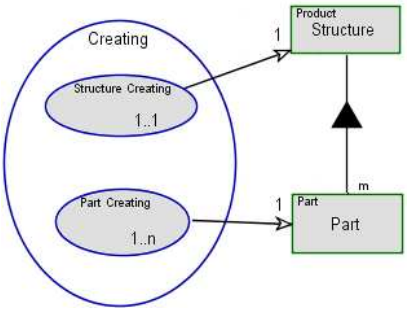
Problem model:



Solution model:

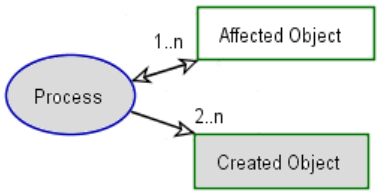


Correction model:

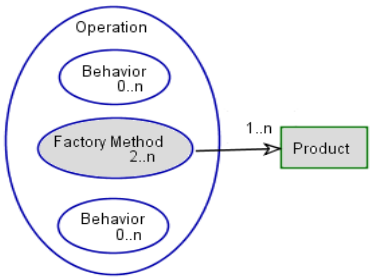


The Factory Method design pattern **Category: Creational**

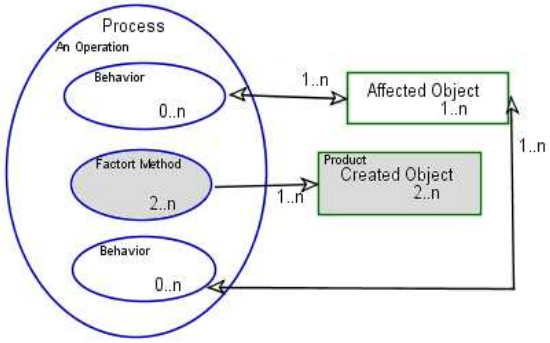
Problem model:

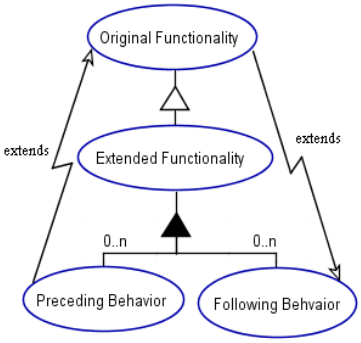
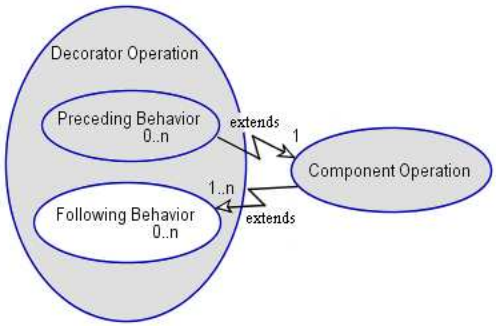
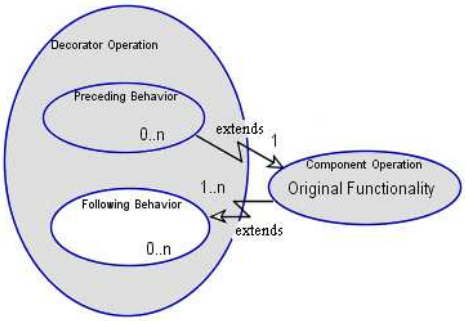


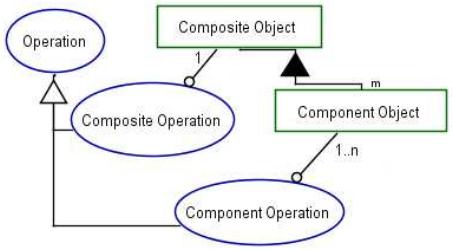
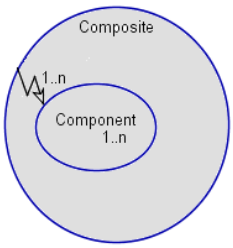
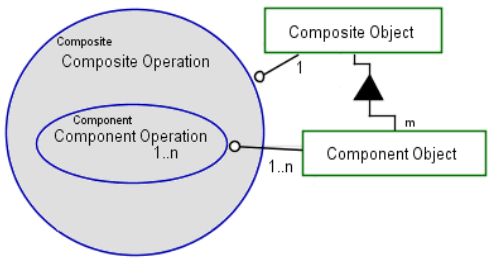
Solution model:

