A Mapping Study of Language Features Improving Object-Oriented Design Patterns

William Flageol[a], Eloi Menaud[b], Yann-Gaël Guéhéneuc[a], Mourad Badri[c] and Stefan Monnier[d]

[a] *Concordia University, 1455 boulevard de Maisonneuve Ouest, Montreal, Québec, Canada, H3G 1M8*
[b] *IMT Atlantique, 4 Rue Alfred Kastler, Nantes, France, 44300*
[c] *Université du Québec à Trois-Rivières, 3351 boulevard des Forges, Trois-Rivières, Québec, Canada, G9A 5H7*
[d] *Université de Montréal, 2900 boulevard Édouard-Montpetit, Montréal, Québec, Canada, H3C 3J7.*

ARTICLE INFO

Keywords:
systematic mapping study
design patterns
object-oriented programming
language features
software measures
programming paradigms

ABSTRACT

Context: Object-Oriented Programming design patterns are well-known in the industry and taught in universities as part of software engineering curricula. Many primary studies exist on the impact of design patterns on software, in addition to secondary studies summarizing these publications. Some primary studies have proposed new language features and used them to re-implement design patterns as a way to show improvements. While secondary studies exist, they mainly focus on measuring the impact of design patterns on software.

Objectives: We performed a systematic mapping study to catalogue language features in the literature claiming to improve object-oriented design patterns implementations, as well as how primary studies measure these improvements.

Methods: We performed a search in three databases, yielding a total of 874 papers, from which we obtained 34 relevant papers. We extracted and studied data about the language features claiming to improve design patterns implementations, the most often cited design patterns, the measures used to assess the improvements, and the case studies and experiments with which these improvements were studied.

Results: Using the results, we catalogue 18 language features claimed in the literature to improve design patterns and categorize them into paradigms. We find that some design patterns are more prevalent than others, such as Observer and Visitor. Measures related to code size, code scattering and understandability are preferred. Case studies are done in-vitro, and experiments are rare.

Conclusion: This catalogue is useful to identify trends and create a road map for research on language features to improve object-oriented design patterns. Considering the prevalence of design patterns, improving their implementation and adding language features to better solve their underlying concerns is an efficient way to improve object-oriented programming. We intend in the future to use this as a basis to research specific language features that may help in improving object-oriented programming.

1. Introduction

Object-Oriented Programming (OOP) is the de-facto general programming paradigm. In many universities, software engineering is taught almost exclusively using OOP concepts.

Much of the concepts of OOP are inherited from languages from the 60s and 70s, such as Simula and Smalltalk. There have been many popular object-oriented languages, such as Java, C# and Python, and languages that adopted object-oriented concepts, such as C++, Common Lisp, and JavaScript. Some of the concepts are present in some form or another in most modern languages.

Object-Oriented Programming matured over the last few decades and is at a stage where it has well-defined practices. Some of these practices are catalogued as *design patterns*. A design pattern is a generic way to solve a recurring problem in software programming. Design patterns are distilled from coding experience from professionals and recurring structures in real-world software [1].

Arguably the most popular design patterns relate to OOP and come from the 1994 book by Gamma et al. [1] and describe 23 design patterns implemented in real-world software in C++ (with some other examples in Smalltalk). We refer to these 23 design patterns as the Gang of Four (GoF) patterns and the implementation examples in the book as the classical implementations of design patterns.

The GoF patterns and other such OOP design patterns help solving problems that would otherwise prove difficult in OOP programming languages. They are also solutions to problems within the paradigm itself: problems that cannot be solved simply using OOP concepts, such as creating callbacks on particular events (Observer pattern) or encapsulating code inside data (Command pattern).

Design patterns are presented as "templates" that can be adapted to specific problems for which using "naive" OOP solutions would not be optimal. However, they can introduce new problems, such as raising complexity, requiring that programmers be familiar with these design patterns to get the most out of their advantages. Overuse of design patterns, as well as "over-engineering" [2], should be avoided.

One secondary study by Zhang and Budgen [3] considered empirical studies on the effects of applying design patterns in software design. They showed that there was a lack of empirical studies on most design patterns and that empirically founded advantages and disadvantages are difficult to find in the literature. It also discussed design patterns implementations sometimes being harmful to program understanding,

ORCID(s):

while they do help with maintenance and evolution.

Primary studies tried to find other object-oriented patterns for situations not covered by the initial 23 design patterns. Others tried to use object-oriented approaches to improve the implementations of existing patterns [4]. Yet others tried to leverage language features from other programming paradigms to improve specific measures of design patterns [5, 6, 7, 8, 9]. Yet, there has not been much focus on the relationship between programming language features and design patterns implementations.

To gain a better understanding of this relationship between design patterns implementations and language features, we propose to study the literature on design pattern improvements. We specifically look for language features which helps the implementation of design patterns by simplifying them (we give an example of this in Section 2) or by improving specific measures (e.g., coupling between objects, cohesion, etc.).

We perform a systematic mapping study to identify language features claimed to improve design patterns implementations. The *goal* is to identify which language features have been suggested to improve OOP design patterns implementations, which design patterns implementations are being improved upon, which measures are used to evaluate these improvements, and what empirical data was collected to assess these improvements.

We perform a search query in three databases (Ei Compendex, Inspec, GEOBASE) using the Elsevier search engine Engineering Village. Our initial query yields 874 papers, which we assess for quality and use for snowballing, resulting in a total of 34 primary studies. We then extract relevant data from these 34 studies and discuss and catalogue our observations and their meaning.

We catalogue 18 language features claimed in the primary studies to improve design patterns implementations and categorize them into paradigms. This catalogue is useful to identify trends and create a road map for research on language features to improve object-oriented design patterns. Considering the popularity of design patterns, improving their implementation and adding language features to better solve their underlying concerns is an efficient way to improve Object-Oriented Programming. We intend to use this in the future as a basis to research specific language features that may help in improving Object-Oriented Programming.

The rest of this paper is organized as follows: in Section 2, we further explain how design patterns can be improved by language features and give a running example. In Section 3, we present an overview of other related secondary studies on the topic. In Section 4, we shortly present the concept of the systematic mapping study, we pose our research questions and present the methodology to perform our mapping study. In Section 5, we lay out the results related to our different research questions. In Section 6, we discuss these results and their implications. In Section 7, we list notable or interesting findings and our recommendations for future work. Section 8 lists threats to the validity of this study. In Section 9, we conclude with the main findings of this paper and further possible research.

2. Background

We illustrate how a design pattern can be improved by specific language features using the *Factory Method* design pattern, as discussed in the GoF book [1]. Figure 1 shows the first example of an implementation of *Factory Method* in the book. We present this example because of how directly the *Factory Method* design pattern maps to object instantiation language features. The example was designed with the C++ language in mind, but would be applicable to most other modern OOP languages, such as C# and Java.

The goal of this pattern is to decouple an application from concrete implementations of abstract classes or interfaces. To achieve this goal, we must encapsulate object instantiation. *Factory Method* suggests creating an abstraction of this creation process and using this abstraction throughout the software instead of the normal object creation feature of the language (e.g., the `new` keyword). In Figure 1, the class `Application` represents the abstraction of the creation process of documents. If we wanted the application to instantiate documents of type `MyDocument`, we would use an application of type `MyApplication`.

This pattern encapsulates the normal object instantiation mechanism provided by the language. The solution of the design pattern is to encapsulate this mechanism because the instantiation mechanism in C++ (and similar languages) does not allow for decoupling an abstraction from its concrete implementations (i.e., the `new` keyword only allows for the instantiation of concrete objects).

A variant to this solution is given a few pages later [1] with Smalltalk. In Smalltalk, there is no keyword for object creation. Object creation is accomplished using a method declared in the meta-class of the class (also called `new`). Because classes are also objects in Smalltalk, it is possible to call this method on a variable, effectively the *Factory Method* pattern. Thus, the document example could be implemented in Smalltalk with a method in `Application` that returns the class to use to create document objects. The `MyApplication` class could implement this method as returning `MyDocument`, as such:

```
"In class MyApplication"
documentClass
    ^ MyDocument
createDocument
    ^ documentClass new
```

In this implementation, there is no need to supersede the normal object creation mechanism of the language. We instead use it normally on the class returned by `documentClass`[1]. By using Smalltalk's meta-class and object instantiation language features, the pattern becomes *part of the language* and is more easily combined with other language features.

[1]Using this, we could restructure the model in different ways. For example, we could remove the `MyApplication` class and replace it with a collection holding the different concrete classes used to create objects for the application.

There are other examples in the GoF book [1] where design patterns have implementations that are simpler in Smalltalk than C++. This discrepancy between implementations suggests that design patterns implementations depend on what features are supported by the programming language.

In some languages, design patterns can even become obsolete and disappear entirely when the concept is fully supported by the language [10].

In this work, we are interested in cataloguing language features that impact the 23 GoF design patterns, and see what this impact is and how it was measured.

3. Related Studies

Multiple secondary studies exist about object-oriented design patterns.

In 2012, Zhang et al. [3] published a systematic literature review about the effectiveness of software design patterns. They targeted empirical studies made on the GoF design patterns. They concluded that there is a lack of empirical studies about design patterns and concrete advantages and disadvantages are difficult to find in the literature. They also concluded that design patterns may not help with understandability, sometimes decreasing it dramatically.

In 2013, Ampatzoglou et al. [11] performed a mapping study on the impact of design patterns on software quality. They focused on the GoF design patterns and categorized research into different categories (formalization, detection, and application). They found that research on the detection of design patterns and their impact was the most active. In another paper, the same authors proposed a catalogue [12] of alternative designs for the GoF patterns. These alternative designs usually add new functionality to existing patterns, either to make them more flexible [13] or to adapt them to specific situations [14].

In 2016, Mayvan et al. [15] published a state of the art on research on design patterns with the goal of presenting an entry-level summary to those who would seek to enter the research field. They perform a systematic mapping study to identify popular research trends, such as commonly used keywords, most active researchers and venues, and the distribution of publications by topics. They identify Pattern Development as the most popular topic, which groups the introduction of new patterns or pattern variants and categorization of design patterns by field (e.g., mobile applications, security, etc.).

These studies, and the studies they review, focus mainly on evaluating the impact of design patterns on object-oriented software. Many of them compare code that was developed without the usage of design patterns against code that was developed with. Others explore alternative design patterns to respond to specific problems. Our study catalogues language features used in the literature claiming to improve object-oriented design patterns implementations, without modifying their functionality, as well as how these improvements are measured and the empirical data available.

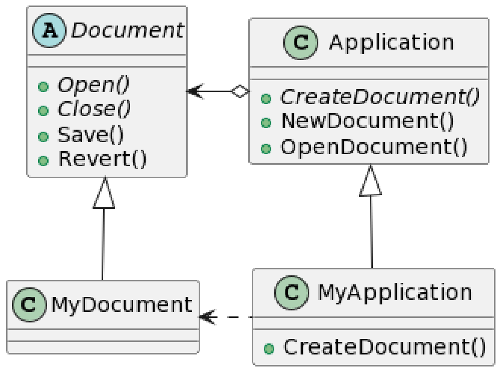

Figure 1: Implementation example of *Factory Method* [1]

4. Methodology

To reach our goal and answer our research questions, we perform a systematic mapping study following similar practices as introduced by Peterson et al. [16] in their systematic mapping studies guidelines for software engineering and executed by Ampatzoglou et al. [11]. Our methodology also draws from general guidelines on systematic literature reviews [17, 18].

