Analyzing Object-Oriented Design Patterns from an Object-Process Viewpoint

Galia Shlezinger[1], Iris Reinhartz-Berger[2], Dov Dori[1]

[1]Faculty of Industrial Engineering and Management,
Technion-Israel Institute of Technology, Haifa 32000, Israel
galias@tx.technion.ac.il, dori@ie.technion.ac.il

[2]Department of Management Information Systems,
University of Haifa, Haifa 31905, Israel
iris@mis.haifa.ac.il

Abstract. Design patterns are reusable proven solutions to frequently used design problems. To encourage software engineers to use design patterns effectively and correctly throughout the development process, design patterns should be classified and represented formally. In this paper, we apply Object Process Methodology (OPM) for representing and classifying design patterns. OPM enables concurrent representation of the structural and behavioral aspects of design patterns in a single and coherent view. Comparing OPM and UML models of twelve poplar design patterns, we found that the OPM models are more compact, comprehensible and expressive than their UML counterparts. Furthermore, the OPM models induce a straightforward classification of these design patterns into four groups: creational, structural composition, wrapper, and interaction design patterns.

1 Introduction

One of the biggest threats to software reliability is maintenance. Maintenance of a system might eventually reach the point of extending and even altering the system. The changes need not necessarily be large scale and need not require redesign or even reanalysis of the complete system; even small modifications, which seem harmless, may seriously impair the reliability of the system, either individually or in combination with other changes. Furthermore, the anxiety software engineers experience when modifying a reliably working system due to newly introduced malfunctions often causes them to prefer adding new elements over changing existing ones. More often than not, such additions unnecessarily complicate the system model and implementation, making its maintenance even harder. In order to avoid this tendency towards "system model entropy", reuse approaches suggest different concepts for reusable artifacts: *components*, which are self-contained ready-to-use building blocks, *frameworks*, which are skeletal system parts to be adapted by the system developer, and *patterns* that describe generic solutions for recurring problems to be customized for a particular context.

Design patterns have attracted the interest of researchers and practitioners as proven reusable solutions to frequently occurring design problems. Shalloway and Trott [29] suggested several reasons for using, studying, and dealing with design patterns, including the reuse of existing, high-quality solutions, establishing common terminology to improve communications within teams, shifting the level of thinking to a more abstract perspective, deciding whether the design is correct (and not just happens to work), improving the code modifiability, and discovering alternatives to large inheritance hierarchies. However, deploying these solutions to develop complex information systems is a tedious task that involves integration issues and iterative development. It is, hence, important to describe design patterns, the problems they intend to solve, the context in which they can be reused, and their consequences in a formal, unambiguous way that can be easily understood by designers. As discussed in [32], the amount of "understandability" is directly derived from the similarity between the "system model" that reflects the designer perspective and the "mental model" that describes the user or client viewpoint. The closer the system model is to the mental model, the more understandable the system is to both designers and clients.

The increasing number of design patterns that have been proposed and published over the last decade emphasizes the need for a good design patterns representation language, as well as a framework for organizing, classifying, categorizing, and evaluating design patterns, which will help designers in choosing and implementing the correct solutions to the problems at hand. In this work, we suggest

Object Process Methodology (OPM) [8] for both purposes. OPM, which supports two types of equally important elements, objects and processes, enables the representation of both structural and behavioral aspects of design patterns in a single visual and coherent view. Furthermore, the OPM design pattern models naturally lend themselves to a clear, useful classification. This classification is directly derived from the OPM models and does not require justifications and further explanations, as is the case with many object-oriented design pattern classification schemes. The contribution of our work is therefore two-fold: First, we apply OPM to model and portray the essence of design patterns in a more direct, complete, and comprehensible way than what can be done using object-oriented languages. The completeness of the pattern models is due to OPM's ability to represent both the structure and the behavior of the patterns. The comprehension of the pattern models is due to the high level of similarity between the patterns' system and mental models with OPM as the modeling approach. Secondly, the categories of design patterns defined in this work are solidly grounded by distinctive characteristics of their respective OPM models. Based on this classification, design patterns can be used regardless of the chosen modeling language.

The rest of the paper is structured as follows. In Section 2, we review work relevant to design pattern languages and classification frameworks. Section 3 provides a short overview of Object Process Methodology (OPM), while Section 4 presents OPM models of several design patterns, making the point that these models induce an accurate design pattern classification scheme. An example for using design patterns in an OPM model of an application is presented in Section 5. Section 6 concludes and refers to the future research plan.

2 Motivation and Background

Design patterns provide solutions for recurring design problems. The idea of documenting design solutions as patterns is attributed to the American architect Christopher Alexander [1]. Applying his idea to object-oriented design and programming, design patterns are usually described using a template. The template used by the Gang of Four (GoF) in [15], for example, consists of *pattern name and classification, intent (motivation), applicability, structure, participants, collaboration, consequences, implementation, sample code, known uses,* and *related patterns.* Often the structure of the design pattern is illustrated by a graphical representation, such as Object Modeling Technique (OMT) [28] or Unified Modeling Language (UML) [23]. These representations are accompanied by brief (textual) descriptions of the basic elements (i.e., classes) that compose the design pattern. The behavioral aspect of the design pattern usually gets much less attention, and is sometimes described informally in text or through partial diagrams that specify the collaboration between the various elements. Such semi-formal representations of design patterns hinder their rigorous, systematic use.

Sets of design patterns have been developed for diverse computer-related disciplines, including distributed processing [5], reactive systems [21], and human-computer interaction design [17]. Different languages have been proposed to represent design patterns formally. Some employ mathematical descriptions, while others suggest visual representations or markup languages. The Language for Patterns Uniform Specification (LePUS) [11], for example, is a formal specification language based on a theory of object-oriented design in mathematical logic. LePUS is used to specify design patterns, class libraries, and object-oriented frameworks. It defines several building blocks, including classes, variables and relationships between them, and introduces a visual language to represent them. However, LePUS abstractions are difficult for average designers to work with. This problem is common to most of the solutions that use formal mathematical descriptions, as they require a fair amount of mathematical skills.

Several languages for describing design patterns are based on UML or its extensions [7, 13, 19, 20, 31]. France et al. [13], for example, presented a UML-based specification technique, which is composed of static and interactive pattern specifications. The Static Pattern Specification (SPS) contains classifiers and relationships among them, as well as constraint templates expressed in Object Constraint Language (OCL) [33]. The Interactive Pattern Specification (IPS) contains an interaction role, which consists of a lifeline and messages. These two types of specifications must be consistent with each other in order to coherently describe the complete design patterns. Generally, consistency and integrity are Achilles' heel of UML [24, 26].