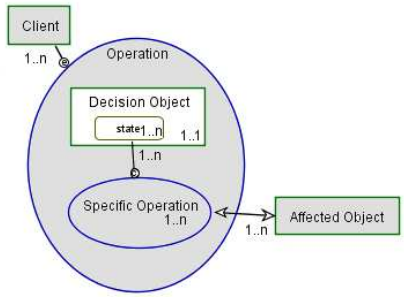
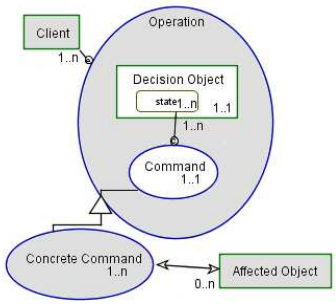
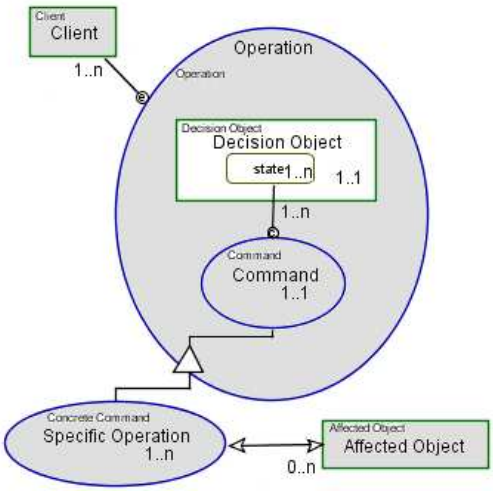


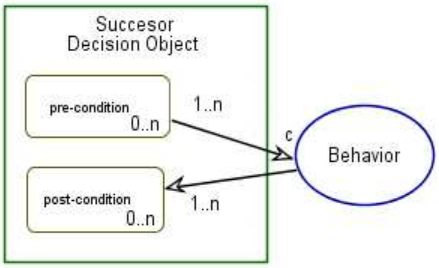
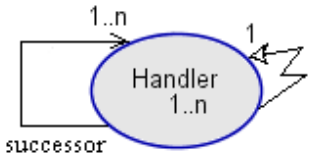
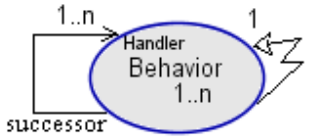
Correction model:



The Decorator design pattern	Category: Wrapper
Problem model: 	**Solution model:** 
Correction model: 	

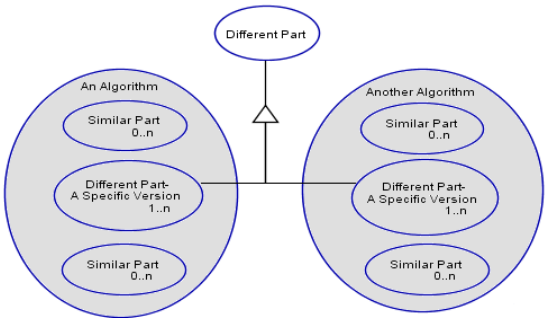
The Composite design pattern	Category: Structural composition
Problem model: 	**Solution model:** 
Correction model: 	

The Command design pattern	Category: Interaction
Problem model: 	**Solution model:** 
Correction model: 	

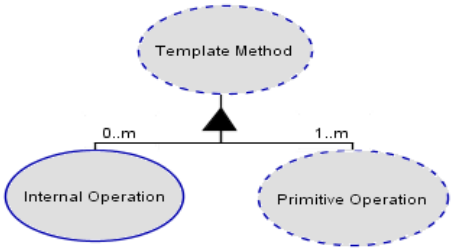
The Chain of Responsibility design pattern	Category: Structural composition
Problem model: 	**Solution model:** 
Correction model: 	

The Template Method design pattern **Category: Structural composition**

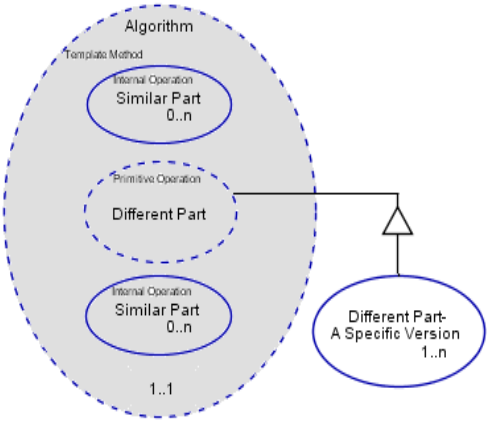
Problem model:



Solution model:



Correction model:



Appendix C: The ExamPal system model

The ExamPal system is an automatic application for composing, taking, checking, and grading analysis and design exams. The OPM design model of this system is given below. All candidates found by the Pattern Candidate Finder algorithm for the Command design pattern are marked by – – –, whereas all candidates for the Observer design pattern are marked by • • • • • • .