Our objective is to find language features in research done to improve OOP design patterns. In particular, we want to survey the techniques and tools developed to make these improvements, as well as the measures used to assess them. We are also interested in understanding the kind of experiments done to evaluate these tools and techniques. We want to answer the following research questions:

1. RQ1: What language features have been suggested to improve design patterns implementations?
2. RQ2: Which design patterns have the most associated language features suggestions?
3. RQ3: What measures have been used to evaluate the impact of these language features on design patterns implementations?
4. RQ4: What experiments have been done on these language features?

To answer these questions, we perform a search to find and select the papers that will be studied in this study. Then, we ensure that the identified papers are related and offer answers to the research questions by performing a quality assessment. Finally, we extract data from the papers and synthesize it to reach our goal.

To reduce bias as much as possible, we perform the literature search in multiple databases and gather a large number of papers, which we then systematically study. We also perform a snowballing step to include all papers citing or cited by any paper which is not rejected in the manual sorting, followed by other snowballing iterations on the resulting set until reaching a fixed point.

Consequently, we follow these nine steps (as shown in Figure 2):

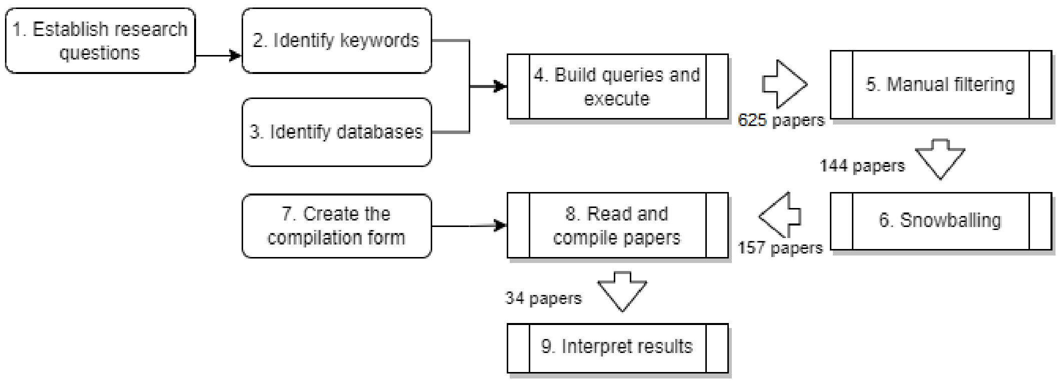

Figure 2: Flowchart of the steps in our approach

1. Identify keywords for searching for papers.
2. Identify databases for searching for papers.
3. Build a search query and execute it based on the keywords on each database.
4. Filter the results manually based on the abstract and keywords.
5. Perform a forward and backward snowballing step where each paper's references and papers that reference it are also considered (manually).
6. Create a form for compiling the data provided by the papers.
7. Read and compile each paper, removing any paper that does not provide data for at least one of our research questions.
8. Interpret the compiled data and answer the research questions.

4.1. Keywords for the Search Query

Based on our goal, we identify keywords to find papers potentially providing data to answer our research questions. These keywords do not encompass all possible synonyms and are general. For example, we include "object-oriented programming" as a keyword, but do not add "OOP", "object-oriented", or other synonyms. We add synonyms in Step 4 when we build the database query. We are investigating improvements in design patterns, thus we added some keywords related to weaknesses and anti-patterns. While these could be seen as unrelated, there is no disadvantage in having a broader query and unrelated papers will be removed in later steps of the study.

We identify the following keywords to use for building the search query: *object-oriented programming, design patterns, anti-patterns, weakness, disadvantage, improvement, enhancement, refinement, better, expand.*

4.2. Databases for the Search Query

To perform as large a search as possible, we use multiple online databases for scientific papers. These databases are part of an online tool called Engineering Village, hosted by Elsevier. They contain data from a large number of scientific journal databases, such as ACM and IEEE. Given the focus of this study on the GoF design patterns, we consider papers published between 1995 and 2022.

4.3. Query Building and Execution

We are using only one search engine, thus only one query is needed. All three of the databases are accessed from Elsevier Engineering Village Web site [19]. We restrict the scope of the query to the title, abstract, and keywords of the papers. We opt to use the more general term "object" (and then filter manually) instead of the various ways of writing "object-oriented programming" to avoid accidentally excluding relevant studies that used a slightly different expression (e.g., object oriented software).

Through several iterations, we add a large number of keyword exclusions ("NOT" keywords) to increase the relevance of the studies and reduce the number of unrelated studies. We exclude these keywords specifically from the "keywords" field in the database (meaning the paper should be about these topics), and not from the abstract (e.g., we are not excluding a paper because "test" is found in its abstract). The final query is shown in Table 1. This query yields **874 studies**.

The final query results are shown in Table 2. Removing duplicates is done via a script, as we found the Engineering Village search engine functionality to remove duplicates was not accurate. After removing duplicates, we retain 625 studies for manual filtering.

4.4. Filtering

We manually read the title and abstract of each of these studies and determine whether the study has *potential* to provide data to answer one of our four research questions. We choose to be conservative in this step and include any study that could provide data for a research question, even if remotely. Following quality assessment guidelines [16], inclusion criteria were applied to the titles and abstracts:

1. Study is in the field of software engineering.
2. Study is related to the improvement of design patterns and must propose a language feature to do so. (Studies relating exclusively to measuring the impact of existing design patterns against non-pattern code are not kept.)
3. Study was published between 1995 and 2022.

Query	Comments
((oop OR object) AND (design pattern OR design patterns OR anti-pattern OR anti-patterns OR paradigm) AND (weakness* OR disadvantage* OR improve* OR enhance* OR refine* OR help OR better OR expand*) WN AB)	Must be linked to design patterns Initially, we intended the scope to be broader, but this shouldn't impact the final results Must be about improving or finding weaknesses These keywords are located in abstract text
NOT ((teach* OR learn* OR test* OR modeling OR automated OR automation OR tool* OR mobile OR optimiz* OR simulation OR mining OR medical OR bio* OR hardware OR hdl OR parallel* OR api OR find* OR sql) WN KY)	These keywords are excluded to avoid papers about learning, automating, optimizing tools, or other non-related subjects encountered during the first passes of the query. Excluded keywords must not appear in the keyword field.
NOT ((pattern recognition OR pattern identification OR pattern detection OR object recognition OR object detection OR feature extraction OR computer vision OR pattern clustering OR learning (artificial intelligence) OR image segmentation OR pattern classification OR data mining OR computer aided design OR formal specification OR image classification OR user interfaces OR computer simulation OR internet OR image processing OR codes (symbols) OR image enhancement OR embedded systems OR image reconstruction OR cameras OR distributed computer systems OR product design OR artificial intelligence OR iterative methods OR neural nets OR neural networks OR object tracking OR genetic algorithms OR classification (of information) OR learning systems OR mathematical models OR middleware OR internet of things OR cloud computing OR decision making OR electroencephalography OR virtual reality OR risk management OR health care OR distributed object management OR query processing OR knowledge based systems) WN KY)	Finally, a large list of expressions was excluded from the keywords because they tended to be attached to papers about concepts unrelated to this research.

Table 1
Database Query

Database	Results
Compendex	316
Inspec	371
Geobase	13
All databases	700
After de-duplication	625

Table 2
Query Results by Database

Similarly, criteria were used to exclude papers:

1. Study is not related to one or more specific design patterns.
2. Study is not peer-reviewed.
3. Study is not written in English.
4. Study has no accessible full-text online.
5. Documents that are books or gray literature.
6. Study is a duplicate. (We kept the longer version.)

After this filtering step, **144 papers** remain.

4.5. Snowballing

We consider both the references of each study and any study referencing the said study for each of the 144 studies yielded by the previous step.

To search for forward references, we use the Scopus search engine. We apply the same year restriction (1995–2022) and the same criteria for filtering. After the third iteration of snowballing, we notice that all of the results are duplicates of already included studies, so we stop the snowballing process. Both the first and second authors of this study independently perform this step. The second author was not privy to the research questions being assessed until the process was done. We consolidate the results afterwards and address any discrepancies. After eliminating all duplicates, we obtain a total of **157 studies** that could provide data to answer our research questions.

4.6. Compilation Form

To compile the information from the obtained studies systematically and consistently, we build a form that we fill out for each study. Our form has the following information:

1. The title of the study.
2. The type of the study (experiment, case study, conceptual analysis, literature review, or survey).
3. The study publication venue.
4. The study publisher (or digital source).
5. A list of all design patterns mentioned by the study.
6. A list of all language features proposed by the study to improve the specified design patterns.

7. A list of all measures used by the study to evaluate the suggested improvements.
8. In the case of experiments, information about the participants (number and expertise).
9. In the case of case studies, information about the studied cases (number and whether or not they are in-vivo or in-vitro).

After using the form to extract data from the 157 compiled studies, we remove 123 as they did not provide an answer to any of our research questions. Many of the studies were evaluating the impact of existing design patterns, rather than any improvements, and this does not fit the goal of this study. (Thus, we remove 123 of the obtained studies, confirming that we were conservative in our process and likely did not miss any relevant paper.) We finally obtain **34 studies**.

5. Results

We classify the 34 remaining studies in various ways to answer our research questions. We first perform a general analysis of the sources (publication venues and publishers) the papers came from. Finally, we categorize each study into a type and take a more detailed look at the experiments and case studies.

Table 3 shows the number of papers for each publication venue. As we can see, there are a large variety of venues in our dataset. Some venues show up multiple times, such as PLoP[2], ECOOP, OOPSLA and SAC, but the rest of the papers all come from different venues. The large variety of publication venues could indicate that the subject of using language features to improve object-oriented design patterns is rather niche and that there is no specific venue for this.

Figure 3 shows the distribution of publication years of the studies. Interest in the subject seems to have peaked between 2010 and 2016.

Out of the 34 papers, there are 18 conference papers, 11 journal papers, and 5 workshop papers. The full data collected on each paper, including publishers, is available in our replication package[3].

We extract from each paper any language feature used to improve object-oriented design patterns. We use a broad definition of language features. The extracted data is, as expected, very diverse. Some features are closely related to specific programming paradigms (e.g., aspect-oriented pointcuts). Other features are more general and could be applied in many ways (e.g., mixins). To make sense of this data and obtain a global view, we create a mind map of these improvements that can be seen in Figure 4.

In this figure, nodes represent paradigms, paradigm applications and language features. Colours represent the type of node. This mind map helps navigate and make sense of the data. We classified language features into the paradigms to which they relate the most. However, some features could fit into multiple categories. When this happened, we classified the feature according to how it was presented in the

[2]PLoP entries include EuroPLoP (3 papers) and VikingPLoP (1 paper).
[3]https://www.ptidej.net/downloads/replications/ist22/

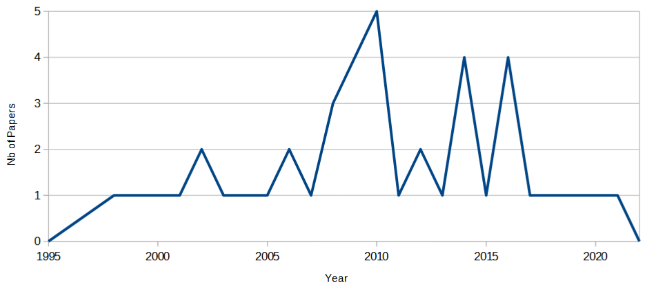

Figure 3: Papers distribution by year

study where it appeared. For example, we found different studies pertaining to Mixins in the context of both Aspect-Oriented Programming and Object-Oriented Programming, thus we added the feature to both paradigms. We explain our categorization in the following.

