Towards Easier Implementation of Design Patterns

Ruslan Batdalov and Oksana Nikiforova

Department of Applied Computer Science
Riga Technical University
Riga, Latvia
Email: Ruslan.Batdalov@edu.rtu.lv, Oksana.Nikiforova@rtu.lv

***Abstract*—Design patterns help in managing complexity of software systems, but in many cases their implementation may entail even greater complexity. We argue that this complexity is caused, at least partially, by the lack of expressiveness of the mainstream programming languages. In order to support this hypothesis, we propose a set of potential language-level features that might make implementation of design patterns easier, identified by dissecting some widely used design patterns.**

***Keywords*–design patterns; design patterns implementation.**

I. Introduction

At present, design patterns are widely used in development of software systems. Erich Gamma et al. defined the role of patterns as follows: 'A pattern gives a solution to a recurring problem and allows developers not to invent a design from scratch. [1]' As Frank Buschmann et al. pointed out, '[a]ny significant design will inevitably draw on many patterns, whether consciously or otherwise. [2]'

At the same time, introducing design patterns into a software system may increase the level of its complexity. This danger was anticipated yet by Erich Gamma et al.: 'Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection and that can complicate a design and/or cost you some performance. [1]' Frank Buschmann et al. in the first book of the series 'Pattern-oriented Software Architecture' saw one of the goals of the patterns in helping developers to manage software complexity [3], but in the fifth one admitted: 'There are also many tales of failure to relate: stories about systems in which patterns were used intentionally and explicitly in design, but whose architectures were penalized by unnecessary and accidental complexity. [2]' Peter Sommerlad, an author of many design patterns and a co-author of the same series, went even further and bluntly argued that design patterns are bad for software design because of unnecessary complexity [4].

We argue that the excessive complexity introduced by the design patterns stems, at least partially, not from the patterns, but from insufficient expressiveness of the existing programming languages. The languages have trouble expressing the ideas, on which the patterns are built, and that is a reason why the implementation of the patterns is too complex. To support this hypothesis, we propose an approach to coping with the mentioned complexity by means of including necessary constructs into the programming languages.

The goal of the study is to identify a set of language-level features that could facilitate implementation of the common design patterns. This result, as well as our approach in general, could be used by the designers of programming languages to identify new features that might be useful for reducing systems complexity.

The remainder of this paper is organised as follows: Section II briefly describes the background of the study related to the common design patterns and their implementation. Section III presents our approach and illustrates it with a detailed example. Section IV contains the proposed set of language-level features, their motivation and drawbacks. Related work is covered in Section V, and Section VI concludes the paper.

II. Background of the Study

There exist a lot of design patterns appearing again and again in different systems. They are described in a huge number of books and papers. The seminal work by Erich Gamma et al. [1] popularised the idea of a pattern in software design and gave an impetus to studying, discovering and applying new and new design patterns. Among other work, we should mention the series 'Pattern-oriented software architecture', which gives not only a catalog of patterns, but also many useful insights into purposes of design patterns and relationships between them.

These books are by no means the comprehensive description of the field. In 2007, Grady Booch mentioned that he had catalogued almost 2,000 design patterns found in the literature [5]. By now, this number is obviously much greater since the process of detection of new patterns has not been standing still. Nevertheless, the mentioned works describe perhaps the most commonly used and verified patterns, so it is logical to refer to them in the first place. In this paper, we work with the design patterns described by Erich Gamma et al. in 'Design patterns' [1] and by Frank Buschmann et al. in the first volume of 'Pattern-oriented software architecture' [3].

The books describing implementation of design patterns in the existing programming languages are numerous too. In general, they are less relevant to our study, but we can get some helpful insight from them. In particular, it is quite clear from such descriptions that the process of extending programming languages for the needs of the design patterns users is already going on. For example, enumerators and for-each loop in C# facilitate implementation of the Iterator pattern, and query expressions provide the interface of this pattern to the results of database queries [6]. These features were not supported in the first version of the language. Our study is to support this direction of programming languages development by providing some ideas on what features could be included.