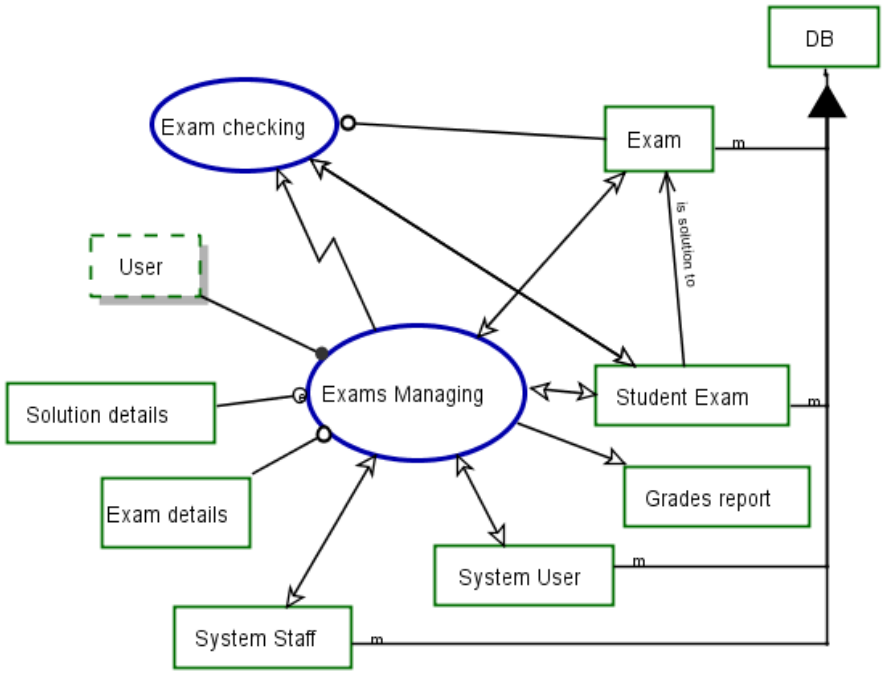


Figure 8. ExamPal top level diagram

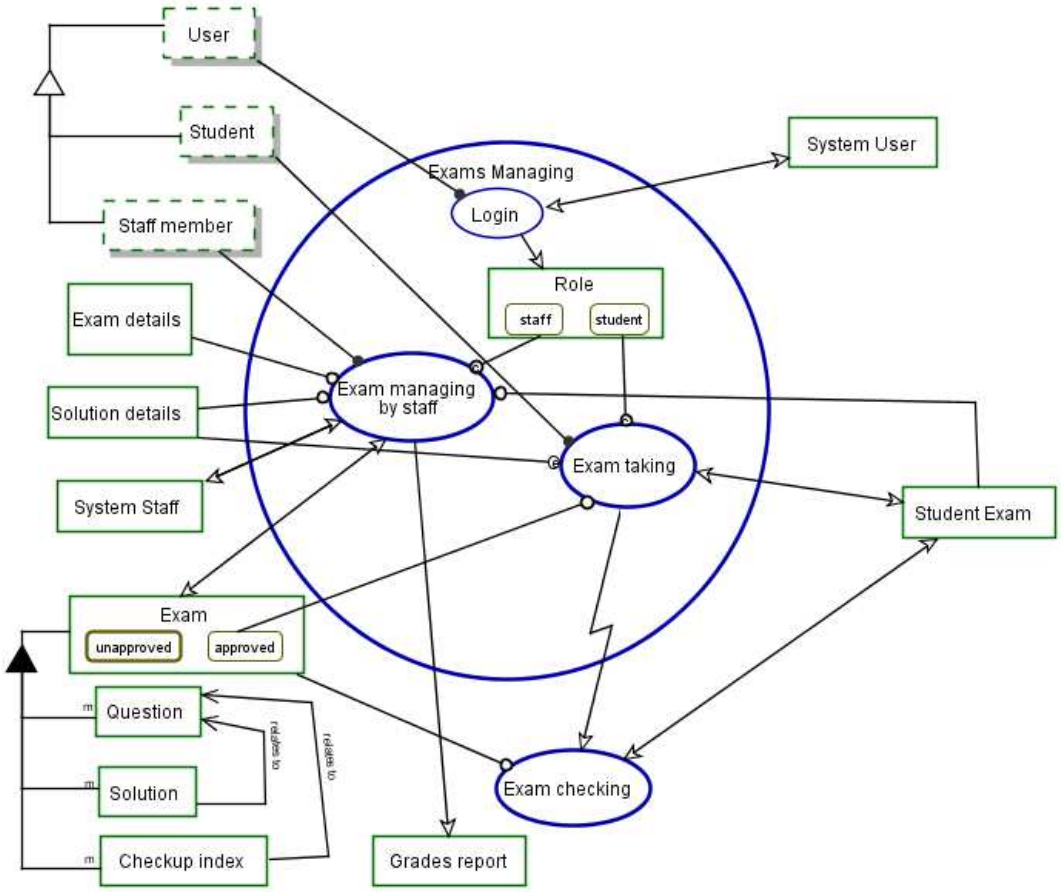


Figure 9. Exams Managing in-zoomed