Purple nodes represent programming paradigms. We define a programming paradigm as a set of cohesive features for linking domain concepts with programming concepts and for organizing these programming concepts. We identified in our dataset four main paradigms: Object-oriented Programming, Functional Programming, Meta-programming and Reactive Programming.

Green nodes in the mind map represent language features. We define a language feature as an element of the syntax or grammar of a programming language that can be used to express a solution to a problem. For example, classes are a language feature of object-oriented languages used to represent real-life categorization of objects and create and manipulate instances of those objects. In our mind map, we assigned each language feature found in the studies to its underlying paradigm.

Table 4 offers a detailed view of the information discussed in this section. The table contains references to every paper in the study and can be used to find papers on a particular language feature or paradigm. We also specify the implementation languages presented in the papers for each of the features where available.

5.1. Meta-Programming

Meta-programming is a paradigm allowing programs to generate or modify their structure (either at compile, load-time, or run-time). It is possible to effectively implement new paradigms using Meta-programming, its classification is slightly different from that of the other paradigms. Directly under Meta-programming, we classify applications of Meta-Programming (as yellow nodes). These applications are specific usage of Meta-programming to create new languages. For example, we classify Aspect-Oriented Programming as an application of Meta-programming because it modifies the structure of the target program at compile-time. Meta-programming applications are paradigm extensions to a target language, rather than stand-alone paradigms. For example, Krishnamurthi et al. [S34] use the MzScheme language's meta-programming capabilities to implement Zo-

Publication Venue [# Papers]
Conference on Pattern Languages of Programs (PLoP) [4 papers]
European Conference on Object-Oriented Programming (ECOOP) [2 papers]
Special Interest Group on Programming Languages (SIGPLAN) [2 papers]
Symposium on Applied Computing (SAC) [2 papers]
Computer Languages, Systems & Structures [1 paper]
International Conference on Advanced Communication Control and Computing Technologies (ICACCCT) [1 paper]
International Conference on Computer Science and Information Technology (CSIT) [1 paper]
International Conference on Informatics and Systems (INFOS) [1 paper]
International Conference on Objects, Components, Models and Patterns [1 paper]
International Conference on Software Engineering Advances (ICSEA) [1 paper]
International Conference on Software Technology and Engineering (ICSE) [1 paper]
International Conference on Systems, Programming, Languages and Applications: Software for Humanity [1 paper]
International Journal of Software Engineering and Knowledge Engineering [1 paper]
International Journal of Software Innovation [1 paper]
International Symposium on Foundations of Software Engineering (FSE) [1 paper]
International Workshop on Context-Oriented Programming [1 paper]
Journal of Object Technology [1 paper]
Journal of Software [1 paper]
Journal of Systems and Software [1 paper]
Journal of the Brazilian Computer Society [1 paper]
Lecture Notes in Computer Science [1 paper]
Second Eastern European Regional Conference on the Engineering of Computer Based Systems [1 paper]
Software Engineering and Knowledge Engineering (SEKE) [1 paper]
Software Technology and Engineering Practice (STEP) [1 paper]
Software: Practice and Experience [1 paper]
Workshop on Aspects, components, and patterns for infrastructure software (ACP4IS) [1 paper]
Workshop on Generic Programming (WGP) [1 paper]
Workshop on Reuse in Object-Oriented Information Systems Design [1 paper]

Table 3
Publication Venues

Paradigm	Language Feature	Papers	# papers	Language
Functional Programming	Case Classes	[S1]	1	Scala
Functional Programming	Closures	[S2, S3]	2	Haskell
Functional Programming	Immutability	[S4]	1	Scala
Meta-Programming	AOP Annotations	[S5, S6]	2	Java
Meta-Programming	AOP Mixins	[S7, S8]	2	Java
Meta-Programming	AOP Join-point	[S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25]	17	Java
Meta-Programming	Layer Objects	[S26]	1	Pseudo-Java
Meta-Programming	Pattern Keywords	[S27, S28]	2	Java
Meta-programming	Reflection	[S29, S30, S31]	3	Java
Object-Oriented Programming	Chameleon Objects	[S3]	1	N/A
Object-Oriented Programming	Class Extension	[S12]	1	Java
Object-Oriented Programming	Default Implementation	[S3]	1	N/A
Object-Oriented Programming	Extended Initialization	[S3]	1	N/A
Object-Oriented Programming	Mixins	[S3, S4, S32]	3	Java, Scala
Object-Oriented Programming	Multiple Inheritance	[S4]	1	C++
Object-Oriented Programming	Object Interaction Styles	[S3]	1	N/A
Object-Oriented Programming	Subclassing members in a subclass	[S3]	1	N/A
Reactive Programming	Signals	[S33]	1	Scala

Table 4
Summary of papers on language features for improving design patterns implementations

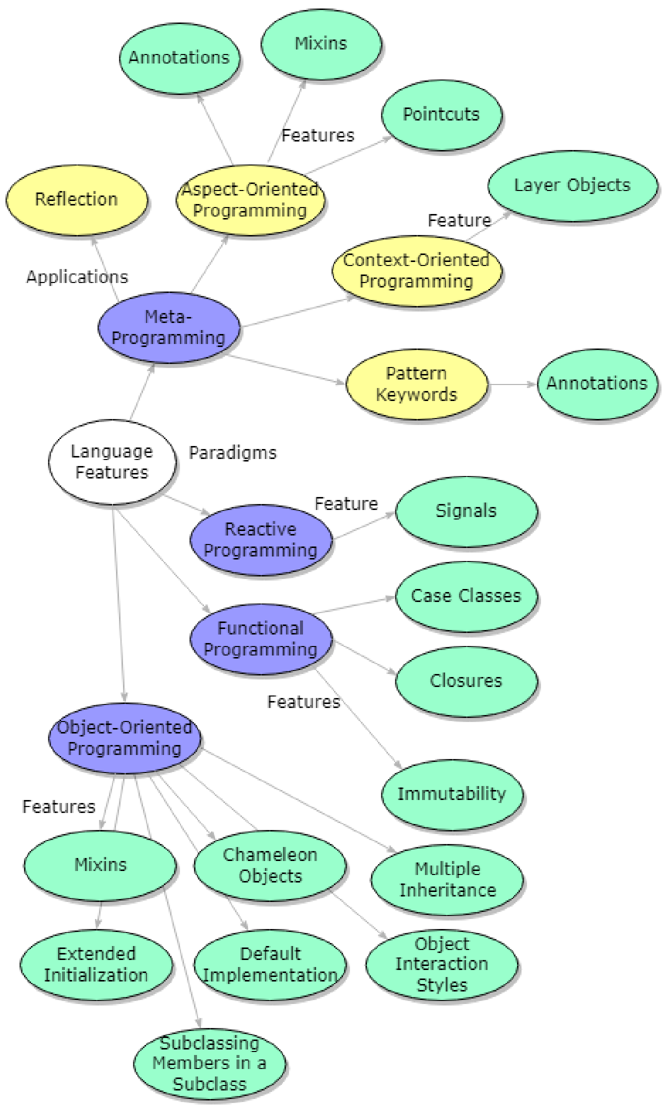

Figure 4: Mind map of paradigms, languages, and features for improving design patterns

diac, an extension to the language combining features from Object-Oriented Programming and Functional Programming to implement the *Visitor* pattern.

5.1.1. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a paradigm extension to procedural programming that was introduced in 1997 by Kiczales et al. [6]. The goal of AOP is to increase modularity by encapsulating cross-cutting concerns into code units called Aspects. Aspects are a language construct similar to classes in OOP. OOP/AOP implementations, like AspectJ [20], are the most popular in the literature, even though the original study did not associate AOP directly to OOP.

AOP works by making changes to the structure of the program at specific points. For example, it is possible to create an aspect which adds logging functionality to every method of a given class. Thus, with AOP, it is possible to extract recurring functionality to encapsulate them in aspects and avoid cluttering classes with unrelated, or cross-cutting, concerns.

Among the primary studies, we found three main implementations of AOP within OOP. The most common implementation [S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25] uses pointcuts and advices (also called the join-point model) to modify existing classes. Consider the following pointcut[4]:

```
pointcut setter(): target(Point) &&
                 (call(void setX(int)) ||
                  call(void setY(int)));
```

This pointcut designates every location where a method named `setX` or `setY`, with a single argument of type `int` and a return value of `void`, is called on an instance of the class `Point`.

The second part of this model is the advice, which designates the code inserted at every location referenced by the pointcut. Advices typically take the form of a regular method of the extended language, with the exception that it is usually decorated with a keyword such as `after`, `before`, or `around`, which dictates where the advice should be inserted relative to the pointcut.

The example shown is unrelated to design patterns implementations, since using AOP to implement design patterns typically involve multiple files and many lines of code, which would make it more difficult to see the important features of the paradigm.

Hannemann and Kiczales [S12] present a full implementation of every design patterns using AspectJ and compare them to Java implementations. They reported that they could remove many code duplications. For example, in the case of *Observer*, they extract the record-keeping of the observer list, as well as the `add` and `detach` methods, into an abstract aspect named `ObserverProtocol`.

AOP can be implemented in a different way using code annotations [S5, S6]. This approach is similar to the join-point model, replacing the pointcuts with annotations. This approach is used by the C# framework PostSharp [21]. Using this framework, one could implement Microsoft WPF `INotifyPropertyChanged` interface, which is an implementation of the *Observer* pattern, with the following annotation[5]:

```
[NotifyPropertyChanged]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Address Address { get; set; }
}
```

Using this approach, Giunta et al. [S5] and Jicheng et al. [S6] create annotations to represent the *Factory Method*, *Observer*, and *Singleton* design patterns.

[4]https://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html

[5]https://www.postsharp.net/postsharp

Yet another implementation of AOP uses mixins or an equivalent feature. Kuhlemann et al. [S7] use Jak, a Java language extension that combines classes and mixins, to achieve similar functionality to AspectJ. They implement every design pattern in Jak and compare their implementations to AspectJ implementations [S12]. They conclude that Jak has better support for modularizing cross-cutting concerns, but that the extension should be complementary to AspectJ. Axelsen et al. [S8] propose the concept of Package Templates to implement the *Observer* pattern and compare it to an AspectJ implementation. Package Templates function like module mixins (at the package level) in that they can introduce (generate) new classes, methods and attributes when used. They showed that their approach provides a sufficiently powerful framework to implement design patterns with minimal AOP mechanism compared to the join-point approach.

5.1.2. Layer Objects

Springer et al. [S26] present the concept of Layer Objects to implement the *Decorator*, *Observer*, and *Visitor* design patterns using Context-Oriented Programming, which allows changing the behaviour of objects based on scope (or context). A Layer Object can be activated (e.g., with the `with` keyword) within a specific scope, and will override the behaviour of specified classes within that scope. They present examples implemented in a Java-like pseudo-code.

With this feature, it is possible to add behaviour to an existing object (as per the *Decorator* pattern). For example, if we had a `Field` class, representing a tile in a game and wanted to add a "burning" by which a player takes damage when entering the field, the `BurningFieldDecorator` could be implemented as such:

```
class Field {
  def enter(player) {
    // Do something when a player enters the field.
  }
}

class BurningFieldDecorator {
  def damage = 15;

  def Field.enter(player) {
    player.health -= thisLayer.damage;
    proceed(player);
  }
}

// Usage example
def decorator = new BurningFieldDecorator();
field.activate(decorator);
```

5.1.3. Pattern Keywords

Some studies opted for introducing design patterns directly. Ghaleb et al. [S27] propose integrating *Decorator*, *Observer*, and *Singleton* as keywords. For example, a *Singleton* could be declared like this:

```
public singleton class A
{
    instantiate A as s1;

    public static void main(String[] args)
    {
        instantiate A as s2, s3;
    }
}
```

The `singleton` keyword marks the class `A` as a *Singleton*, of which its unique instance can be obtained using the `instantiate` keyword.