Markup languages, such as XML and OWL [6, 25], are also used for describing design patterns. While this approach may be very useful for building tools that support design patterns, markup languages are machine-understandable at the expense of being cryptic to humans [9].

Since the number of design patterns is continuously increasing, there is a growing need not only to formally describe the different aspects of design patterns, but also to organize and classify them according to their purpose or structure. Noble and Zimmer, for example, have classified relationships among design patterns that are mostly hierarchical. Noble [22] divided relationships among design patterns to *primary relationships,* which are *"uses", "refines",* and *"conflicts"*, and *secondary relationships*, which are *"used by", "refined by", "variant", "variant uses", "similar", "combines", "requires"*, *"tiling"*, and *"sequence of elaboration".* While primary relationships are the main relationships identified, secondary relationships are either inverses of primary relationships or more complex relationships between patterns. Zimmer [34] defined three categories of relationships: *"x uses y in its solution", "x is similar to y",* and *"x can be combined with y".* This categorization of relations brought him to the conclusion that the Factory Method design pattern is not really a pattern, but rather a manifestation of the "X uses Y" relationship between two other design patterns: Abstract Factory and Template Method. He also suggested modeling behaviors as objects and introduced a new design pattern, called Objectifying Behavior, for this purpose.

Claiming that design patterns are not independent of programming languages, Gil et al. [16] presented a taxonomy of design patterns according to which they can be divided into three categories: *clichés*, which are low level patterns, *idioms*, which are patterns mimicking features found in other languages, and *cadets*, which are patterns that are candidates for becoming language features. Cadets are further divided into *relators* and *architects*. Relators describe relationships between a small number of entities, while architects involve many entities.

The GoF [14, 15] have used two dimensions for classifying design patterns: *scope*, which specifies whether the pattern applies to classes or objects, and *purpose*, which reflects the intension of the patterns. According to the purpose dimension, a design pattern can be creational, structural, or behavioral. *Creational patterns* are concerned with instantiating objects, *structural patterns* deal with composition of classes or objects, and *behavioral patterns* describe the way in which objects and classes interact and distribute responsibilities.

Another design pattern classification scheme [2] organizes patterns according to their granularity, functionality, and structural principles. *Granularity* refers to three abstraction levels with which the design pattern is concerned: *architectural frameworks*, which have to do with the basic structure of applications, *design patterns*, which apply to a lower level of abstraction than whole system architecture, and *idiom*s, which are concrete realization and implementation of particular design issues. The pattern's *functionality* can be one of *creation, communication, access,* or *organizing the computation of complex tasks*. Finally, the patterns are also categorized according to four structural principles: *abstraction, encapsulation, separation of concerns,* and *coupling and cohesion.* According to this categorization, design patterns are only one group of patterns that belong to a specific level of granularity. Design patterns were further divided into *structural decomposition, organization of work, access control, management,* and *communication* [3].

More purpose-related classification schemes also exist. Hanemman et al. [18], for example, classified design patterns according to their usage of roles, since they found out that this feature helps using design patterns in an aspect-oriented design.

Since the classification schemes of design patterns are basically object-oriented, their categorization, justified primarily with objects in mind, is often non-convincing and inconclusive. Adopting Object-Process Methodology, which departs significantly form the object-oriented paradigm, our approach to modeling and categorizing design patterns is fundamentally different. OPM provides a comprehensible and expressive language for specifying design patterns. Indeed, as we show below, the resulting OPM models of many design patterns, especially those involving dynamic aspects, are more comprehensible, compact, and straightforward than their counterpart UML models. Analyzing the object-oriented GoF classification of design patterns [15] with respect to their OPM design pattern models, we offer a significantly improved classification scheme.

3 Object-Process Methodology

Object Process Methodology (OPM) is an integrated modeling paradigm to the development of systems in general and information systems in particular. Defining two types of equally important things, objects and stand-alone processes, object and process classes differ in the values of their perseverance attribute: the perseverance value of object classes is static, while the perseverance value of process classes is dynamic. The combination of objects and processes in the same single diagram type (called Object-Process Diagram, OPD) clarifies the two most important aspects that any system features:

structure and behavior, enabling the specification of system's structure and behavior in a single view. OPM supports the specification of structural and procedural links between things. Structural links express static relations such as aggregation-participation and generalization-specialization between pairs of things with the same perseverance (two objects or two processes). Procedural links, on the other hand, connect things with complementary perseverance (an object and a process) to describe the behavior of a system, i.e., how processes transform, use, and are triggered by objects. More about OPM can be found at [8].

The synergy of structure and behavior in the same model makes OPM suitable for modeling design patterns, especially those involving dynamic aspects. Table 1 summarizes the OPM elements used in this paper.

Table 1. Main OPM elements, their symbols, and semantics

Element Name	**Symbol**	**Semantics**
Object	▭	A thing that has the potential of unconditional existence
Process	⬭	A pattern of transformation that objects undergo
Instrument link	—○	A procedural link indicating that a process requires an unaffected object (input) for its execution
Effect link	⟷	A procedural link indicating that a process changes an object
Event link	—●	A procedural link indicating that a process is activated by an event (initiated by an object)
Invocation link	↯	A procedural link indicating that a process invokes another process
Structural Link	→	A structural relation between objects
Aggregation-Participation	▲	A structural relation which denotes that a thing (object or process) consists of other things
Exhibition-Characterization	◬	A structural relation representing that a thing (object or process) exhibits another thing
Generalization-Specialization	△	A structural relation representing that a thing is a sub-class of another thing

4 Classification of Design Patterns

In this section we examine the rationale behind the GoF classification of design patterns in terms of OPM.

4.1 Creational Design Patterns

Creational design patterns relate to class instantiation. They can be further divided into class-creational and object-creational patterns. Class-creational design patterns use inheritance in the instantiation process, while object-creational patterns use delegation to get the job done. *Abstract Factory* and *Singleton* are examples of class-creational design patterns. *Factory Method*, *Prototype*, and *Builder* are examples of object-creational design patterns. Figures 1-3 respectively describe three creational design patterns: *Abstract Factory*, *Factory Method*, and *Builder*. Each description includes the pattern's problem definition, suggested solution, a UML model (from [15]) and an OPM model with its explanation.