III. Approach to Identification of Necessary Language-level Features

The simplest way to support design patterns that comes to mind is direct inclusion of the known design patterns into a programming language. In a sense, that is what happened with the Iterator pattern in the example given above. Joseph Gil and David H. Lorenz described the general characteristics and the typical steps of this process in 1998 [7]. In our opinion, this approach is fruitful, but not universal. In the late 1990s, just a few dozens of patterns were documented well enough, but there are thousands of them at present. The enormous number of the patterns and the pace of their emergence clearly do not allow to make them all language constructs. However, the fact that patterns are so numerous suggests that there should be recurring themes in them. So, we can try to find common elements of different design patterns and discuss whether they are able to become language-level features.

Another reason not to include a design pattern into a language directly is that many patterns are rather instructive than formal constructs and allow too high variation. Frank Buschmann et al. give the example of the Observer pattern, whose particular implementation may depend on the model of interaction between the subject and the observer (push or pull), on the presence of a middleware for the exchange of messages, on the chosen data structures, etc. They conclude that design patterns are not generic, but *generative* [2]. In our opinion, decomposition of patterns may allow to implement variation in patterns semantics by means of combining primitive elements having strictly defined semantics. These elements are better candidates for becoming language-level features than the patterns themselves.

According to the goal of our study, we are interested in the elements that are logically similar to the existing language-level constructs, but are not supported currently. The motivation here is that introduction of design patterns into a software system or getting rid of them may require substitution of such elements in place of standard language-level constructs or vice versa. If the required syntactical constructs are significantly different, these operations may require complex co-ordinated changes in different parts of the system, which is undesirable. Examples of such differences may be seen below.

Eliminating design patterns from a working system as a source of complexity was mentioned by Peter Sommerlad, who stated that it is (somewhat counter-intuitively) more difficult than their introduction [4]. On the basis of this observation, he argued against using design patterns too frequently, but, in our opinion, facilitating of such transitions is a more fruitful approach.

So, we can try to use the following approach to identification of potential language constructs:

- dissect a design pattern and decompose it into logical elements at level with language constructs (in the sense of the level of abstraction): elementary operations, simple relationships, etc.,
- find elements that are similar to the existing language constructs and/or recur in different patterns,
- analyse the extent to which the discovered elements are supported in the existing languages (they may be partially supported) and similarities between the elements and the existing language constructs,
- describe semantics of the discovered elements,
- analyse consequences of introduction of the potential language features.

As an example, we try to dissect the Abstract Factory pattern, described by Erich Gamma et al. [1], and identify the language-level features that would make its usage easier.

According to the original description, Abstract Factory is intended to provide an interface for creating families of related or dependent objects without specifying their concrete classes. Its application includes the following steps:

1) Declare an interface for creation of abstract objects (**AbstractFactory** creating **AbstractProduct**).
2) Implement the operations for creation of concrete objects (**ConcreteFactory** creating **ConcreteProduct**).
3) Choose a concrete implementation of the family of objects, depending on either compile-time or run-time conditions (i.e., choose a **ConcreteFactory** that will be used).
4) Instantiate required objects by means of methods declared in the **AbstractFactory** and implemented in the **ConcreteFactory** (usually implemented using the Factory Method or Prototype pattern) [1].

A point in this description that can attract our attention is that the last step is performed by auxiliary methods instead of the standard instantiation operator (`new` or whatever a particular programming language uses). That means that introduction of the pattern or getting rid of it requires changes in every client that instantiates the objects. It does not cause problems in the languages where factory methods are the primary means of objects instantiation (e.g., Perl), but in the languages with a separate instantiation operator it might be difficult. The problem is that such languages do not allow to use the instantiation operator with an abstract class since its concrete implementation that should be instantiated is not known in advance. The cause of this 'ignorance', in turn, is that the third step (choosing a concrete implementation) is not supported at the level of the programming language and needs to be implemented in the application itself. We may suppose that turning this step into a language-level operation can make implementation simpler.

We can turn to other patterns as well and see that the same idea of choosing an implementation is utilised, for example, in Builder, Bridge, Command and Strategy. So, this operation is a recurring theme in design patterns and probably deserves support at the language level.

The idea is to introduce an operator that would set a default implementation for an abstract class. It could be used either at the compile time or at the run time. If the choice has been made by the compile time, the compiler should substitute the constructor of the concrete class for the one of the abstract class. If the choice is delayed until the run time, the compiler should create an internal (hidden from the developer) Abstract Factory and redirect all requests for instantiation of an object of the abstract class to this factory.