Zhang and Oliveira [S28] do something similar with the *Visitor* pattern by introducing EVF. EVF is a framework to implement *Visitors* in Java. It introduces new keywords in the form of annotations (such as `@Visitor`) to create data structures with internal or external visitors.

Adding keywords to a language to directly support design patterns requires either the use of Meta-programming or the creation of language extensions or domain-specific language. Unlike the approach presented by Krishnamurthi et al. [S34], where they used the macro feature of MzScheme to implement Zodiac, an extension to the language for implementing *Visitor*, the studies presented under this feature used third-party tools to add keywords to Java. The effect, however, is the same as if they had used macros to create these keywords (which Java does not support). Whether design patterns should be language features is however debatable [22].

5.1.4. Reflection

Reflection is an application of meta-programming. It allows developers to manipulate the data structures at runtime. For example, one could add a new method to an existing class, change the type of an attribute, etc. Unlike macros or Aspect-Oriented Programming, Reflection typically does not allow modifying code directly (e.g., inserting lines of code in the middle of a method). Reflection is available in mainstream programming languages like Java and C#.

Fortuin [S29] uses Reflection to implement the *Abstract Factory* pattern in Java. They use a static method that, based on the given arguments, constructs an object by accessing the proper class constructors and passing the arguments using a hashtable.

Mai and Champlain [S30] and Hussein et al. [S31] use Reflection to implement the *Visitor* pattern to overcome the limitation of Java of only supporting single dispatch (as discussed in Section 5.3.1 about Case Classes). Mai and Champlain [S30] defined a `findMethod` method that gets the correct `Visit` method on the visitor object. Reflection could also be used to automatically implement `accept` methods on the target, removing the cross-cutting concern from the visitable object.

5.2. Reactive Programming

Reactive Programming [S33] is a design paradigm which addresses asynchronous programming logic, especially concerning value updating and change propagation. It can be combined with Functional and Object-Oriented Programming to facilitate dealing with problems related to data updating (the same kind of problems targeted by the *Observer* design pattern). Scala offers libraries such as REScala [23] and Scala.react [24] to enable reactive programming in the language.

5.2.1. Signals

Signals are a feature of Reactive Programming that allows expressing dependencies among values [S33]. When a dependency of a signal changes, the expression defined by the signal is automatically recomputed from the new values of the dependencies. This functionality is similar to the *Observer* pattern. For example, take the following Scala code:

```
val a = Var(1)
val b = Var(2)
val s = Signal{ a() + b() }

println(s.getVal()) // 3
a() = 4
println(s.getVal()) // 6
```

When the value of variable a changes, so does the value of signal s. Thus, support of Signals and Reactive Programming in a language renders the *Observer* pattern obsolete.

5.3. Functional Programming

Functional Programming is a paradigm that uses pure functions to assemble higher-order functions (i.e., functions that receive other functions as arguments, such as map, reduce, etc.) to create programs. We classified studies that used features related to this paradigm: case classes [S1] (also called pattern matching), closures [S2, S3], and immutability [S34]. We admit that immutability is less of a feature and more of a property, but it is a property of pure function and functional languages usually support and sometimes enforce immutability. Immutability can also affect how design patterns are implemented [S34], which is why we chose to include it here.

5.3.1. Case Classes

Case Classes, also known as pattern matching, are a feature of functional programming languages. In Scala, Case Classes are used to create sets of classes that can be distinguished using the match keyword. Oliveira et al. [S1] use Scala to present many variations of the *Visitor* design patterns, as well as a library to use them. They use Case Classes to implement the accepting mechanism of this design pattern. They then use pattern matching to distinguish the different types to visit. Using their library, it is possible to implement the *Visitor* design pattern:

```
// Visitor structure for a Binary Tree.
trait Tree {
    def accept[R] (v :TreeVisitor[R]):R
}
case class Empty extends Tree {
    def accept[R] (v :TreeVisitor[R]):R = v.empty
}
case class Fork (x :int,l : Tree,r: Tree) extends Tree {
    def accept[R] (v :TreeVisitor[R]):R =
        v.fork (x,l,r)
}

trait TreeVisitor[R] {
    def empty :R
    def fork (x : int,l :Tree,r: Tree):R
}

// Concrete implementation of visitor to calculate
// the depth of the Tree.
def depth  = new CaseTree [External,int] {
    def Empty = 0
  def Fork (x : int,l :R[TreeVisitor],r:R[TreeVisitor]) =
        1+max (l.accept (this),r.accept (this))
}
```

This implementation does not require both accept and visit methods usually present in the *Visitor* implementation. They only implement an accept method.

5.3.2. Closures

Closures are the main feature of the Functional Programming paradigm. A Closure is an encapsulation of behaviour and data, much like Classes are in OOP. Unlike Classes, a Closure only encapsulates one function and its data cannot be directly accessed from outside the closure (with some exceptions). Closures are supported by most modern languages, such as Java, C#, C++, Python (with some caveats), JavaScript, Kotlin, etc. In most languages, Closures are created using some kind of lambda syntax such as :

```
var i = 42
var closure = (argument) => {
  // Some code which can use the i variable.
}
```

Gibbons [S2] present advanced uses of the map, fold, and other standard FP methods in Haskell. They discuss functional implementations of the *Composite* (using recursive data structures), *Iterator* (using map), *Visitor* (using fold), and *Builder* (using unfold) design patterns.

Compared to Closures, the *Command* design pattern has the advantage of being able to expose its data to the outside world if needed, allowing some degree of control over the behaviour of the object. Batdalov and Nikiforova [S3] propose to replace conventional functions with a generalization of the *Command* design pattern. Every function or method would become a Functor (an object with a single public method) and would make usage of the *Command* pattern obsolete.

5.3.3. Immutability

Immutability is a property of many Functional Programming languages. The Haskell language enforces immutability, which in turn makes the use of certain functional design patterns, such as monads, ubiquitous in the language.

Immutability has an effect on design patterns implementations. For example, we stated above that the *Command* design pattern is similar to Closures, with the exception that it can expose its internal data and mutators. In an immutable context, this advantage becomes unconsequential as the exposed data could not be interacted with in any significant way. In an immutable context, Closures make the *Command* pattern obsolete.

Sakamoto et al. [S4] present two variations on the *State* design pattern implemented using Scala. One of these is the Deeply Immutable State pattern, which makes use of the immutability features of Scala (such as the ability to easily clone Type Classes while modifying certain attributes, similar to Record Updating in OCaml[6]) for its implementation.

5.4. Object-Oriented Programming

Object-Oriented Programming is a paradigm that uses four general features to organize code: encapsulation, abstraction, inheritance, and polymorphism. Features classified under this paradigm directly affect OOP or require it to function. For example, the Chameleon Objects [S3] feature allows objects to dynamically change class. This requires and affects classes and objects, features which are directly related to OOP.

5.4.1. Chameleon Objects

Chameleon Objects is a feature proposed by Batdalov and Nikiforova [S3] to help with the implementation of the *State* and *Factory Method* design patterns. The feature allows an object to change class at runtime. While no language was presented in Batdalov and Nikiforova's paper, some languages allow such change already (e.g., Common Lisp, Perl). With this feature, it is possible to implement the *State* pattern by changing the class of an object when it changes state. The same feature could be used to allow a class constructor to change the instantiated object class depending on its arguments, effectively implementing a *Factory Method*.

In Common Lisp, a class can be changed using the following simple line:

```
(change-class target-object target-class)
```

The same can be achieved in Perl with the `bless` keyword:

```
bless $targetObject, 'Package::TargetClass';
```

5.4.2. Class Extension

Hanneman and Kiczales [S12] use AOP to implement design patterns using different features of AspectJ for different patterns. For *Adapter, Bridge, Builder, Factory Method, Template Method*, and *Visitor*, they specifically use the open class feature of AspectJ, which allows classes to be extended with new methods and attributes outside of their declaration (at compile-time).

With class extensions, it becomes possible to encapsulate duplicated code from a design pattern implementation into an extension. For example, we could extract the `accept` method of the *Visitor* pattern into a class extension, effectively allowing a *Visitor*-free implementation to exist independently of its pattern implementation.

5.4.3. Default Implementation

Default Implementation is a feature proposed by Batdalov and Nikiforova [S3]. It allows abstract classes to specify a default concrete implementation. This feature is similar to how the *Abstract Factory* pattern is implemented in Dependency Injection frameworks, such as Spring[7]. Beyond *Abstract Factory*, the authors argue this feature would help implementations of the *Builder*, *Bridge*, *Command*, and *Strategy* patterns.

5.4.4. Extended Initialization

Extended Initialization is a feature proposed by Batdalov and Nikiforova [S3] to help with the implementation of the *Builder* and *Fatory Method* patterns. This feature would allow object creation to be divided into multiple steps (effectively achieving the same functionality as the *Builder* pattern and rendering it obsolete).

5.4.5. Mixins

While we already introduced Mixins in the context of Aspect-Oriented Programming in Section 5.1.1, this feature can be used separately from AOP. Mixins are an extension of the OOP paradigm to combine classes together. Unlike inheritance, Mixins do not enforce an "is-a" relationship between the individual classes.

Burton and Sekirinski [S32] present implementations of the *Decorator*, *Proxy*, *Chain of Responsibility*, and *Strategy* design patterns using Mixins in Java. For example, a Decorator may be implemented by combining the Decorator with the target class, as such:

```
class Component { Operation(); }

class ConcreteComponent implements Component { ... };

class DecoratorMixA implements Component
                needs Component
    Operation() { ... Component.Operation() };

class DecoratorMixB implements Component
                needs Component
    Operation() { ... Component.Operation() };

// Usage
class Client {
    main() {
        ConcreteComponent cc =
           new ConcreteComponent with DecoratorMixA;
        extend cc with DecoratorMixB;

        cc.Operation();
    }
}
```

[6]https://dev.realworldocaml.org/records.html

[7]https://spring.io/

The authors declared that both decorators are of type Component but also require a concrete instance of Component as a Mixin. In the usage section, they mix the concrete instance `ConcreteComponent` with the `DecoratorMixA` class, and then extend it with `DecoratorMixB`, effectively creating an object which is a combination of all three classes.

Sakamoto et al. also use Mixins in their implementation of the *State* pattern in Scala. Batdalov and Nikiforova [S3] propose a concept of Responsibility Delegation, which would allow a class to delegate a part or all of its responsibility to another class (perhaps similar to the `DoesNotUnderstand` method in Smalltalk), achieving a behaviour similar to Mixins.

5.4.6. *Multiple Inheritance*

Many OOP languages only support single inheritance. This led to the introduction of interfaces to circumvent the limitation of an object only being able to be part of one hierarchy. Interfaces, however, do not typically allow reuse.

Some design patterns have duplicated code that we could factor into another class, to reduce code duplication and cross-cutting concerns. For example, one could extract the bookkeeping logic of the *Observer* pattern into another class and have observable objects inherit that class. However, in a single-inheritance context, this would "hijack" the only inheritance possibility for the observable object.

Sakamoto et al. [S4] present an implementation of the State design pattern in Java (using single inheritance) and C++ (using multiple inheritance). They evaluated that the Java implementation has code duplication, while the C++ implementation does not.

5.4.7. *Object Interaction Styles*

Object Interaction Styles is a feature proposed by Batdalov and Nikiforova [S3] to help with the implementation of the *Proxy*, *Observer*, and *Facade* patterns. They would allow different ways of interacting with objects through method calls. The default interaction style is synchronous calls, where a caller awaits the result of the method before continuing to the next instruction. Other interaction styles include asynchronous request/response, broadcast, and publish/subscribe. An example of this would be the async syntax in C# and JavaScript.