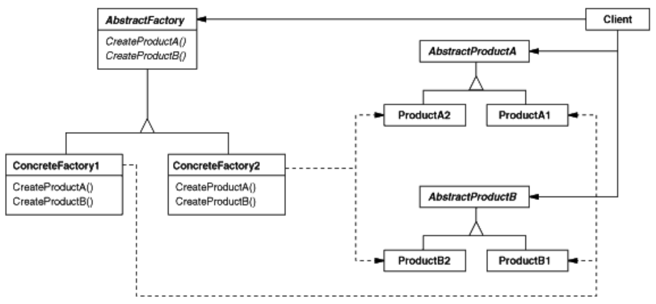
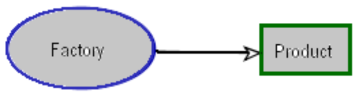
ABSTRACT FACTORY	
Problem Definition: Different families of related things should be created in different cases.	**Suggested Solution:** Separating the logic of how to create these families ("the factory") from the products themselves.
UML Model:  AbstractFactory CreateProductA() CreateProductB() Client AbstractProductA ProductA2 ProductA1 ConcreteFactory1 CreateProductA() CreateProductB() ConcreteFactory2 CreateProductA() CreateProductB() AbstractProductB ProductB2 ProductB1 From [15], p. 88.	**OPM Model:**  The "Factory" process, which holds the knowledge of how to create a product, creates the "Product" object.

Fig. 1. The Abstract Factory design pattern

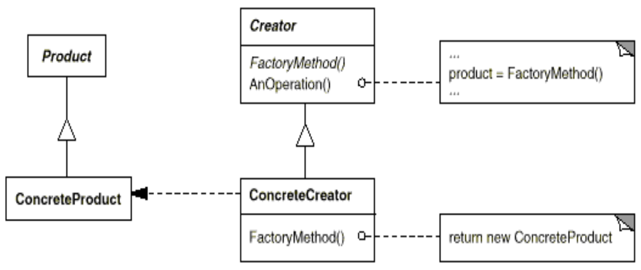
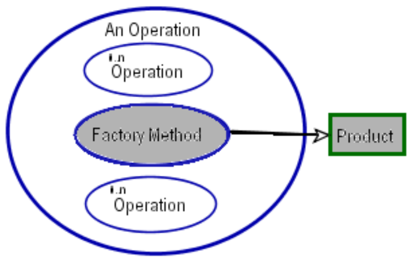
FACTORY METHOD	
Problem Definition: A creation algorithm is common to entities of several types.	**Suggested Solution:** Decouple the algorithm for creating an entity from the actual type of the entity being created.
UML Model:  Product Creator FactoryMethod() AnOperation() ... product = FactoryMethod() ... ConcreteProduct ConcreteCreator FactoryMethod() return new ConcreteProduct From [15], p. 108.	**OPM Model:**  An Operation 1..n Operation Factory Method Product 1..n Operation "Factory Method" is a process which represents an algorithm for creating "Products". This algorithm can be embedded in another process, called "An Operation", with any number of operations preceding and succeeding it.

Fig. 2. The Factory Method design pattern

As the models in these three figures demonstrate, the UML and OPM models of the design patterns differ in both orientation and abstraction level. While the UML models are object-oriented, i.e., they comprise object classes only, the OPM models are composed of both object classes and process classes, representing things with static and dynamic perseverance, respectively. Thanks to the notion of stand-alone processes, the OPM design pattern models do not require supplementary object classes which are used only as owners of methods (or operations) or as containers of other classes.

To demonstrate this basic difference, consider, for example, the "Abstract Factory" design pattern in Figure 1. The "Abstract Factory" in the UML model exists mainly as a container for the operations it exhibits. Zimmer [34] names this phenomenon as "objectifying behavior". OPM avoids these unnecessary classes by recognizing process classes as "first-class citizens", which exist in the model as equal partners to object classes rather than being necessarily operations underneath object classes. In the OPM model of the Abstract Factory design pattern (Figure 1), "Factory" is a process rather than an operation of a class, enabling a drastic reduction of the UML model.

BUILDER	
Problem Definition: One construction sequence may construct different kinds of products, some of which may be complex.	**Suggested Solution:** Separating the construction of a product from its representation and internal structure.

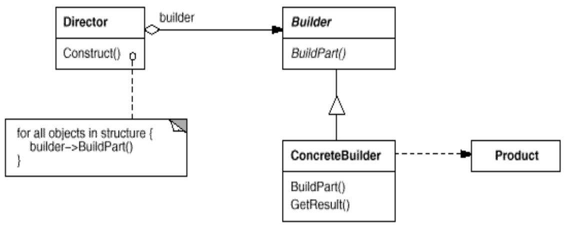
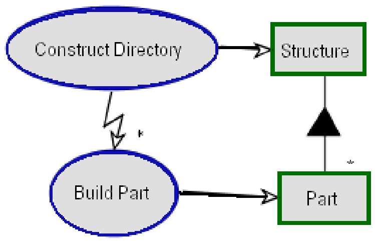
UML Model:
 Director
Construct()
builder
Builder
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
BuildPart()
GetResult()
Product
From [15], p. 98.
	OPM Model:
	
	The process "Construct Directory" directs the construction of a "Structure" which is composed of "Parts". These "Parts" are created by "Build Part" processes which are invoked by the "Construct Directory".

Fig. 3. The Builder design pattern

The ability of OPM to concurrently model both structure and behavior enables modeling the design patterns more accurately, more succinctly, and more formally. The UML model of Factory Method design pattern, for example, requires specification of calling the factory method from an operation and indicating by informal notes that the factory method returns a concrete product. The OPM model, on the other hand, specifies these requirements formally by using multiplicity constraints (zero to many, as discussed next) on the number of operations before and after the Factory Method, and a result link, indicating that the Factory Method generates and returns Product. Multiplicity constraints in OPM can be associated not only to relations but also to object and process classes. A multiplicity constraint on a thing (object or process) in a design pattern model indicates how many occurrences of this thing can appear in an application that implements the design pattern. In our case, the process "Operation" can appear zero or more times, as indicated by the "0..n" label at the upper left corner of the process ellipse symbol, implying that the "Factory Method" is called from somewhere within the process called "An Operation", including its very first or last operation. The OPM model of the Abstract Factory pattern also demonstrates another benefit of OPM: one can specify the details in an OPM model without loosing the "big picture" of the system or pattern being modeled. This is achieved via the scaling mechanisms that are built into OPM for enabling abstraction and refinement of things. The mechanism used in this case is in-zooming, in which an entity is shown enclosing its constituent elements. The vertical axis is the time line, so within an in-zoomed process it defines the execution order of the subprocesses, such that subprocesses that need to be executed in a sequence are depicted stacked on top of each other with the earlier process on top of a later one. "An Operation" is in-zoomed here to display its subprocesses: first a set of zero or more operations is performed, then the "Factory Method" is activated (creating Products), and finally another set of 0 or more operations is performed.