We can foresee at least two problems related to this approach. First, the choice of an implementation is a form of binding, and its scope should be clearly defined. On the one hand, the scope should not be restricted to a single class (otherwise, we still need to change each and every client class to change the used implementation). On the other hand, a change of the binding from a class that would affect all other classes in the program may lead to unpredictable results. The second problem is that this binding is a shared state, so it should be carefully treated in a concurrent environment.

The mentioned problems can make one conclude that this feature should *not* be introduced, and such an argument would be reasoned. Nevertheless, we believe that what we see here is not a weakness, but a strength of the proposed approach. The problems of the scope and the global state belong to the pattern itself, not to its language-level support. An ad-hoc implementation of the pattern still needs to deal with the same issues. A native language-level support may provide a developer with a ready solution of the potential problems (inevitably limited, but suitable for most cases).

In a similar manner, we can see that responsibility delegation appears really often in various design patterns and clearly deserves its place in a programming language. The Builder, Command and Command Processor patterns generalise common ideas of object initialisation, function and executor respectively. A number of patterns (Blackboard, Pipes and Filters, etc.) provide alternatives to standard synchronous calls between objects. The State pattern abandons the assumption that a binding between an object and its class should remain the same throughout the whole life time of the object. The Template Method pattern involves switching between portions of code implemented in a class and its superclass (in both directions), which causes problems described below. We are not going to describe analysis of these patterns at the same level of detail as with the Abstract Factory pattern since the argument is very similar. Instead, the next section presents concrete language features that might be based on these observations.

IV. Proposed Features

This section contains the results of the analysis of the patterns described in [1][3]. For each feature, we give a context, in which it might be needed, and briefly describe the current situation, our proposal, and the negative consequences or issues that should be considered if the proposal is accepted.

The current state differs between programming languages, but we do not have a goal to observe all existing languages in this paper. Therefore, the description of the current situation is generally limited to a few mainstream object-oriented languages (C++, Java, C#). Occasionally, other languages are mentioned when it can give useful insight.

Limiting the scope of observed languages means that some proposed features may be already supported in other languages. Nevertheless, we believe that even in this case motivation from the design patterns angle might be useful. It may be illustrated by the example of for-each loop, iterating through collection-like data. It has been in use in Unix shell since the late 1970s, later it was added to some other languages (e.g., Pascal, Perl), but the mainstream languages ignored this feature for a long time, and incorporated it only after the rise of the patterns (the Iterator pattern in particular).

Table I lists all proposed features and the patterns, implementation of which they might facilitate. The table also includes a classification, inspired by the classification of design patterns according to their purpose by Erich Gamma et al. [1]. The features are classified according to whether they relate to the object lifecycle (creation, initialisation, association to the class), object behaviour (object's actions between its creation and destruction), or structural aspects (class hierarchy). It does not mean that if a particular feature is described as related to, for example, the lifecycle, then it is associated with creational patterns only. Behavioural patterns may also include elements related to the lifecycle or structure, etc.

A. Default Implementation

Context: An object may be declared to have an abstract class and instantiated with a concrete subclass.

Current situation: The concrete subclass is defined by the instantiation statement. If we need to change the used implementation, we have to change every instantiation statement. The concrete implementation is fixed at the compile time and cannot be changed at the run time.

Proposal: Introduce an operator that would choose the default implementation of an abstract class. If the default implementation has been defined (either at the compile time or at the run time), an instantiation statement may refer to the abstract class.

Known drawbacks: It is not clear what scope the binding of a class to its default implementation should have. If the scope is not local, the binding will be a shared state and require extra care in concurrent environments.

B. Extended Initialisation

Context: Some languages make difference between phases of the object lifecycle. For example, an object may be declared constant, so that it is initialised once only and cannot change its state later.

TABLE I. SUMMARY OF THE PROPOSED FEATURES

Feature	**Type**	**Related patterns**	
		GoF [1]	***POSA1 [3]***
Default implementation	Lifecycle	Abstract Factory, Builder, Bridge, Command, Strategy	
Extended initialisation	Lifecycle	Builder, Factory Method	
Chameleon objects	Lifecycle	State, Factory Method	
Generalised functions and executors	Behaviour	Command, Strategy	Command Processor
Object interaction styles	Behaviour	Façade, Proxy, Observer	Blackboard, Broker, Forwarder-Receiver, Master-Slave, Pipes and Filters, Proxy, Publisher-Subscriber
Responsibility delegation	Behaviour	Adapter, Bridge, Chain of Responsibility, Composite, Decorator, Façade, Flyweight, Mediator, Proxy	Broker, Layers, Master-Slave, Microkernel, Proxy, Whole-Part
Subclassing members in a subclass	Structure	Template Method, Visitor	

Current situation: An object declared as constant must be fully initialised with a single instantiation statement. At the same time, the Builder pattern constructs a complex object in a few operations. This prohibits using constant declaration for such objects even if they are never changed later.