5.4.8. *Subclassing Members in a Subclass*

Batdalov and Nikiforova [S3] propose that OOP languages allow subclassing a member in a subclass to help with the implementation of the *Template Method* and *Visitor* patterns. A subclass may redefine one of its members by restricting its class.

To understand this feature, let us take the example from [S3] about Android development. Take the following pseudocode:

```
class Activity { ... }
class Fragment {
    ...
    Activity getActivity() { ... }
}
```

If we wanted to subclass `Activity` into a new class `MyActivity` and `Fragment` into `MyFragment`, we would end with the following subclasses:

```
class MyActivity extends Activity { ... }
class MyFragment extends Fragment {
    ...
    Activity getActivity() { ... }
}
```

However, if we knew the `getActivity` method of `MyFragment` could ever only return instances of `MyActivity`, we could not make the change, as the interface of the parent class `Fragment` requires this method to return the type `Activity`.

The proposed feature would allow us to make that change. The feature does not violate covariance or contravariance but could be considered unsafe in some cases [25].

5.5. Design Patterns

For each primary study, we extracted the design patterns for which improvements were suggested. Figure 5 shows the distribution of the number of studies by design pattern. As discussed in Section 4, we only considered patterns from the GoF book [1].

Figure 5 shows researchers' interest in design patterns. We notice that some patterns are more prevalent than others in our dataset. In particular, *Observer*, *Visitor*, and *Decorator* receive the most attention from researchers.

If we look at the number of features associated with each pattern, *Observer* and *Visitor* also come on top, but there is an interesting relationship at play here. While *Observer* has received overwhelmingly more attention from papers (20 papers against *Visitor* 13), they have the same number of associated features. Also, they have a similar amount of papers focusing solely on them ([S8, S9, S13, S14, S17, S33] for *Observer* and [S1, S23, S28, S30, S31, S34] for *Visitor*). Other patterns have very few papers that address only them. The only other two are *Abstract Factory* [S29] and *State* [S4].

5.6. Measures

We extracted the different measures used in the studies to compare the suggested improvements to design patterns to the classical implementations. Some studies did not measure or compare their implementation and were demonstrations of new language features or of a language extension. We observed 39 different measure names.

Some measures are more prevalent than others to measure improvements. The top ten measures are related to the usage of inheritance (DIT), coupling and cohesion (CBC, LCOO), concern diffusion (CDC, CDLOC, CDO), code size (LOC, NOA, WOC), and reusability. This matches the stated goal of design patterns [1].

Most of these measures (DIT, CBC, LCOO, LOC, WOC) come from the Chidamber and Kemerer suite of measures [26]. Concern diffusion measures relate to measuring improvements from Aspect-Oriented Programming regarding cross-cutting concerns [S11].

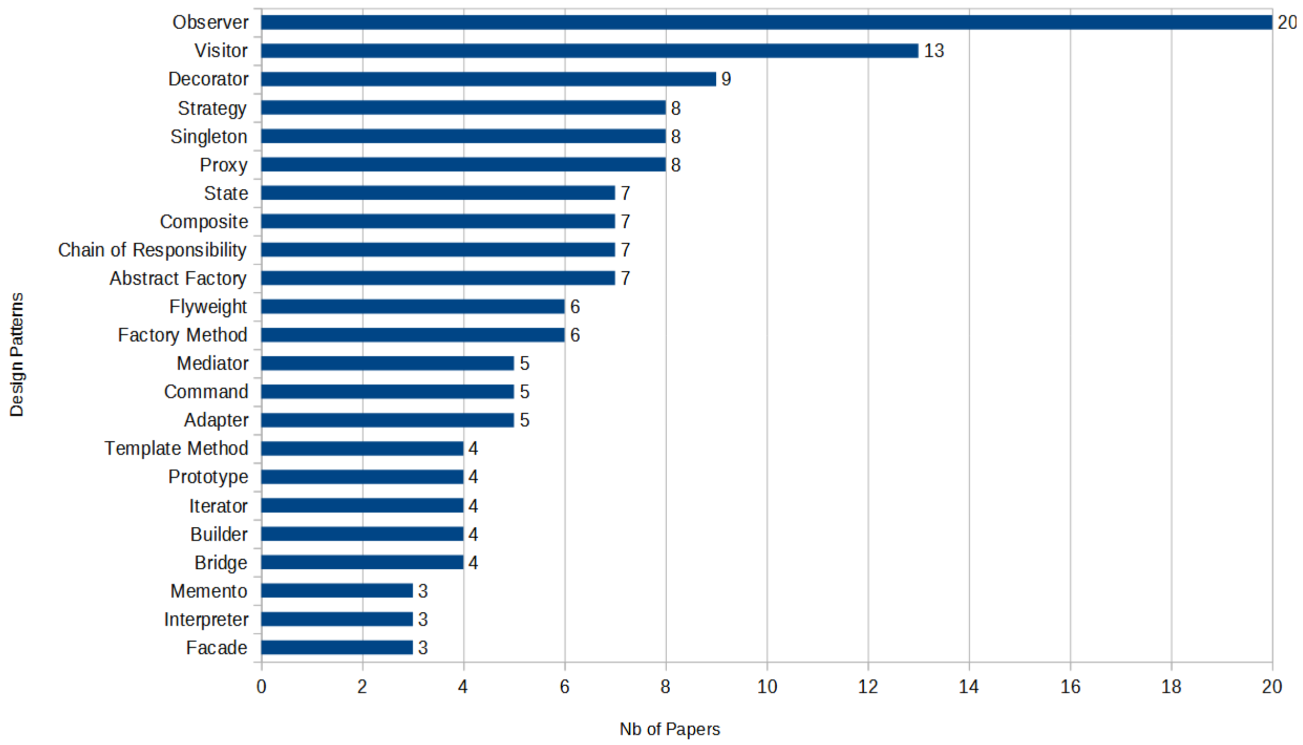

Figure 5: Number of papers by design pattern

Many of the other measures are unique to only one paper. For example, Teebiga and Velan [S10] use a measure they call lines of class code (LOCC), which they attribute to Ceccato and Tonella [27]. LOCC measures the number of lines of code within classes (as opposed to within aspects). The measure is similar to the number of lines of code (LOC), but the context in which it is used (i.e., to differentiate between aspect and class code) is different enough that we opted to classify it as its own entry.

Most of the measures extracted are used in AOP studies. The results presented by the studies were mixed. In general, AOP implementations of design patterns show smaller code size (LOC and NOA), a decrease in concern diffusion (CDC, CDO, CDLOC) and an increase in cohesion (LCOO) [S10, S11, S12, S20, S21, S24, S25]. However, some studies report AOP pattern implementations with increased code size and higher complexity (WOC) [S11, S24]. One study by Giunta et al. [S15] compared the execution time of AOP design patterns implementations to classical implementations and found that AOP implementations tend to be slightly slower. Note that despite DIT being the most popular measure, results in most studies showed no difference between the AOP implementations and the classical implementations.

Only three non-AOP papers make use of any measure at all [S4, S28, S33].

Zhang and Oliveira [S28] use lines of code to compare various *Visitor* implementations. They unsurprisingly find that the version using a Pattern Keyword, via annotations, has a smaller code size.

Salvaneschi et al. [S33] ask participants to answer specific questions to test their comprehension of Reactive Programming. They then measure their correctness in answering the questions (score of the participants on the task), their comprehension of the code (time required to complete the task) and the skill floor (a correlation between measured programming skill and performance during the task). They find no significant relationship with regards to comprehension and skill floor but find that the correctness of the participants is increased in the group analyzing code programmed using Reactive Programming.

Sakamoto et al. [S4] compare the amount of duplicated code and the number of classes and coupling between implementations of the *State* patterns. They find that using Multiple Inheritance to implement the pattern reduced the amount of code duplication and that using mixins reduced both code duplication and the number of classes and relations.

The other non-AOP papers were descriptive studies presenting a new feature without any comparison. They argued improvements in an informal way.

5.7. Experiments and Participation

We extracted out of the papers those that were case studies and experiments to answer our fourth research question: What experiments have been done on these language features?

We found 10 case studies and only one experiment out of the 34 studies in our data. The case studies compare a new feature against established OOP implementations. Case studies can be divided based on the source of their subjects: either in-vivo or in-vitro. In-vivo case studies are done on already-existing software, often available online as open-source projects. In-vitro case studies are done on software made specifically for the case study. Every case study in our analyzed studies was in-vitro because there are not many (if any) opportunities to find projects developed by third parties using new tools or methods proposed by a paper.

We found only one study that performed a controlled experiment. Salvaneschi et al. [S33] performed an empirical experiment on program comprehension of design patterns using reactive programming with 38 undergraduate students (which we discuss in Section 5.6).

These findings mean that there was no study involving practitioners (e.g., professional developers). The only experiment used students and every case study used a project developed in-vitro. We interpret this fact to mean that most of the new features proposed have not been extensively tested.

6. Discussion

6.1. Language Features

Our first research question is: *What language features have been suggested to improve design patterns implementations?*

We extracted 18 features from 34 selected papers. These features range across a variety of programming paradigms and improve design patterns implementations in various ways.

Many of these features (7 out of 18) were implemented as extensions to Java. In general, Java seems to be the most popular language to discuss improvements in design patterns. The next most popular language seems to be Scala, which is another JVM language, most often used to discuss functional programming features. Even when using dynamically-typed pseudo-code, Springer et al. [S26] describe it as "Java-like", even though to us it looks more like Python with brackets. There seems to be a prevalence of Java in design patterns and OOP research, which may be a concern because Java does not represent every OOP language, and many features available in other OOP languages, such as C#, Common Lisp, Kotlin, and Smalltalk, are not available in Java.

Most features stem from Meta-programming applications, which modify or insert code into OOP software. Features like Reflection and Pattern Keywords are used to reduce the amount of code duplication and code size of design patterns. It is not surprising that Meta-programming is a popular tool to address code duplication and size, as one of its main strengths is the ability to factor out code and remove duplication.

Aspect-Oriented Programming is a major part of the Meta-programming features. AOP and Layer Objects, while also reducing code duplication, focus on reducing cross-cutting concerns in design patterns [S12]. Despite AOP being a language and paradigm-agnostic feature (the original paper was presented using Common Lisp [6]), every paper in our dataset discusses it using various Java extensions. While there are solutions which borrows from AOP in other languages, such as the PostSharp library in C#, the paradigm itself does not seem to have gained much traction in other languages.

Some studies also presented domain-specific languages added on top of an existing language, such as Zodiac, an extension of MzScheme used to implement the *Visitor* pattern [S34].

Many features were also suggested to extend OOP. Chameleon Objects allow changing the class of an object at runtime. Mixins allow combining multiple classes together to facilitate reuse through composition. Multiple Inheritance solves some issues with code duplication by allowing multiple sources of reuse. Code reuse in pure OOP is usually limited to inheritance and subtyping mechanisms, so naturally, the features suggested interact with these mechanisms.

Some features come from Functional Programming. Case Classes in Scala are an implementation of pattern matching, which can be used to circumvent the issues of single dispatch languages and facilitate the implementation of the *Visitor* pattern. The feature has been gaining popularity of late, being added to existing languages such as C# and Java in the form of Records.