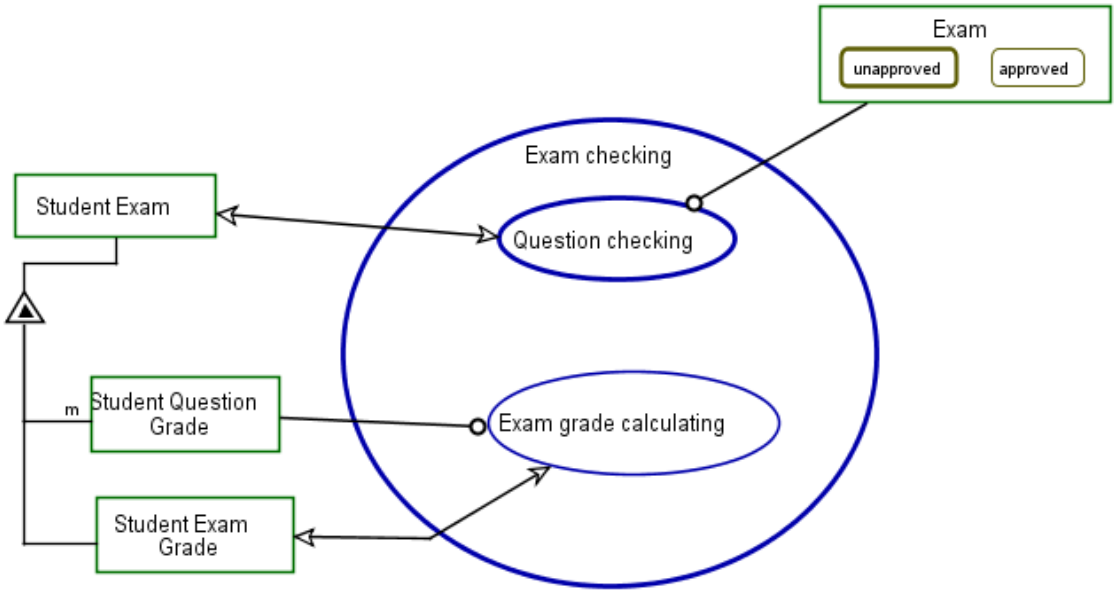


Figure 10. Exam checking in-zoomed

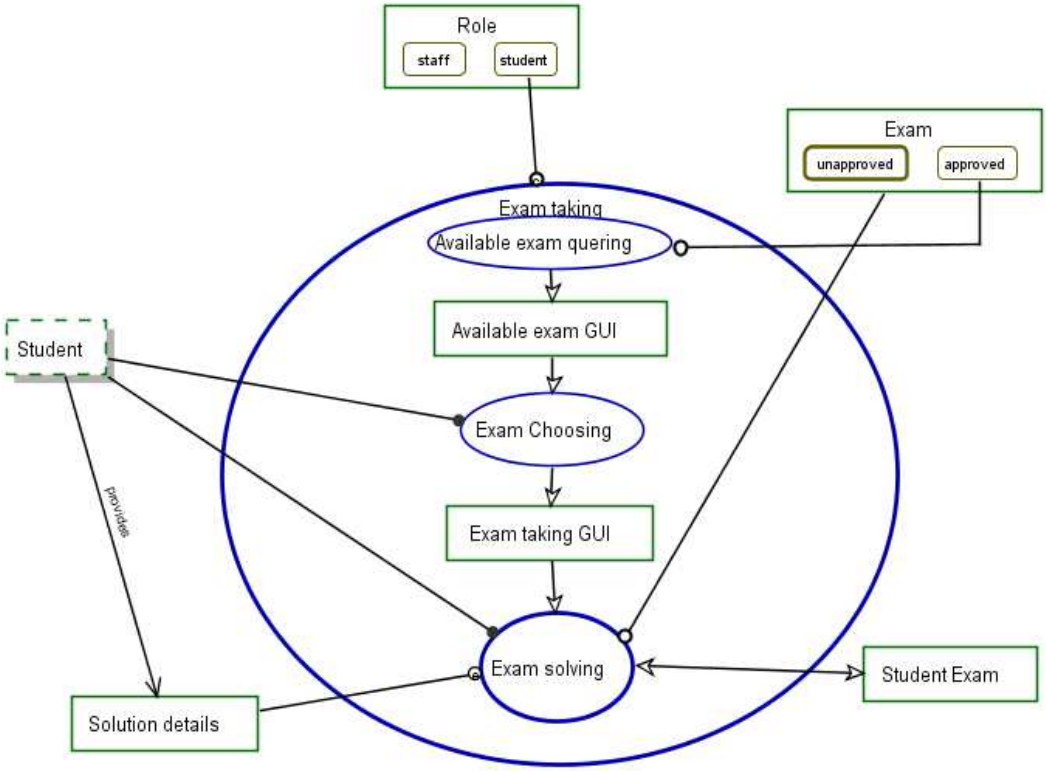


Figure 11. Exam taking in-zoomed