The second difference between the UML and OPM design pattern models is in their support of abstraction levels. While in the UML models both the abstract and concrete classes appear in the models, in the OPM models only the "abstract" classes appear, while the concrete classes that implement the abstract ones appear only in the actual models that make use of the design pattern. In these concrete application models, the application elements will be classified according to the design pattern elements using a mechanism similar to UML stereotyping. This separation of the design pattern models from the application models results in more abstract, formal, compact, and comprehensible design pattern models, which define only the necessary guidelines for and constraints on the application model. The OPM design pattern models can therefore be considered as templates for applications that make use of these design patterns as guiding metamodels taken from some meta-library. We elaborate on this aspect of OPM in section 5.

Studying the OPM models of the creational design patterns reported in [15] clearly shows that the basic idea behind creational design patterns is separating the construction logic from the objects. The construction logic can be specified as a process that creates the required object(s) as denoted by the result link (the procedural link with the closed arrowhead) from the process to the object. This recurring **process – result link – object** pattern is distinguishable in the OPM models in figures 1-3 by the pertinent object and process being marked in grey and with thick lines. This pattern indeed justifies the name of this group of design patterns as (object- or class-) creational.

4.2 Structural Design Patterns

Structural design patterns relate to class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality. Examples of design patterns in this category are *Bridge*, *Decorator*, *Flyweight*, *Adapter*, *Composite*, *Façade*, and *Proxy*. Figures 4-7 respectively describe four structural design patterns: *Bridge*, *Decorator*, *Adapter*, and *Composite.* Each description includes the pattern's problem definition, suggested solution, a UML model (from [15]) and an OPM model with its explanation.

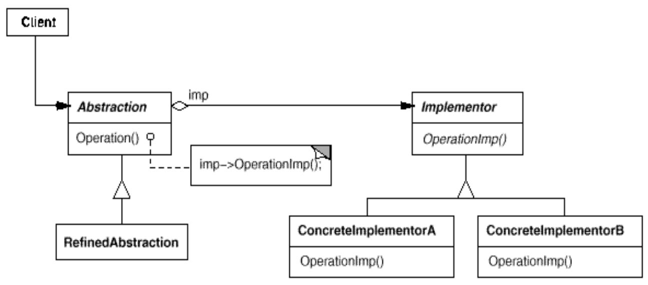
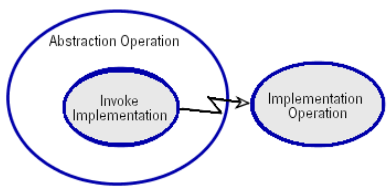
BRIDGE	
Problem Definition: An abstraction may have several possible implementations. We might want to maintain the abstraction and the implementations independent of one another.	**Suggested Solution:** Maintain the abstraction as an independent entity.
UML Model:  Client Abstraction imp Implementor Operation() OperationImp() imp->OperationImp(); RefinedAbstraction ConcreteImplementorA OperationImp() ConcreteImplementorB OperationImp() From [15], p.153.	**OPM Model:**  Abstraction Operation Invoke Implementation Implementation Operation "Abstraction Operation" is an abstract process which invokes the specific "Implementation Operation" via the subprocess "Invoke Implementation".

Fig. 4. The Bridge design pattern

The design patterns in this group further emphasize and clarify the fundamental differences between the UML and OPM design pattern models: All the constraints which are specified in the UML models as notes (in plain text) are expressed formally in the OPM models. The Decorator design pattern in Figure 5, for example, requires adding behavior to a component. This addition is specified in the UML model as a note in which the decorator operation (which includes the component operation) is first called and then followed by the added behavior. However, this is only one possibility defined in the design pattern problem section, which reads: "There is a need to add functionality that **precedes or follows** a basic functionality…" The OPM model enables any number of operations **before and after** the basic functionality, which is called in the model "Component Operation".

Another example of the expressiveness of OPM is in the Adapter design pattern in Figure 6. The OPM model of this design pattern explicitly refers to the existence of an interface which should be adapted in order to fit to a specific request. The UML model of this design pattern, on the other hand, refers only to the need to define an "Adapter" which wraps the "Adaptee".

The UML and OPM models of the Composite design pattern are completely different. The UML model uses an aggregation relation between "Composite" and "Component", while in the OPM model, "Composite" is zoomed into "Components". However, zooming into an OPM process has both structural aspects (containment, part-whole relations), and procedural aspects (execution order). Furthermore, we have found that the inheritance relation between "Composite" and "Component" in the UML model of this pattern is redundant in the OPM model, since its only purpose is to express the fact that both are processes. In the context of the design pattern, they do not share attributes, operations, or relations. OPM enables specifying this fact simply by using two types of classes: objects and processes.

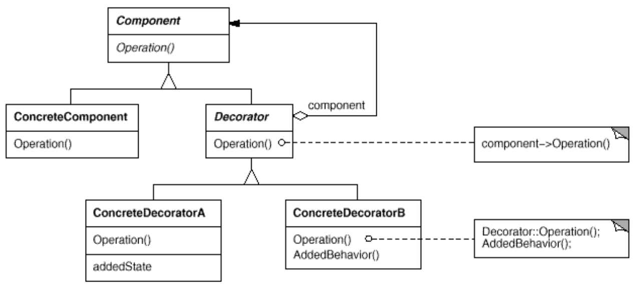
The recurrent pattern in the OPM models of this group of design patterns is **process – invocation link – process**. Another observation is that all the design patterns models contain a process which is further zoomed into subprocesses, one of which invokes another process. Although on the surface this pattern should be classified as behavioral rather than structural, a deeper look at the OPM models shows that they basically define the structure of components, each of which is a process. The OPM models of the Factory Method and Decorator design patterns reinforce this observation. The two design patterns are similar in that they both have an operation (performing some function) that may be preceded and/or followed by any number of other operations. However, the Factory Method emphasizes the creation of a product, while the Decorator focuses on the separation between the basic functionality—the "Component Operation" and the operation that uses it—the "Decorator Operation".

DECORATOR

Problem Definition:
There is a need to add functionality that precedes or follows a basic functionality without using sub-classing.

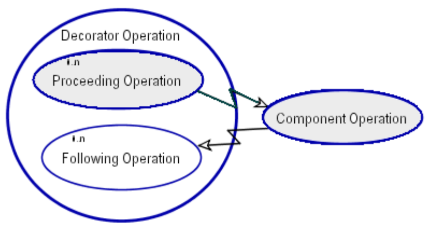
Suggested Solution:
Add the functionality as an independent entity.

UML Model:



From [15], p. 177.

OPM Model:



The "Decorator Operation" process contains subprocesses which may proceed or follow the process "Component Operation". The last proceeding subprocess invokes "Component Operation" which invokes the first "Following Operation".