Closures are another functional feature that has been implemented in many OOP languages, such as Java, C++, and C#, but also part of older languages, such as Smalltalk (code blocks) and Common Lisp. They can make the implementation of many design patterns easier by storing behaviour as values. It is interesting, and telling, that the feature was already included in Smalltalk, one of the originators of OOP, but was then excluded from later implementations of the paradigm in Java and C++, only to be added back to these languages later.

Immutability, when combined with other features such as Closures, can render certain patterns obsolete (e.g., *Command*) and simplify the implementation of others (e.g., *State*). Patterns that are built around the idea of encapsulating changing state are made much simpler when state changes are eliminated entirely.

Finally, Signals and Reactive Programming have been created to solve the same underlying problem as the *Observer* pattern. Managing events and changes in a program is a common concern. Especially in GUI development, many platforms propose different approaches to this concern. Some, like Java Swing, make use of the *Observer* pattern. Others, like Microsoft WPF, propose a system based on the *Command* pattern. Reactive Programming incorporates change management at the language level.

Research Question #1

What language features have been suggested to improve design patterns implementations?
We catalogued 18 language features used to improve design patterns implementations. Meta-programming features are the most suggested, with many studies have written about Aspect-oriented Programming. However, our timeline analysis suggests that AOP is not as prevalent nowadays and that Functional Programming features are becoming more and more popular.

6.2. Design Patterns

The second research question is *RQ2: Which design patterns have the most associated language features suggestions?*

Our data indicate that some design patterns are more prevalent than others as targets for improvement. Some papers include every 23 GoF design patterns in their study (3 papers), but most analyze specific patterns.

We observe that the most prevalent patterns to improve are the *Observer*, *Visitor*, and *Decorator* design patterns. We could interpret pattern prevalence in different ways. Perhaps these patterns are those that have the most obvious flaws, and that is why many papers propose features to improve them. We thought perhaps the most prevalent patterns in research would also be the most used in practice, but that does not seem to be the case. *Observer* may well be the most used pattern of all, but it would be difficult to argue that *Visitor* is.

The fact that *Observer* and *Visitor* are the most prevalent patterns, that they have the most associated language features, and that they have the most papers focusing solely on them would indicate that the underlying problems these patterns are solving are of great interest to OOP.

In the case of *Observer*, there is a need in software systems for controlling data flow and change propagation. The *Observer* pattern as described in the GoF book [1], however, comes with some limitations. As discussed in Section 5.4.6 about Multiple Inheritance, the *Observer* implies duplication that cannot be easily abstracted using inheritance. It also has cross-cutting concerns [6, S12] because it requires the observed object to know and manage its observers. The proposed language features improve on these aspects.

In the case of *Visitor*, modern programming languages (such as Java, C++, and C#) lack support for dynamic dispatch on function parameters. In OOP, dynamic dispatch is only supported on the object on which a method is invoked (allowing the use of polymorphism). Some languages, such as the Common Lisp Object System (CLOS), support both OOP and dynamic dispatch on function arguments, effectively allowing multiple dispatch. In such languages, the visitor pattern becomes trivial as there is no longer any need for the accept/visit mechanism.

Most of the solutions proposed a way to circumvent the lack of dynamic dispatch on function arguments through the use of Reflection, Case Classes (pattern matching), or Aspects. Many studies propose ways to make the implementation of the *Visitor* easier by integrating it into the language using aspects [S12], factoring its complexity using reflection [S30, S31], or simplifying it using pattern matching [S1].

Research Question #2

Which design patterns have the most associated language features suggestions?
Research done to improve design patterns seems to favour specific patterns. *Observer*, *Visitor*, and *Decorator* (in order of appearances) are most often studied. In particular, *Observer* and *Visitor* appear to stem from a lack of language support of certain features (such as propagation of change and dynamic argument dispatch).

6.3. Measures

For the third research question, *RQ3: What measures have been used to evaluate the impact of these language features on design patterns implementations?*, we extracted the measures used by each paper and looked at their distribution.

The measures most often used were those related to reusability. In particular, two sets of measures were prevalent: the CK measures [26] and AOP-related measures [S11].

Most of the papers with measurements and comparisons were related to AOP. Most non-AOP papers were descriptive studies and offered no measurement or comparison of the performance of their proposed features.

For AOP papers, measure results were mixed. While many AOP implementations showed improvements with regard to concern diffusion, cohesion, and code size, some studies reported increased code size and higher complexity. Non-AOP papers showed more straightforward results, with reduced code size for Pattern Keywords, reduced code duplication for Multiple Inheritance and Mixins, and improved understandability for Reactive Programming.

The measures most used indicate that design patterns may be a source of complexity and that maintainability and understandability could be improved by making their implementation simpler using certain language features. There is also a concern about concern diffusion in design patterns. For example, the objects involved in the *Visitor* pattern have the added concern of being part of the pattern (i.e., having Visit or Accept methods). Some studies propose to extract this concern from the involved objects and abstract it into its own unit (e.g., an aspect).

Research Question #3

What measures have been used to evaluate the impact of these language features on design patterns implementations?
The measures most often used in the literature to evaluate improvements to object-oriented design patterns are measures related to maintainability and understandability, especially those proposed by Chidamber and Kemerer [26]. AOP-related papers also add measures related to concern diffusion, or cross-cutting concerns.

6.4. Empirical Studies

Our final research question concerns empirical studies: *RQ4: What experiments have been done on these language features?*

Most of the studies in our dataset are descriptive studies (23 papers) proposing or explaining a new feature or language extension. These typically do not offer concrete data to compare the new features with existing approaches.

There are also 10 case studies in our dataset. Every case study on the subject is done in-vitro, with the studied subjects created specifically for the case study. It would be difficult to find software "in the wild" using new features just proposed by a scientific paper. Because in-vivo case studies are impractical, controlled experiments are important to evaluate the effectiveness of new technology.

We only found one experiment done on evaluating improvement to design patterns implementations with 38 undergraduate students. As mentionned above and in Section 5.7, there is a clear need for more empirical experiments to evaluate the effects of proposed new features on design patterns implementations and on object-oriented development in general. In general, practitioners' participation was lacking. While experiments on students have their advantages, experiments should also be done on professional developers to gain better insight into the impact that the proposed features have on development.

Research Question #4

What experiments have been done on these language features?
Most of the studies are descriptive studies without comparison data. There are also many in-vitro case studies, with a lack of studies done on professional projects developed independently. We found only one experiment, which used student participants.

7. Recommendations

We list below notable or interesting findings and add our recommendations for future work.

Meta-programming features seem the most prevalent to implement new features to potentially improve OOP design patterns implementations. The past prevalence of the AOP paradigm indicates that it had good potential for improving OOP design patterns. Perhaps the introduction of another layer of complexity (i.e., aspects and the popular join-point mechanism used by AspectJ) on top of the OOP rebuked developers from adopting the paradigm. While AOP popularity has faded in recent years, it may still be of some use as a comparison point for new methods or improvements.

Meta-programming in general offers the possibility to extend languages by adding new features without the need for external tools. While some features have been prevalent in mainstream languages, such as Reflection in Java and C#, many Meta-programming features are absent in mainstream languages. For example, Lisp-style macros, allowing the effective addition of new keywords within a language, are seldom seen in modern languages.

While we found few works that used Functional Programming to improve OOP design patterns, the popularity of functional programming appears to be on the rise. As more functional programming features are added to OOP languages, it would be interesting to study how we can use those features to improve OOP design patterns. There certainly seems to be a trend in recent years to try to fuse OOP and FP together. We believe the impact of this is worth studying.

Most existing research seems to favour specific design patterns. *Observer*, *Visitor*, and *Decorator* were studied much more often than others. Thus, we have more data on approaches to improve the implementation of these specific patterns. Some design patterns only appear in studies concerned with all 23 GoF patterns (most likely for the sake of completeness). It would be interesting to study the less prevalent patterns to understand what kind of approaches would specifically improve those. Each design pattern solves a recurring design problem, so the community should strive to study these problems and, perhaps ideally, find features to solve them.

For reproducibility and comparability with existing research, when evaluating or comparing improvement to OOP design patterns, it would be beneficial to include measures pertaining to maintainability and understandability. In particular, the CK measures [26] are particularly prevalent in this kind of research. These include depth of inheritance tree, lack of cohesion in operations, coupling between components, number of lines of code, and weighted operations per component. AOP-related measures might be interesting to use also, for comparison with the many papers published proposing improvement using AOP. AOP measures are mainly focused on cross-cutting concerns.

However, these measures do not cover every possible improvement. Some improvements are difficult to measure using quantitative data. It would be important to explore less prevalent measures, or survey developers' opinions on code complexity and reusability and perform qualitative reviews.

Most papers we found were descriptive studies. More empirical studies on real-world case studies and experiments using professional developers would yield a deeper understanding of the techniques, tools and features that can be used to improve OOP design patterns, and OOP in general.

8. Threats to Validity

In this section, we discuss potential threats to the validity of this study. While we tried to keep the study and its results as objective as possible, there are some threats that need to be addressed. We divided the threats into Internal, External, Construct, and Conclusion Validity.

8.1. Interval Validity

The manual sorting, filtering, and compilation steps of the review were done by the first author, which increases the consistency of the results. Yet, it also introduces a threat to reliability and trustworthiness, so we added a second reviewer to help with the snowballing step and add some redundancy and a second viewpoint.

8.2. External Validity

Our results come from papers from a very diverse set of publication venues. The papers are not concentrated in popular venues concerning programming languages and design patterns, such as TOPLAS and POPL. We attribute this to the fact that this particular subject, proposing language features to improve OOP design patterns implementation, is more of a niche subject and less popular in prominent publications, and pertains more to software engineering than programming languages. Most of the results in our query that were from major publication venues tended to be about measuring the impact of design patterns, which was not within the scope of this study (and already has a mapping study [11] discussed in our related studies section).

8.3. Construct Validity

We defined our methodology as precisely as possible to make our results reproducible. The dataset used for this study is available online[8]. We took some decisions that could affect the results of our study. The query used has many exclusion keywords to obtain more precise results. We tried queries with fewer exclusion keywords, but they did not add more relevant results which made the manual filtering more difficult. We believe any missing relevant results that were lost because of this decision would have been caught during the snowballing step.

The manual sorting, filtering, and compilation steps of the review were done by the first author, which increases the consistency of the results. Yet, it also introduces a threat to reliability and trustworthiness, so we added a second reviewer to help with the snowballing step and add some redundancy and a second viewpoint.

8.4. Conclusion Validity

We believe that the previous threats are acceptable and that different choices would not significantly alter the results of this mapping study, whose goal is to identify trends in research on the subject of language features to improve OOP design patterns implementations. As such, we limit our conclusions to that goal and propose recommendations based on our findings.

9. Conclusion

In this paper, we presented a mapping study of the primary studies on language features to improve Object-Oriented Programming design patterns implementations. Our objective was to catalogue the language features that were used in the literature to improve OOP design patterns. We asked the four following research questions:

1. What language features have been suggested to improve design patterns implementations?
2. Which design patterns have the most associated language features suggestions?
3. What measures have been used to evaluate the impact of these language features on design patterns implementations?
4. What experiments have been done on these language features?

[8]https://www.ptidej.net/downloads/replications/ist22/

We devised and followed a methodology (which we describe in Section 4) that would yield the data needed to answer these questions and the main objective of the mapping study and create a catalogue of 18 language features claiming to improve design patterns implementations.

We performed a search query in three databases which yielded 874 papers, which we then assessed for quality and used for snowballing, resulting in a total of 34 primary studies. We then extracted relevant data from these 34 studies and discussed and catalogued our observations and their meaning.