Proposal: Treat single-line and multi-line initialisation uniformly. Allow the initialisation phase to consist of a few operations, requiring it to have a definite boundary nevertheless. After the boundary has passed, a constant object cannot be changed anymore.

Known drawbacks: Separation of initialisation phase will require careful definition for work in concurrent environments.

C. Chameleon Objects

Context: The State pattern is used when an object needs to change its behaviour at the run time (according to the original definition, the object *appears* to change its class [1]).

Current situation: Usually the class of an object is fixed at its creation. Some languages (e.g., Perl) allow to change objects' classes later, but it is uncommon.

Proposal: Allow to change an object's class at the run time.

This feature is also related to the Factory Method pattern. A factory method may decide on the concrete class of an object depending on its parameters. Being able to change class, we could implement the same logic in a class constructor.

Known drawbacks: It is not obvious how to ensure at least two natural requirements. First, the change should not involve data conversion (i.e., the classes should differ in behaviour only, not in their data structure). Second, in order to avoid unexpected behaviour, both the client and the object itself should 'know' the limits within which the class may be changed. For example, the client might expect that all possible classes of the object are descendants of one abstract class. If the state is changed by the state object itself (which is allowed by the original description [1]), it should declare that the class will never be changed to something else. Then the client may rely on the interface defined in the base class.

D. Generalised Functions and Executors

Context: The Command pattern describes a generalisation of conventional functions, which has the following features: is a first-class object, uses inheritance, may store internal state, supports undo, redo and logging. Some of this functionality may be implemented in a Command Processor instead.

Current situation: The existing languages have different forms of generalised functions (lambda-expression, functors, generators, etc.) that have some of these features, but not all. These forms are summarised in Table II.

Proposal: Replace conventional functions with a generalisation having the power of the Command pattern. Implement classes responsible for functions execution (threads, executors, debugging environments) according to the Command Processor pattern.

Known drawbacks: Such generalisation may be too complicated for simple functions. On the other hand, the proposal does not require implementing all mentioned operations for each and every function. The essence of the proposal is the opportunity to add them as smoothly as possible when they *are* needed. Moreover, these operations may be needed even for simple functions. For example, Online Python Tutor supports undoing and redoing of every executed statement for the purposes of studying and debugging [8] (in terms of the design patterns, it means implementing undo and redo in the Command Processor).

Another problem is that it may be hard to find a design that is simple enough and suitable for the general case. The proper distribution of responsibilities between functions and executors is not obvious as well.

TABLE II. GENERALISED FUNCTIONS

	First-class objects	**May inherit**	**May store state**	**Undo, redo, logging**
Conventional functions	Yes/No[a]	No	No	No
Lambda-expressions (Java, Python, etc.) Delegates (C#)	Yes	No	No	No
Functors (C++)	Yes	Yes	Yes	Yes/No[b]
Generators (Python) Enumerators (C#)	Yes	No	Yes	No

[a] Depending on the language.
[b] May be implemented, but syntactically differ from conventional function calls.

E. Object Interaction Styles

Context: Using the Façade or Proxy patterns, a single class may represent a whole component, a subsystem or an external system. With the Broker pattern, such a system may be even distributed.

Current situation: There are different ways how components may interact: synchronous request-response, asynchronous request-response, pipe&filter, broadcast, blackboard, publish-subscribe [9]. Nevertheless, synchronous call is the predominant interaction style between objects in the existing languages. Other styles are usually implemented with a complex sequence of synchronous calls.

The need for a simpler approach may be illustrated by the existence of interface definition languages for concurrent and distributed systems. These languages describe interaction between systems using richer sets of interaction mechanisms (at least, including asynchronous requests and responses).

Proposal: Introduce constructs that would represent different interaction styles at the language level. They may better represent the way how developers think about components. Interface definition languages would not be needed any more since we would be able to define the required interactions directly.

Known drawbacks: The resulting syntax may be difficult to learn.