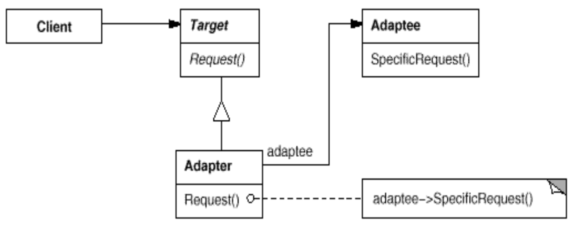
Fig. 5. The Decorator design pattern

ADAPTER (the object version)[1]

Problem Definition:
A component has a desired data and behavior but its interface does not match the required interface.

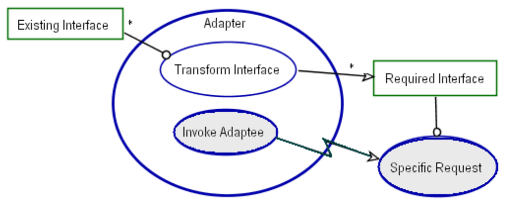
Suggested Solution:
Wrapping the existing interface by an adapter interface.

UML Model:



From [15], page 141.

OPM Model:



The "Adapter" process is composed of "Transform Interface" which transforms "Existing Interfaces" into "Required Interfaces". This subprocess is followed by a subprocess which invokes "Specific Request" which uses the "Required Interface" to carry out the process.

Fig. 6. The Adapter design pattern

[1] The adapter design pattern has two possible object-oriented versions: the class adapter and the object adapter. They differ in the method of implementation: the class adapter uses multiple-inheritance to solve the problem, while the object adapter uses object referencing. In OPM we do not need to maintain these two versions, since the Adapter is modeled as a stand alone process that can access the adaptee (see the figure).

COMPOSITE	
Problem Definition: A system contains simple components which can be grouped to build composite components. Other parts of the application should not be aware whether a component is composite or not.	**Suggested Solution:** Compose entities into tree structures to represent part-whole hierarchies.

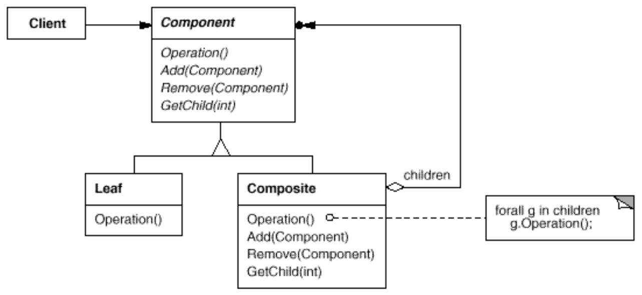
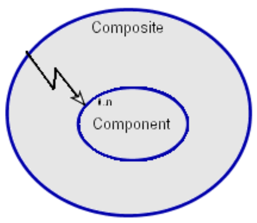
UML Model:
 Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();
From [15], p. 164.
	OPM Model:
	
	The "Composite" process contains any number of "Component" subprocesses which are also invoked by it.

Fig. 7. The Composite design pattern

The models of the Decorator, Adapter, and Bridge design patterns have another common structural element: the invoked process is external to the in-zoomed process. This common element justifies their classification by several design pattern classification schemes, including [14], [15], and [29], as *wrappers*.

4.3 Behavioral Design Patterns

Behavioral design patterns define algorithms and object responsibilities. They also help design communication modes and interconnections between different classes and objects. Design patterns in this category include *Command*, *Iterator*, *Memento*, *Mediator*, *Interpreter*, *Chain of Responsibility*, *Observer*, *Template Method, State, Strategy, and Visitor*. Figures 8-12 respectively describe five behavioral design patterns: *Command*, *Chain of Responsibility*, *Observer*, *Template Method*, and *Mediator.* Each description includes the pattern's problem definition, suggested solution, a UML model (from [15]) and an OPM model with its explanation.

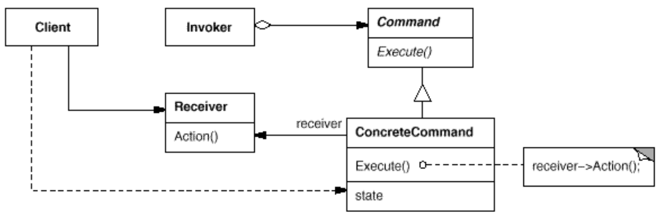
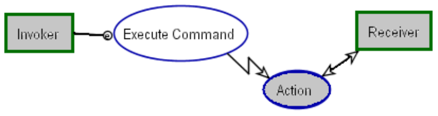
COMMAND	
Problem Definition: A request should be issued without knowing anything about the operation being requested (can be used as support for undoing operations).	**Suggested Solution:** Model the behavior as an independent entity.
UML Model:  Client Invoker Command Execute() Receiver Action() receiver ConcreteCommand Execute() state receiver->Action(); From [15], p. 236.	**OPM Model:**  "Invoker" is an object which raises an event that invokes the "Execute Command" process. "Execute Command" invokes "Action" (which is also a process) that affects the "Receiver" object.

Fig. 8. The Command design pattern

As the OPM models of these behavioral design patterns clearly show, the focus here is on behavior and way of invocation. In the OPM model of the Command design pattern, the trigger for executing the command is an event (denoted by the event link) from the "Invoker". When executed, the "Action" process is invoked, affecting the "Receiver". This way, the "Invoker" is not aware of the "Receiver" and even not of the specific "Action" that is performed. In the OPM model of the Chain of Responsibility design pattern, a "Handler" invokes (triggers) its successor. In the OPM model of the Observer design pattern, there are two different processes: "Notify", which is triggered by the "Subject" and affects the relevant "Observers" (defined by the structural link between "Subject" and "Observer"), and "Update", in which the "Observer" can change the state of the "Subject". Finally, in the OPM model of the Mediator design pattern, any "Colleague" may trigger the "Mediator", which in turn affects the "Colleagues".

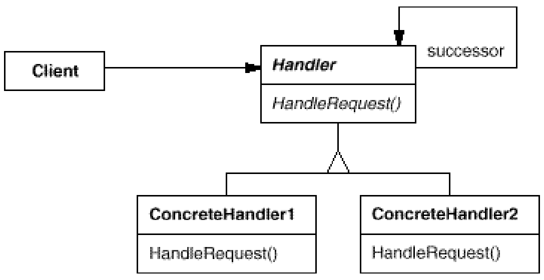
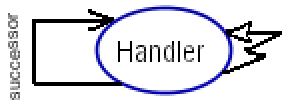
CHAIN OF RESPONSIBILITY	
Problem Definition: There is a need to initialize a behavior without specifying which implementation should be used.	**Suggested Solution:** Create a chain of behaviors and pass our request along this chain until it is handled
UML Model:  Client Handler HandleRequest() successor ConcreteHandler1 HandleRequest() ConcreteHandler2 HandleRequest() From [15], p. 225.	**OPM Model:**  The "Handler" process has a relation to another process of the same kind which is its successor. It also invokes this successor process.