We categorized the language features found in the papers into categories based on programming paradigms. We presented a summary of this map with every paper in our study, as well as a catalogue describing every feature extracted. We found that most of the research on the topic suggested approaches related to Meta-programming, with Aspect-oriented Programming often considered, although interest in it seems to have faded in recent years. The design patterns most often cited in papers on the topic were the *Observer*, *Visitor*, and *Decorator*. We also extracted every measure used in each paper into a table to find the most frequently used. We found that measures related to maintainability and understandability were most often used, with the Chidamber and Kemerer [26] measures being particularly prevalent.

With these findings, we contribute a catalogue of 18 language features proposed in the literature to improve Object-Oriented design patterns. These features mostly aim to improve maintainability and understandability by reducing concern diffusion and code duplication.

Our goal in finding language features that improve the implementation of design patterns in OOP is to improve the OOP paradigm itself. As design patterns are solutions to recurring problems, including features in an OOP language that makes it easier to solve these problems makes the paradigm and languages themselves easier to use for practical purposes.

This study can be used as a road map of existing literature that we intend to use to research improvement to the Object-Oriented Programming paradigm. We will look at what features are needed in a language to improve maintainability and understandability and reduce code size, coupling, and concern diffusion. We will use the identified measures to compare alternatives. We also intend to perform empirical studies, in particular experiments.

It may also be interesting to replicate this study on gray literature, with a similar systematic approach adapted to general public search engines (i.e., Google, Bing, DuckDuckGo, etc.). This would likely yield even more language features to improve design patterns, and perhaps different approaches to measuring improvements.

10. Acknowledgement

This work was supported by the Fonds de Recherche du Québec en Nature et Technologies (Grant No: 273582).

References

[1] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[2] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, 2004.

[3] Cheng Zhang and David Budgen. "What Do We Know About the Effectiveness of Software Design Patterns". In: *IEEE Transactions on Software Engineering* 38 (2012), pp. 1213–1231.

[4] Daniel von Dincklage. "Iterators and encapsulation". In: *Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems* (2000).

[5] H. Masuhara and R. Hirschfeld. "Classes as Layers: Rewriting Design Patterns with COP: Alternative Implementations of Decorator, Observer, and Visitor". In: *Proceedings of the 8th International Workshop on Context-Oriented Programming* (2016), pp. 21–26.

[6] G. Kiczales et al. "Aspect-oriented programming". In: *Proceedings of the European Conference on Object-Oriented Programming* (1997), pp. 220–242.

[7] Tobias Dürschmid. "Design pattern builder: a concept for refinable reusable design pattern libraries". In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (2016), pp. 45–46.

[8] Jan Hannemann and Gregor Kiczales. "Design pattern implementation in Java and AspectJ". In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming* (2002), pp. 161–173.

[9] Khalid Aljasser. "Implementing design patterns as parametric aspects using ParaAJ". In: *Computer Languages, Systems & Structures* 45 (2016), pp. 1–15.

[10] Peter Norvig. *Design Patterns in Dynamic Programming*. 1996. URL: https://norvig.com/design-patterns/design-patterns.pdf.

[11] Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos. "Research state of the art on GoF design patterns: A mapping study". In: *Journal of Systems and Software* 86.7 (2013), pp. 1945–1964. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.03.063. URL: https://www.sciencedirect.com/science/article/pii/S0164121213000757.

[12] Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos. "Design pattern alternatives: what to do when a GoF pattern fails". en. In: *Proceedings of the 17th Panhellenic Conference on Informatics - PCI '13*. Thessaloniki, Greece: ACM Press, 2013, p. 122. ISBN: 978-1-4503-1969-0. DOI: 10.1145/2491845.2491857. URL: http://dl.acm.org/citation.cfm?doid=2491845.2491857.

[13] L. Ferreira and C. M. F. Rubira. "The Reflective State Pattern". In: *Proceedings of the 5th Conference on Pattern Languages of Programs (PLOT '98)* (1998).

[14] M.E. Nordberg III. "The Reflective State Pattern". In: *Proceedings of the 3rd Conference on Pattern Languages of Programs (PLOT '96)* (1996).

[15] B. Bafandeh Mayvan, A. Rasoolzadegan, and Z. Ghavidel Yazdi. "The state of the art on design patterns: A systematic mapping of the literature". In: *Journal of Systems and Software* 125 (Mar. 2017), pp. 93–118. ISSN: 01641212. DOI: 10.1016/j.jss.2016.11.030.

[16] K. Peterson, S. Vakkalanka, and L. Kuzniarz. "Guidelines for conducting systematic mapping studies in software engineering: An update". In: *Information and Software Technology* 64 (2015), pp. 1–18.

[17] B.A. Kitchenham. "Procedures for Undertaking Systematic Reviews". In: *Joint Technical Report, Computer Science Department, Keele University (TR/SE-0401) and National ICT Australia Ltd. (0400011T.1)* (2004).

[18] B. Kitchenham and S. Charters. "Guidelines for Performing Systematic Literature Review in Software Engineering". In: *Technical Report EBSE 2007-001, Keele Univ. and Durham Univ. Joint Report* (2007).

[19] Elsevier. *Engineering Village*. URL: https://www.engineeringvillage.com/.

[20] *The AspectJ Project*. URL: https://www.eclipse.org/aspectj/.

[21] *PostSharp: C♯ design patterns without boilerplate*. URL: https://www.postsharp.net/.

[22] C. Chambers, B. Harrison, and J. Vlissides. "A Debate on Language and Tool Support for Design Patterns". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2000).

[23] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. "REScala: Bridging between Object-Oriented and Functional Style in Reactive Applications". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. Lugano, Switzerland: Association for Computing Machinery, 2014, pp. 25–36. ISBN: 9781450327725. DOI: 10.1145/2577080.2577083. URL: https://doi.org/10.1145/2577080.2577083.

[24] Ingo Maier and Martin Odersky. "Higher-Order Reactive Programming with Incremental Lists". In: *ECOOP 2013 – Object-Oriented Programming*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 707–731. ISBN: 978-3-642-39038-8.

[25] F. S. Løkke. "Scala & design patterns". In: *Master's thesis, University of Aarhus* (2009).

[26] S. Chidamber and C. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering, 20* (1994).

[27] M. Ceccato and P. Tonella. "Measuring the effects of software aspectization". In: *1st Workshop on Aspect Reverse Engineering* (2004).

Primary Studies

[S1] Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. "The visitor pattern as a reusable, generic, type-safe component". In: *ACM SIGPLAN Notices* 43.10 (Oct. 27, 2008), pp. 439–456. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1449955.1449799. URL: https://dl.acm.org/doi/10.1145/1449955.1449799 (visited on 11/01/2022).

[S2] Jeremy Gibbons. "Design patterns as higher-order datatype-generic programs". In: *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming - WGP '06*. the 2006 ACM SIGPLAN workshop. Portland, Oregon, USA: ACM Press, 2006, p. 1. DOI: 10.1145/1159861.1159863. URL: http://portal.acm.org/citation.cfm?doid=1159861.1159863 (visited on 11/01/2022).

[S3] Ruslan Batdalov and Oksana Nikiforova. "Towards Easier Implementation of Design Patterns". In: *The Eleventh International Conference on Software Engineering Advances*. ICSEA 2016. Rome, Italy: International Academy, Research, and Industry Association, 2016.

[S4] Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukukaza. "Extended design patterns in new object-oriented programming languages". In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*. SEKE. Eslevier, 2013, pp. 600–605.

[S5] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. "Using aspects and annotations to separate application code from design patterns". In: *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*. the 2010 ACM Symposium. Sierre, Switzerland: ACM Press, 2010, p. 2183. ISBN: 978-1-60558-639-7. DOI: 10.1145/1774088.1774548. URL: http://portal.acm.org/citation.cfm?doid=1774088.1774548 (visited on 11/01/2022).

[S6] Liu Jicheng, Yin Hui, and Wang Yabo. “A novel implementation of observer pattern by aspect based on Java annotation”. In: *2010 3rd International Conference on Computer Science and Information Technology*. 2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT 2010). Chengdu, China: IEEE, July 2010, pp. 284–288. ISBN: 978-1-4244-5537-9. DOI: 10.1109/ICCSIT.2010.5564893. URL: http://ieeexplore.ieee.org/document/5564893/ (visited on 11/01/2022).

[S7] Martin Kuhlemann et al. “A Multiparadigm Study of Crosscutting Modularity in Design Patterns”. In: *Objects, Components, Models and Patterns*. Ed. by Richard F. Paige and Bertrand Meyer. Red. by Will van der Aalst et al. Vol. 11. Series Title: Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 121–140. DOI: 10.1007/978-3-540-69824-1_8. URL: http://link.springer.com/10.1007/978-3-540-69824-1_8 (visited on 11/01/2022).

[S8] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. “A reusable observer pattern implementation using package templates”. In: *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software - ACP4IS ’09*. the 8th workshop. Charlottesville, Virginia, USA: ACM Press, 2009, p. 37. ISBN: 978-1-60558-450-8. DOI: 10.1145/1509276.1509286. URL: http://portal.acm.org/citation.cfm?doid=1509276.1509286 (visited on 11/01/2022).

[S9] Matthew F. Tennyson. “A study of the data synchronization concern in the Observer design pattern”. In: *2010 2nd International Conference on Software Technology and Engineering*. 2010 2nd International Conference on Software Technology and Engineering (ICSTE 2010). San Juan, PR, USA: IEEE, Oct. 2010, p. 5608911. ISBN: 978-1-4244-8667-0. DOI: 10.1109/ICSTE.2010.5608911. URL: http://ieeexplore.ieee.org/document/5608911/ (visited on 11/01/2022).

[S10] R Teebiga and S Senthil Velan. “Comparison of applying design patterns for functional and non-functional design elements in Java and AspectJ programs”. In: *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT). Ramanathapuram, India: IEEE, May 2016, pp. 751–757. ISBN: 978-1-4673-9545-8. DOI: 10.1109/ICACCCT.2016.7831740. URL: http://ieeexplore.ieee.org/document/7831740/ (visited on 11/01/2022).

[S11] Cláudio Sant’Anna et al. “Design patterns as aspects: a quantitative assessment”. In: *Journal of the Brazilian Computer Society* 10.2 (Nov. 2004), pp. 42–55. ISSN: 0104-6500. DOI: 10.1590/S0104-65002004000300004. URL: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-65002004000300004&lng=en&nrm=iso&tlng=en (visited on 11/01/2022).

[S12] Jan Hannemann and Gregor Kiczales. “Design pattern implementation in Java and aspectJ”. In: *ACM SIGPLAN Notices* 37.11 (Nov. 17, 2002), pp. 161–173. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/583854.582436. URL: https://dl.acm.org/doi/10.1145/583854.582436 (visited on 11/01/2022).

[S13] Tobias Dürschmid. “Design pattern builder: a concept for refinable reusable design pattern libraries”. In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. SPLASH ’16: Conference on Systems, Programming, Languages, and Applications: Software for Humanity. Amsterdam Netherlands: ACM, Oct. 20, 2016, pp. 45–46. ISBN: 978-1-4503-4437-1. DOI: 10.1145/2984043.2998537. URL: https://dl.acm.org/doi/10.1145/2984043.2998537 (visited on 11/01/2022).

[S14] J. Borella. “The observer pattern using aspect oriented programming”. In: *Proceedings of the Viking Pattern Languages of Programs*. Viking PLOP. 2003.

[S15] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. “AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC ’12*. the 27th Annual ACM Symposium. Trento, Italy: ACM Press, 2012, p. 1243. ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2231971. URL: http://dl.acm.org/citation.cfm?doid=2245276.2231971 (visited on 11/01/2022).