F. Responsibility Delegation

Context: A class may delegate its responsibility to another class. Usually, this delegation involves some changes in the call, but sometimes a request is simply forwarded.

Current situation: Delegation adds an extra level of indirection, even if it is unnecessary. Long chains of delegation may be resource-consuming. Peter Sommerland mentioned delegation as one of the main sources of excessive complexity introduced if we use design patterns carelessly [4].

Another problem is that once the request is forwarded, the reference to the object originally receiving the request is no longer available (*self* problem) [10].

Proposal: Allow a method to be explicitly marked as delegated. There may be different types of delegation, for example, keeping the original reference or not. A starting point in identifying which types of delegation should be supported may be the types identified by Jan Bosch in [10].

Provided that the delegate is known in advance and the method signatures are the same, the compiler or the run-time environment may get rid of the extra level of indirection. Even if this optimisation is impossible, delegation may be supported in integrated development environments. It would facilitate grasping code that uses delegation extensively.

Known drawbacks: Conditions of when the optimisation is possible may happen to be very restrictive. Probably, most delegations will not allow it. Furthermore, in our opinion, the self problem often should *not* be solved, since keeping the original reference may easily lead to violation of Demetra's law. Nevertheless, sometimes the original receiver of the call is really needed.

G. Subclassing Members in a Subclass

Context: A class may use a member declared in its superclass, but restrict the set of possible values (for example, choose a concrete implementation of an abstract class). The behaviour of the subclass may rely on this restriction since the subclass never assigns an illegal value to this member.

Current situation: A member defined in the superclass keeps its declaration in all subclasses. A subclass that have decided to use a particular subclass for this member must perform run-time casts, which is error-prone. It is especially important when the behaviour is distributed between the class and its superclass (e.g., using the Template Method or Visitor patterns).

For example, programming a user interface for Android requires to subclass standard classes, such as `Activity`, `Fragment`, etc. The application may know that a fragment of class `MyFragment` can be attached only to an activity of class `MyActivity`. Methods of class `MyFragment` may need to refer to methods and data defined in class `MyActivity`. Nevertheless, method `MyFragment.getActivity()` returns an object of class `Activity`, since it is defined in the superclass `Fragment`. The result of every such call should be dynamically casted to class `MyActivity`.

Proposal: Allow to redeclare a member in a subclass, restricting the member's class. It is known as depth subtyping in the type theory [11] and may be generalised to predicate subclassing, proposed earlier by Ruslan Batdalov [12].

Known drawbacks: To be useful, the change of class should be propagated to getters and setters, which is difficult in languages without a language-level association between fields and getters/setters (C++, Java).

Another problem to solve is the issue of variance. Although the proposal itself does not violate variance rules, its propagation to a setter would mean covariance of a method argument, which is considered unsafe [13].

V. Related Work

A number of approaches to incorporation of design patterns into programming languages were proposed. The number of attempts itself shows the desire to have a better support of design patterns at the language level (even though the work in this area is mostly rather old).

Joseph Gil and David H. Lorenz described gradual percolation of design patterns into languages [7]. Unfortunately, they did not provide a systematic approach to how to perform this process and make related decisions.

Jan Bosch described design patterns in terms of layers and delegations and proposed an implementation using so called Layered Object Model [10]. Effectively, the approach involves using a completely new language to solve their tasks. Some ideas of this work are used in ours, but in general, the approach does not solve the problem of identifying separate language-level features.

Perhaps, the most notable are implementations of the design patterns in aspect-oriented languages. Jan Hannemann and Gregor Kiczales described an implementation of the design patterns described by Erich Gamma et al. [1] in Java and AspectJ [14]. Miguel P. Monteiro and João Gomes did the same in Object Teams [15]. Pavol Bača and Valentino Vranić developed the idea further and proposed to replace the commonly known object-oriented design patterns with aspect-oriented ones [16].

Frank Buschmann et al. proposed the hypothesis that it is possible to provide configurable generic implementations for patterns that cover their whole design space, but refuted it in just ten pages [2].

Our approach differs from the mentioned ones in that it provides an insight on *how* we could find new language-level features for easier implementation of design patterns. The proposed features are also supposed to be suitable for object-oriented languages and not require a less common programming paradigm. At the same time, they may in principle be used in aspect-oriented or other extensions as well. So, these approaches can be considered rather complementary than competing.