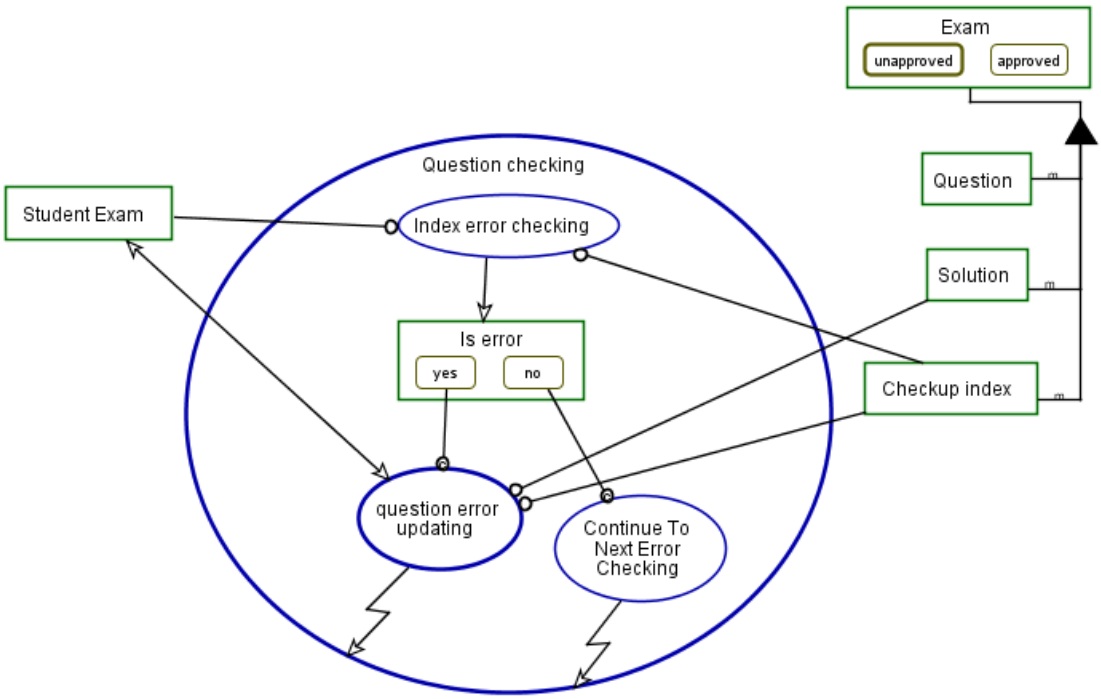


Figure 12. Question checking in-zoomed

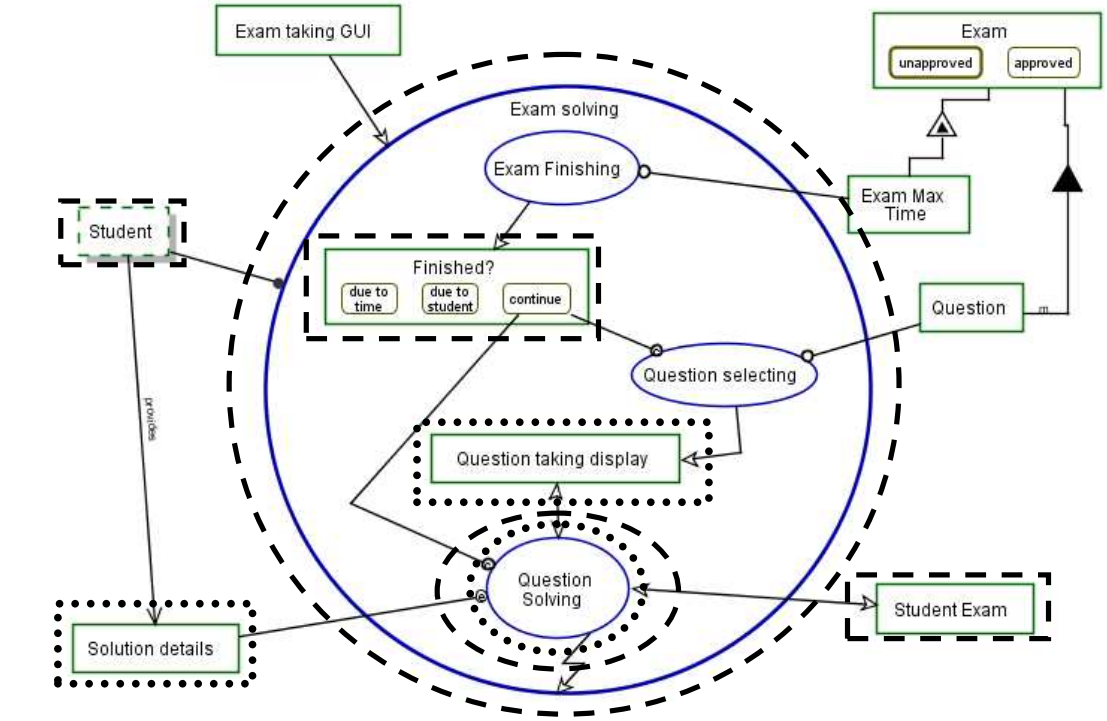