Fig. 9. The Chain of Responsibility design pattern

OBSERVER	
Problem Definition: There is a one to many dependency relations between objects in the system.	**Suggested Solution:** Separate between the true object and its observers.

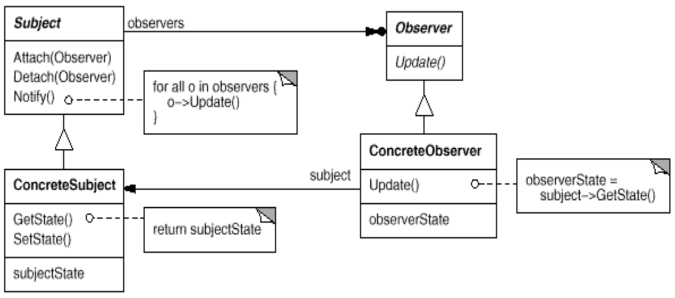
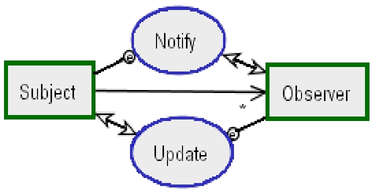
UML Model:
 Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState
From [15], p. 294.
	OPM Model:
	
	"Observer" and "Subject" are objects. A "Subject" may have many "Observers". The "Subject" invokes the "Notify" process that affects the "Observer". An "Observer" can invoke the "Update" process which affects the "Subject".

Fig. 10. The Observer design pattern

Since behavioral design patterns deal with algorithms and (mainly dynamic and functional) responsibilities, it should come as no surprise that most of their OPM models are dominated by processes. However, some of these design pattern models describe structural aspects and they "shift your focus away from flow of control to let you concentrate just on the way **objects** are interconnected" [15, p. 221]

The pattern that characterizes most of the behavioral design pattern OPM models is **object – event link – process – effect link – object**, implying that these design patterns have both triggering and affecting aspects. The two exceptions to this characterizing pattern are Chain of Responsibility and Template Method. When modeling the Chain of Responsibility design pattern in OPM, for example, we get a structure of processes that is quite similar to the Decorator structure.

TEMPLATE METHOD	
Problem Definition: A skeleton of an algorithm is common to several different implementations.	**Suggested Solution:** Encapsulate the skeleton in an abstract class and allow sub-classes to implement or redefine certain steps of the algorithm.

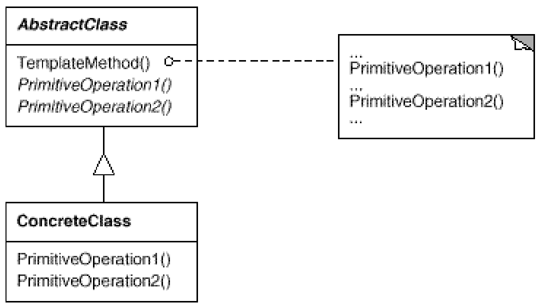
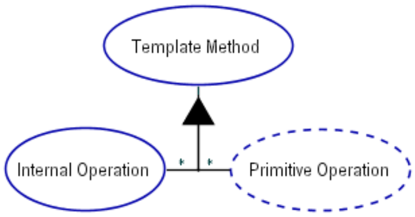
UML Model:
 AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()
From [15], p. 327.
	OPM Model:
	 Template Method
	Internal Operation
	Primitive Operation
	A "Template Method" process is an aggregate of "Internal Operations" and "Primitive Operations". The latter need more specification in a particular implementation of the design pattern.

Fig. 11. The Template Method design pattern

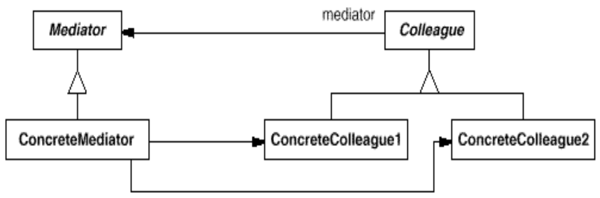
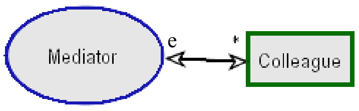
MEDIATOR	
Problem Definition: An object structure contains many connections between objects, making them interdependent. Behavior is distributed.	**Suggested Solution:** Encapsulate how a set of objects interact. This promotes loose coupling between the objects since they do not refer to each other explicitly.
UML Model:  Mediator mediator Colleague ConcreteMediator ConcreteColleague1 ConcreteColleague2 From [15], p. 273.	**OPM Model:**  The "Mediator" process is invoked by a "Colleague" object. The "Mediator" process changes the "Colleague" objects.

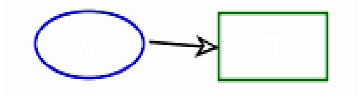
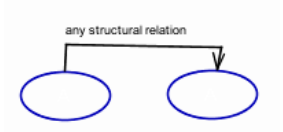
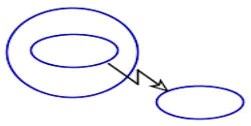
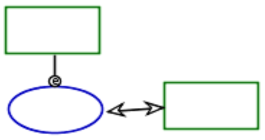
Fig. 12. The Mediator design pattern

The OPM model of the Template Method design pattern uses the notation of an environmental process (dashed border lines). An environmental thing (object or process) in OPM is either external to the system (or, in our case, the pattern) or is an abstract, under-specified thing in a generic model that needs further specification in the target application model. The OPM model of the Template Method design pattern specifies that the "Template Method" consists of "Internal Operations", as well as "Primitive Operations" that should be further specified in the context of an application. This model is quite similar to the OPM model of the Factory Method; they both define functionality which should be embedded in a behavior and can be preceded and/or followed by different operations. However, the focus of the Factory Method design pattern is behavioral—the embedded functionality creates products, while the emphasis of the Template Method design pattern is structural—the template consists of internal operations, as well as external (environmental) ones.

4.4 Classifying Design Patterns from an OPM Viewpoint

As we have seen, the OPM models induce a refined and precise way to classify design patterns. The design patterns we have reviewed in this paper are divided according to this classification into four groups, listed in Table 2. The first group of *creational design patterns* is identical to the creational group in the GoF classification and includes Abstract Factory, Factory Method, and Builder. Its characterizing pattern is process – result link – object, which purely conveys the idea of a process creating an object.