[S16] N. El Maghawry and A. R. Dawood. “Aspect oriented GoF design patterns”. In: *The 7th International Conference on Informatics and Systems*. INFOS. 2010, pp. 1–7.

[S17] Jose M. Felix and Francisco Ortin. “Aspect-Oriented Programming to Improve Modularity of Object-Oriented Applications”. In: *Journal of Software* 9.9 (Sept. 1, 2014), pp. 2454–2460. ISSN: 1796-217X. DOI: 10.4304/jsw.9.9.2454-2460. URL: http://ojs.academypublisher.com/index.php/jsw/article/view/13246 (visited on 11/01/2022).

[S18] Marc Bartsch and Rachel Harrison. “Design Patterns with Aspects: A case study”. In: *Proceedings of the 12th European Conference on Pattern Languages of Programs*. EuroPLoP ’2007. ACM, 2007, pp. 797–810.

[S19] Khalid Aljasser. “Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns”. In: *Computer Languages, Systems & Structures* 45 (Apr. 2016), pp. 1–15. ISSN: 14778424. DOI: 10.1016/j.cl.2015.11.002.

[S20] Pavol Baca and Valentino Vranic. “Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns”. In: *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*. 2011 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2011). Bratislava, Slovakia: IEEE, Sept. 2011, pp. 19–26. ISBN: 978-1-4577-0683-7. DOI: 10.1109/ECBS-EERC.2011.13. URL: http://ieeexplore.ieee.org/document/6037510/ (visited on 11/01/2022).

[S21] João L. Gomes and Miguel P. Monteiro. “Design pattern implementation in object teams”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC ’10*. the 2010 ACM Symposium. Sierre, Switzerland: ACM Press, 2010, p. 2119. ISBN: 978-1-60558-639-7. DOI: 10.1145/1774088.1774534. URL: http://portal.acm.org/citation.cfm?doid=1774088.1774534 (visited on 11/01/2022).

[S22] M.L. Bernardi and G.A. Di Lucca. “Improving Design Pattern Quality Using Aspect Orientation”. In: *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP’05)*. 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP’05). Budapest, Hungary: IEEE, 2005, pp. 206–218. ISBN: 978-0-7695-2639-3. DOI: 10.1109/STEP.2005.14. URL: http://ieeexplore.ieee.org/document/1691649/ (visited on 11/01/2022).

[S23] Q. Hachani and D. Bardou. “Using aspect-oriented programming for design patterns implementation”. In: *Workshop on Reuse in Object-Oriented Information Systems Design*. 2002.

[S24] Alessandro Garcia et al. “Modularizing Design Patterns with Aspects: A Quantitative Study”. In: *Transactions on Aspect-Oriented Software Development I*. Ed. by Awais Rashid and Mehmet Aksit. Vol. 3880. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 36–74. DOI: 10.1007/11687061_2. URL: http://link.springer.com/10.1007/11687061_2 (visited on 11/01/2022).

[S25] Nelio Cacho et al. “Blending design patterns with aspects: A quantitative study”. In: *Journal of Systems and Software* 98 (Dec. 2014), pp. 117–139. ISSN: 01641212. DOI: 10.1016/j.jss.2014.08.041.

[S26] Matthias Springer, Hidehiko Masuhara, and Robert Hirschfeld. “Classes as Layers: Rewriting Design Patterns with COP: Alternative Implementations of Decorator, Observer, and Visitor”. In: *Proceedings of the 8th International Workshop on Context-Oriented Programming*. ECOOP ’16: European Conference on Object-Oriented Programming. Rome Italy: ACM, July 17, 2016, pp. 21–26. ISBN: 978-1-4503-4440-1. DOI: 10.1145/2951965.2951968. URL: https://dl.acm.org/doi/10.1145/2951965.2951968 (visited on 11/01/2022).

[S27] Taher Ahmed Ghaleb, Khalid Aljasser, and Musab A. Alturki. “An Extensible Compiler for Implementing Software Design Patterns as Concise Language Constructs”. In: *International Journal of Software Engineering and Knowledge Engineering* 31.7 (July 2021), pp. 1043–1067. ISSN: 0218-1940, 1793-6403. DOI: 10.1142/S0218194021500327. URL: https://www.worldscientific.com/doi/abs/10.1142/S0218194021500327 (visited on 11/01/2022).

[S28] Weixin Zhang and Bruno C. D. S. Oliveira. “EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse”. In: (2017). In collab. with Marc Herbstritt. Artwork Size: 32 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 32 pages. DOI: 10.4230/LIPICS.ECOOP.2017.29. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7274/ (visited on 11/01/2022).

[S29] Harold Fortuin. “A Modern, Compact Implementation of the Parameterized Factory Design Pattern.” In: *The Journal of Object Technology* 9.1 (2010), p. 57. ISSN: 1660-1769. DOI: 10.5381/jot.2010.9.1.c5. URL: http://www.jot.fm/contents/issue_2010_01/column5.html (visited on 11/01/2022).

[S30] Yun Mai and Michel Champlain. “Reflective Visitor Pattern”. In: *European Conference on Pattern Languages of Programs*. EuroPLoP. ACM, 2001, pp. 299–316.

[S31] Bilal Hussein, Aref Mehanna, and Yahia Rabih. “Visitor Design Pattern Using Reflection Mechanism”. In: *International Journal of Software Innovation* 8.1 (Jan. 1, 2020), pp. 92–107. ISSN: 2166-7160, 2166-7179. DOI: 10.4018/IJSI.2020010106. URL: https://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJSI.2020010106 (visited on 11/01/2022).

[S32] Eden Burton and Emil Sekerinski. “Using dynamic mixins to implement design patterns”. In: *Proceedings of the 19th European Conference on Pattern Languages of Programs - EuroPLoP ’14*. the 19th European Conference. Irsee, Germany: ACM Press, 2014, pp. 1–19. ISBN: 978-1-4503-3416-7. DOI: 10.1145/2721956.2721991. URL: http://dl.acm.org/citation.cfm?doid=2721956.2721991 (visited on 11/01/2022).

[S33] Guido Salvaneschi et al. “An empirical study on program comprehension with reactive programming”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT/FSE’14: 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering. Hong Kong China: ACM, Nov. 11, 2014, pp. 564–575. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635895. URL: https://dl.acm.org/doi/10.1145/2635868.2635895 (visited on 11/01/2022).

[S34] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. “Synthesizing Object-Oriented and Functional Design to Promote Re-Use”. In: *Proceedings of the 12th European Conference on Object-Oriented Programming*. ECCOP ’98. ACM, 1998, pp. 91–113.

A. Tables

Table 5 contains the reference to every study based on the design patterns they discuss and the number of features associated with these design patterns.

Table 6 shows the usage of these measures by number of papers. The last column cross-references which language features were measured using the metric. For quantitative metric, a symbol follows the name of the language feature showing the presented results in the studies. An arrow pointing up ($\Uparrow$) signifies the results showed the measure increasing with usage of the language feature. An arrow pointing down ($\Downarrow$) shows the opposite. An approximation ($\approx$) signifies there was no significant difference found for the measure. Finally, a crossed box ($\boxtimes$) signifies no comparison was done with a classical implementation of the design patterns. This may be because the measure cannot be applied to the classical implementation (e.g., Crosscutting Degree of an Aspect) or because the paper used the measure to compare two non-classical implementations.

Design Pattern	# Features	Papers	# Papers
Abstract Factory	4	[S3, S7, S11, S12, S20, S21, S29]	7
Adapter	4	[S3, S7, S10, S12, S21]	5
Bridge	5	[S3, S7, S12, S21]	4
Builder	6	[S2, S3, S12, S21]	4
Chain of Responsibility	4	[S3, S7, S12, S16, S21, S22, S32]	7
Command	4	[S3, S7, S12, S21, S22]	5
Composite	4	[S2, S3, S7, S12, S15, S18, S21]	7
Decorator	6	[S3, S7, S12, S18, S19, S21, S26, S27, S32]	9
Facade	4	[S3, S12, S21]	3
Factory Method	6	[S3, S5, S7, S12, S15, S21]	6
Flyweight	3	[S3, S7, S12, S15, S20, S21]	6
Interpreter	2	[S7, S12, S21]	3
Iterator	3	[S2, S7, S12, S21]	4
Mediator	3	[S3, S7, S11, S12, S21]	5
Memento	2	[S7, S12, S21]	3
Observer	9	[S3, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S17, S18, S19, S21, S22, S26, S27, S33]	20
Prototype	2	[S7, S11, S12, S21]	4
Proxy	7	[S3, S5, S7, S12, S15, S20, S21, S32]	8
Singleton	4	[S5, S7, S12, S19, S20, S21, S22, S27]	8
State	6	[S3, S4, S7, S11, S12, S21, S22]	7
Strategy	4	[S3, S7, S10, S11, S12, S16, S21, S32]	8
Template Method	4	[S3, S7, S12, S21]	4
Visitor	9	[S1, S2, S3, S7, S12, S18, S21, S23, S26, S28, S30, S31, S34]	13

Table 5
Language features per design patterns

Measures	Type	# Papers	Language Feature
Depth of Inheritance Tree (DIT)	Quantitative	6	AOP≈
Coupling Between Components (CBC)	Quantitative	5	AOP⇕
Lack of Cohesion in Operations (LCOO)	Quantitative	5	AOP⇕
Lines of Code (LOC)	Quantitative	5	AOP⇕, Pattern Keywords⇓
Concern Diffusion over Components (CDC)	Quantitative	4	AOP⇕
Concern Diffusion over LOC (CDLOC)	Quantitative	4	AOP⇕
Concern Diffusion over Operations (CDO)	Quantitative	4	AOP⇕
Number of Attributes (NOA)	Quantitative	4	AOP⇕
Weighted Operations per Component (WOC)	Quantitative	4	AOP⇕
Reusability	Qualitative	3	AOP
Composition Transparency	Qualitative	2	AOP
Coupling on Intercepted Modules (CIM)	Quantitative	2	AOP⊠
Duplicated Code (DC)	Quantitative	2	Mixins⇓, Multiple Inheritance⇓. AOP (qualitative)
Locality	Qualitative	2	AOP
Modularity	Qualitative	2	AOP
Unpluggability	Qualitative	2	AOP
Weighted Operations in Module (WOM)	Quantitative	2	AOP⇕
Abstract-Pattern-Reusability	Qualitative	1	AOP
Binding-Reusability	Qualitative	1	AOP
Cohesion	Qualitative	1	AOP
Comprehension	Quantitative	1	Reactive Programming≈
Correctness	Quantitative	1	Reactive Programming⇑
Coupling Between Modules (CBM)	Quantitative	1	AOP⊠
Coupling on Advice Execution (CAE)	Quantitative	1	AOP⊠
Coupling on Field Access (CFA)	Quantitative	1	AOP⊠
Coupling on Method Call (CMC)	Quantitative	1	AOP⊠
Cross-cutting concerns	Qualitative	1	AOP
Crosscutting Degree of an Aspect (CDA)	Quantitative	1	AOP⊠
Encapsulation	Qualitative	1	AOP
Execution time	Quantitative	1	AOP⇑
Generalization (Inheritance)	Qualitative	1	AOP
Indirection	Qualitative	1	AOP
Lines of Class Code (LOCC)	Quantitative	1	AOP⇓
Number of Children (NoC)	Quantitative	1	AOP≈
Number of classes and relations (NOCR)	Quantitative	1	Mixins⇓
Response For a Module (RFM)	Quantitative	1	AOP⊠
Separation of concerns	Qualitative	1	AOP
Skill floor	Quantitative	1	Reactive Programming≈

Table 6
Measures by usage in papers