Figure 13. Exam solving in-zoomed

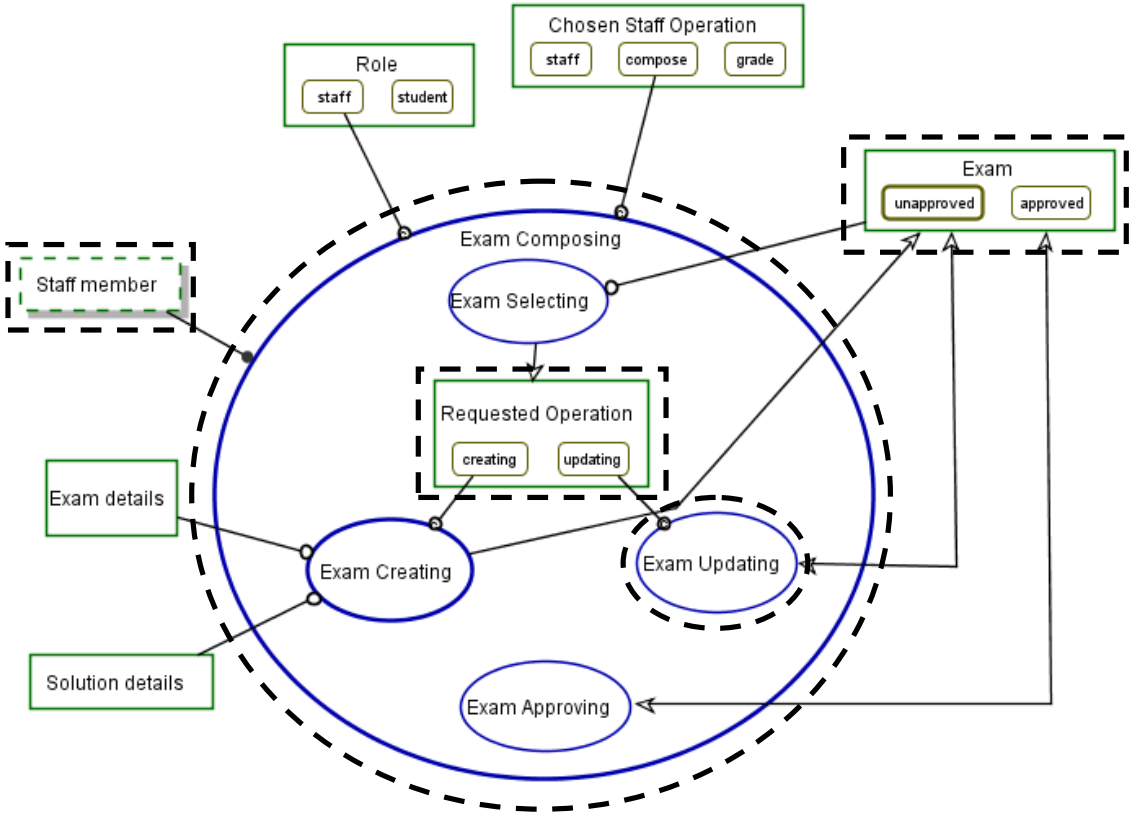


Figure 14. Exam composing in-zoomed

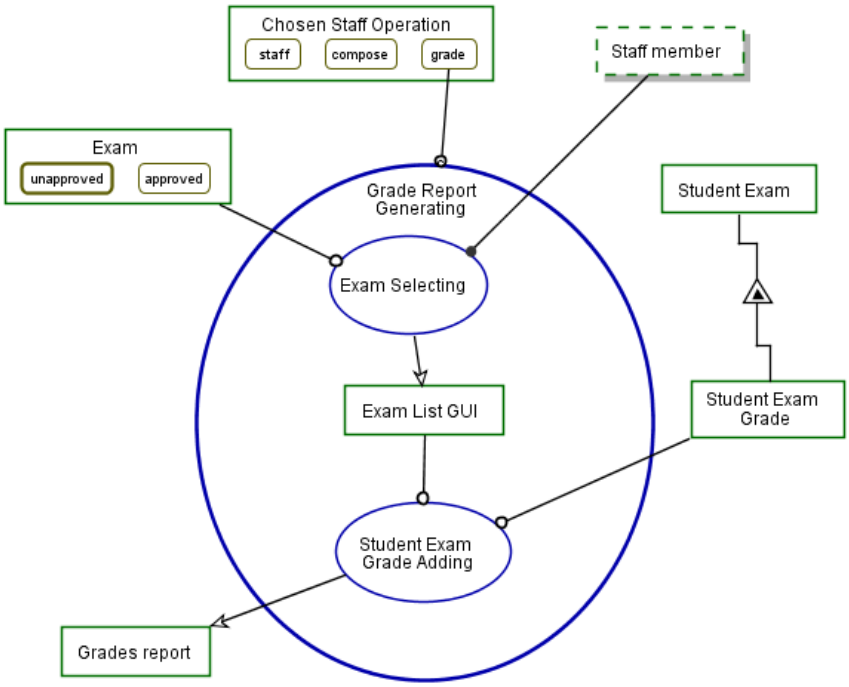