Table 2. Classification of Design Patterns according to OPM models

Design Pattern Category	**Design Pattern Examples**	**Typical OPM Model Core**
Creational	Abstract Factory Factory Method Builder	
Structural composition	Chain of Responsibility Composite Template Method	
Wrapper	Adapter Decorator Bridge	
Interaction	Command Observer Mediator	

The second group, *structural composition design patterns*, consists of Chain of Responsibility, Composite, and Template Method. This group has the most abstract characterizing structure: two processes are connected via a structural relation of any type. This is not surprising since there are several possible kinds of structural relationships. For the Composite design pattern, the in-zooming of "Composite" into "Component" reveals an aggregation relationship. Note that in the case of Chain of Responsibility, one process, "Handler", plays the role of both processes.

The Chain of Responsibility design pattern is often used together with the Composite design pattern. As their OPM models show, these design patterns are very similar, except that in the Composite design pattern there are two types of "Handlers", Composite and Component. Abstracting these two process types to one, we get the Chain of Responsibility design pattern. The containment relationships between "Composite" and "Component" define the hierarchy between the "Handlers".

The Template Method is a generalization of Factory Method. The OPM models of these design patterns justifies Zimmer's claim that the relationship between the Factory Method and Template Method is "X uses Y" [34], as the OPM model of the Template Method divides a process into "Internal Operations" and environmental "Primitive Operations" that should be further specified in the application context. In the Factory Method design pattern, the "Primitive Operations" are the operations that precede and follow the creation operation, while the "Internal Operation" is "Factory Method".

The third, *wrapper design patterns* group, includes the Adapter, Decorator, and Bridge design patterns. These patterns solve their stated problems by wrapping the original functionality. Decorator can be considered a special case of Adapter. While Decorator only adds functionality, Adapter also adapts the interface. Indeed, the Adapter and Decorator have been identified as wrappers in [14], [15], and [29]. We include the Bridge pattern in this category based on the structure of its OPM model, which is very similar to those of Adapter and Decorator. The wrapping essence of these design patterns is supported by their OPM models, in which one process actually wraps a call to another process.

Finally, the fourth group, which includes the Command, Observer, and Mediator design patterns, is what we call the *interaction design patterns* group. These patterns focus on the interaction between static and dynamic aspects of the solution. In OPM terms, these patterns emphasize procedural links, namely effect link and event link, which is responsible for updating and triggering, respectively. This common structure can be observed in different manifestations in the OPM models of the different design patterns that belong to this group. Note that for the Mediator design pattern, the "Colleague" object plays the role of both the affected object and the invoking object. In the OPM model of the Observer design pattern, the characteristic structure of the interaction design patterns appears twice. For the Command design pattern, the characterizing structure is extended by a second process.

5 Using Design Patterns in Applications: The OPM Technique

In the previous section, we explained and demonstrated why OPM is suitable for modeling and classifying design patterns. One of the main differences between the OPM and UML models of the same design patterns is their abstraction level. While the UML models include also the concrete elements, i.e., explanations how the abstract elements are used in the context of an application, the OPM models include only the structural and behavioral constraints of the design patterns. In this section, we show how the OPM models of the design patterns serve also as guidelines for applying the design patterns in particular applications.

As an example for using a design pattern in an application, consider a requirement to display the time of day according to the clock of the operating system. Different users will be comfortable with different time representations, such as digital or analog displays. Hence, the requirement is to present the same information in different manners. This information changes (e.g., every second) and all the displays must be notified of this change, such that they always present the correct time. This design problem can be solved by the Observer design pattern, where the "Operating System Clock" is the subject and the different displays ("Analog Display", "Digital Display", etc.) are the observers.

Figure 13 exemplifies the use of the Observer design pattern in the context of the requirement above. The application elements are mapped to the Observer elements using an OPM-based domain analysis technique [30]. According to this technique, the design patterns are modeled in the domain layer, which is different from the application layer. When modeling an application, the models in the domain layer can be used as validation templates for parts of the modeled application. In order to enable validating the system model against its domain model, the different system elements are classified according the domain elements. This classification, which bears some similarity to the UML stereotype mechanism, is depicted in the upper left corner of the things in the application model. The things in the application model should obey the constraints induced by the corresponding things in the domain model. These constraints are formally defined by an OPM model in the domain layer (also called the domain metamodel). In our case, the Observer model enforces constraints on the time displaying system: an "Observer" triggers an "Update" operation, which, in turn, affects the "Subject", a "Subject" triggers a "Notify" operation, which in turn updates the "Observers", and a "Subject" is connected (via a structural link) to the relevant "Observers". These constraints are indeed fulfilled in the clock system. When the time changes, the "Operating System Clock" triggers "Time Progressing", which notifies the "Digital Displayer" and "Analog Displayer" observers about the change. These observers are not aware of one another, but they are both connected to the same "Operating System Clock". The users may change the time from one of the displayers. In this case "Time Setting" is triggered, changing the time of the "Operating System Clock". This change of the clock triggers "Time Progressing", which notifies the other observers (displayers) about the change as well.

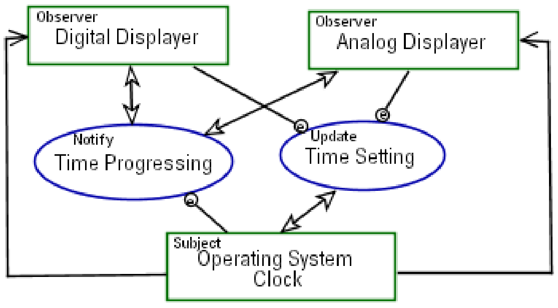


Fig. 13. The Observer design pattern applied in a Time Displaying System

The main benefit of this separation of OPM between design pattern models and their application in systems is achieving compact and formal design pattern models. The UML model of the same Observer design pattern includes 17 elements (classes, operations, attributes, and relations), compared with just 9 elements (objects, processes, and links) in the corresponding OPM model. This compactness does not reduce the model expressiveness. Conversely, the OPM model makes explicit the influence of the processes on the objects, which is only referred to by informal notes in the UML model. The OPM model also makes explicit the triggers of the processes, which are not included in the UML model at all. Furthermore, modeling the design pattern separately from its application in a particular system enables treating the design pattern model as a validation template for the application model. Using

OPCAT [10], the OPM-based CASE tool, this validation is done automatically, providing a powerful tool for conceptual model debugging.

6 Conclusions and Future Work

One way to increase people's trust in information systems is to use proven working solutions. Design patterns are a prime example of such reusable, proven solutions to frequently encountered design problems. To encourage software engineers to employ design patterns throughout the entire software development process, the design patterns should be classified logically and represented formally, so their retrieval be effective and their usage—correct. Since different stakeholders engaged in systems development are more likely to remember visual representations of ideas than textual ones, both UML and OPM suggest ways to model design patterns graphically. While UML offers an object-oriented approach, in which the design pattern and its application are modeled in tandem, OPM suggests a balanced structure-behavior approach, in which the design pattern is a generic metamodel (a model in the domain layer) that is applied (or instantiated) in the system-specific model (a model in the application layer).