Another field of study related to our approach is decomposition of design patterns. Uwe Zdun and Paris Avgeriou tried to identify primitives of which architectural patterns consist [17]. These primitives are of the architectural nature and not directly related to programming languages features. Francesca Arcelli Fontana et al. performed a detailed analysis of micro-structures comprising common design patterns [18][19]. Their primary goal was to facilitate design patterns detection, so the found micro-structures are not necessarily at level with language features either. In our opinion, the observation by Frank Buschmann et al. that design patterns are not generic, but generative [2] refers to many of these micro-structures as well.

VI. Conclusion and Future Work

Our study addressed the problem of excessive complexity, often introduced by application of design patterns. The developed approach allowed to identify a set of language-level features supporting the patterns described in [1][3]. We can conclude that the problem of implementation complexity may, at least in principle, be tackled by extending programming languages, so that the design patterns would be easier to apply. A sample set of such features is the main contribution of this study. At the same time, these features have their drawbacks, and their introduction requires to solve a number of problems.

The work may be continued in order to cover more patterns from different sources. At the same time, we do not expect finding a big number of new features in the course of this work. Although some new discoverings are inevitable, the viability of the system will be best justified if Table I grows more rightwards than downwards. A language should not grow infinitely with the number of expressions in this language.

Another direction of the future research is looking for the ways how the proposed features can be implemented in real programming languages and whether they can be implemented at all. If they can, it will raise, perhaps, the most important question of further study – the practical testing of our hypothesis that introduction of the proposed features allows to reduce complexity of the patterns usage in real software systems.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995.

[2] F. Buschmann, K. Henney, and D. C. Schmidt, Pattern-Oriented Software Architecture, On Patterns and Pattern Languages, ser. Pattern-Oriented Software Architecture. Wiley, 2007.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture, A System of Patterns, ser. Pattern-Oriented Software Architecture. Wiley, 2013.

[4] P. Sommerlad, "Design patterns are bad for software design," IEEE Software, vol. 24, no. 4, pp. 68–71, 2007.

[5] G. Booch, "The well-tempered architecture," IEEE Software, vol. 24, no. 4, pp. 24–25, 2007.

[6] J. Bishop, C# 3.0 Design Patterns. O'Reilly Media, 2008.

[7] J. Gil and D. H. Lorenz, "Design patterns and language design," Computer, vol. 31, no. 3, pp. 118–120, 1998.

[8] P. J. Guo, "Online Python Tutor: Embeddable web-based program visualization for CS education," in Proceedings of the 44th ACM Technical Symposium on Computer Science Education, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 579–584.

[9] I. Crnković, S. Séntilles, A. Vulgarakis, and M. R. V. Chaudron, "A classification framework for software component models," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 593–615, 2011.

[10] J. Bosch, "Design patterns as language constructs," Journal of Object-Oriented Programming, vol. 11, no. 2, pp. 18–32, 1998.

[11] B. C. Pierce, Types and programming languages. MIT press, 2002.

[12] R. Batdalov, "Inheritance and class structure," in Proceedings of the First International Scientific-Practical Conference Object Systems — 2010, P. P. Oleynik, Ed., 2010, pp. 92–95. [Online]. Available: http://cyberleninka.ru/article/n/inheritance-and-class-structure.pdf 2016.07.11

[13] F. S. Løkke, "Scala & design patterns," Master's thesis, University of Aarhus, March 2009.

[14] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," ACM Sigplan Notices, vol. 37, no. 11, pp. 161–173, 2002.

[15] M. P. Monteiro and J. Gomes, "Implementing design patterns in Object Teams," Software: Practice and Experience, vol. 43, no. 12, pp. 1519–1551, 2013.

[16] P. Bača and V. Vranić, "Replacing object-oriented design patterns with intrinsic aspect-oriented design patterns," in Proceedings of the 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC). IEEE, 2011, pp. 19–26.

[17] U. Zdun and P. Avgeriou, "A catalog of architectural primitives for modeling architectural patterns," Information and Software Technology, vol. 50, no. 9-10, pp. 1003–1034, 2008.

[18] F. A. Fontana, S. Maggioni, and C. Raibulet, "Understanding the relevance of micro-structures for design patterns detection," Journal of Systems and Software, vol. 84, no. 12, pp. 2334–2347, 2011.

[19] ——, "Design patterns: a survey on their micro-structures," Journal of Software-Evolution and Process, vol. 25, no. 1, pp. 27–52, 2013.