Figure 15. Grade Report Generating in-zoomed

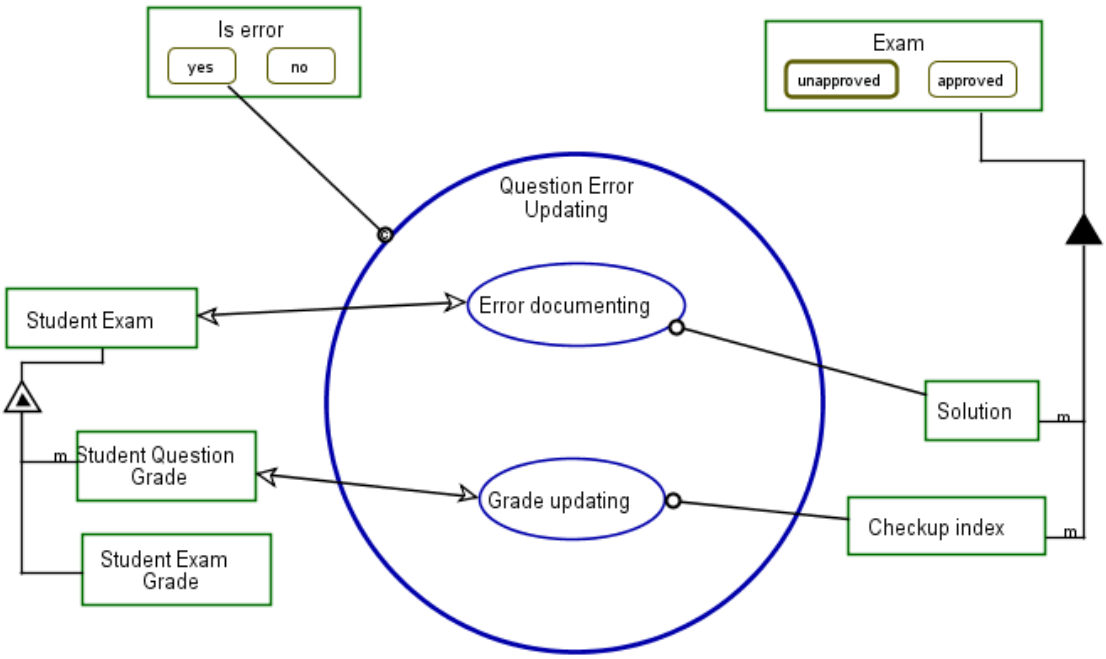


Figure 16. Qestion Error Updating in-zoomed

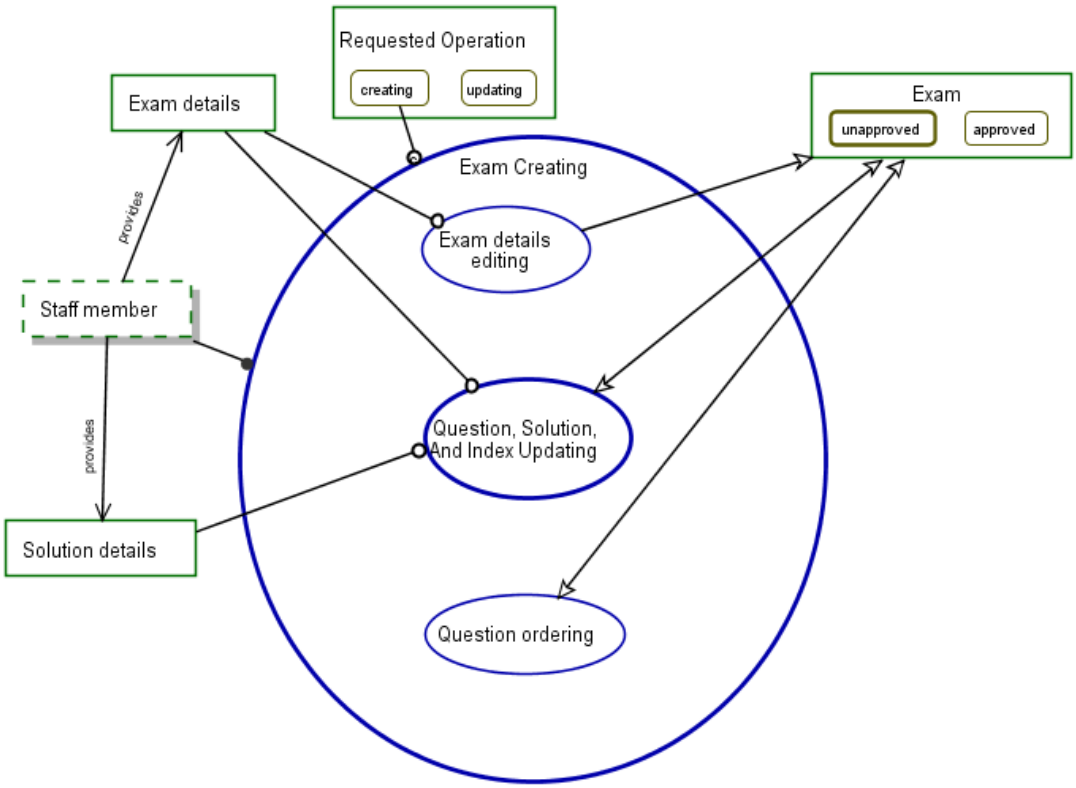