In this paper, we have presented OPM models of twelve popular design patterns. We pointed out how the OPM models of the design patterns convey the essence of the solutions offered by the patterns and how these OPM models help designers integrate them into their application models. Comparing the design patterns OPM models to their UML counterparts, we have shown that the former are closer to the designer's "mental model" and are therefore, more comprehensible. Furthermore, we found out that the OPM models induce a logical classification of the twelve design patterns into four groups: creational, structural composition, wrapper, and interaction. This classification refines the GoF classification in [15]. We identified some problems in the GoF categorization and established, for example, that Chain of Responsibility and Template Method should be categorized as structural design patterns rather than behavioral ones.

We plan to apply our approach to additional design patterns and develop models of complete applications that use a host of design patterns. We also plan to develop ways to retrieve design patterns easily (using OPM models) from the problem domain as query inputs.

References

1. Alexander, C., Ishkawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language. New York: Oxford University Press (1977).
2. Buschmann, F., Meunier, R.: A System of Patterns. In: Coplien, J. O. and Schmidt, D. C. (eds.): Pattern Language for Program Design, Addison-Wesely (1995), pp. 325-343.
3. Buschmann, F., Meunier R., Rohnert, H., Sommerland, P., Stal, M.: Pattern-Oriented Software Architecture: a System of Patterns. Wiley (1996).
4. Brugali, D., Sycara, K.: Frameworks and Pattern Languages: an Intriguing Relationship. ACM computing surveys (march 2000), v.32, n.1es, article no. 2.
5. DeBruler, D. L.: A Generative Pattern Language for Distributed Processing. In: Coplien, J. O. and Schmidt, D. C. (eds.): Pattern Language for Program Design, Addison-Wesely (1995), pp. 69-89.
6. Dietrich, J., Elger, C.: A Formal Description of Design Patterns using OWL. Proceedings of the 2005 Australian software engineering conference (2005), pp. 243-250.
7. Dong, J., Yang, S.: Visualizing Design Patterns with a UML Profile. IEEE Symposium on Human Centric Computing Languages and Environments (2003), pp. 123- 125.
8. Dori, D.: Object-Process Methodology – A Holistic System Paradigm. Springer (2002).
9. Dori, D.: ViSWeb – The Visual Semantic Web: Unifying Human and Machine Knowledge Representations with Object-Process Methodology. The International Journal on Very Large Data Bases, 13, 2, pp. 120-147, 2004
10. Dori, D., Reinhartz-Berger, I., Sturm, A.: OPCAT – A Bimodal CASE Tool for Object-Process Based System Development. Proc. IEEE/ACM 5th International Conference on Enterprise Information Systems (2003), pp. 286-291.
11. Eden, A. H., Hirshfeld, Y., Yehudai, A.: LePUS – A declarative pattern specification language. Technical report 326/98, department of computer science, Tel Aviv University (1998).
12. Eden, A.H.: Precise Specification of Design Patterns and Tool Support in Their Application. PhD thesis, University of Tel Aviv (1999).

13.France, R. B., Kim, D. K., Ghosh, S., Song, E.: A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering (2004), 30 (3), pp. 193-206.

14.Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstraction and Reuse of Object Oriented Design. Proceedings of the 7th European Conference on Object Oriented Programming. Berlin: Springer-Verlag (2003), pp. 406-431.

15.Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994).

16.Gil, J., Lorenz, D.H.: Design patterns vs. Language Design. Proceedings of the Workshops on Object-Oriented Technology (1997). pp. 108-111.

17.Griffith, R., Borchers, J., Stork, A., Pemberton, L.: Pattern Languages for Interaction Design: Building Momentum. Workshop at the Conference on Human Factors in Computing Systems (2000), pp. 363.

18.Hannemann, J., Kicsales, G.: Design Pattern Implementation in Java and AspectJ. Proceedings of the 17th annual ACM conference on Object-Oriented programming, systems, languages, and applications (2002), pp. 161-173.

19.Lauder, A., Kent, S.: Precise Visual Specification of Design Patterns. Lecture Notes in Computer Science (1998), Vol. 1445, pp 114-136.

20.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 (2004), pp. 252-261.

21.Meszaros, G.: A Pattern Language for Improving the Capacity of Reactive Systems. In: Vlissides, J., Choplin, J., O., and Kerth, N., L. (eds.): Pattern Language for Program Design 2, Addison-Wesely (1996), pp. 575-591.

22.Noble, J.: Classifying relationships between Object-Oriented Design Patterns. Proceedings of the Australian Software Engineering Conference (1998), pp. 98-108.

23.Object Management Group. UML 2.0 Superstructure FTF convenience document. http://www.omg.org/docs/ptc/04-10-02.zip

24.Peleg, M., Dori, D.: The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods, IEEE Transaction on Software Engineering (2000), 26 (8), pp. 742-759.

25.Petri, D., Csertan, G.: Design Pattern Matching. Periodica Polytechnica Ser. El. Eng (2003). Vol. 46, no. 3-4, pp. 205-212.

26.Reinhartz-Berger, I.: Conceptual Modeling of Structure and Behavior with UML – The Top Level Object-Oriented Framework (TLOOF) Approach. Proceedings of the 24th International Conference on Conceptual Modeling (2005), Lecture Notes in Computer Science 3716, pp. 1-15.

27.Reinhartz-Berger, I., Dori, D. Katz, S.: Open Reuse of Components Design in OPM/Web. Proceedings of the 26th annual International computer software and application conference(2002), pp. 19-26.

28.Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenson, W.: Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, NJ (1991).

29.Shalloway, A., Trott, J.: Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley (2001).

30.Strum, A.: Developing and Evaluating an Object-Process Methodology-Based Multi-Agent System Framework. PhD thesis, Technion – Israel Institute of Technology (March 2004).

31.Sunye, G., Guennec, A.L., Jezequel, J-M.: Precise Modeling of Design Patterns. Proceedings of UML 2000 Springer Verlag (2000), pp. 482-496.

32.Vicente, K.J., Rasmussen, J.: Ecological Interface Design: Theoretical Foundations. Systems, Man and Cybernetics, IEEE Transactions on (1992). Issue 4, Volume 22, pp. 589-606.

33.Warmer, J. and Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley (1998).

34.Zimmer, W.: Relationships Between Design Patterns. In: Coplien, J. O. and Schmidt, D. C. (eds.): Pattern Language for Program Design. Addison-Wesely (1995), pp. 345-364.