Figure 17. Exam Creating in-zoomed

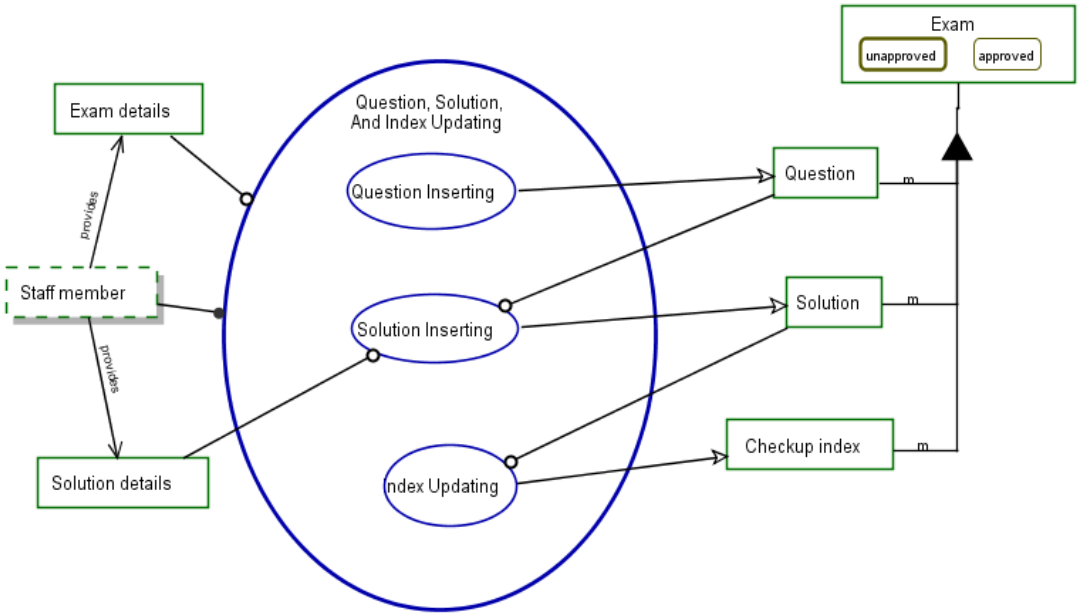


Figure 18. Question, Solution, And Index Updating in-zoomed