JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Smells to avoid when you seek information about GoF patterns on the Web

Apostolos Zarras and Panos Vassiliadis
Department of Computer Science and Engineering, University of Ioannina, Greece

ABSTRACT The target audience of this paper is developers looking for information about the Gang of Four (GoF) patterns on the Web. The information we find on the Web about the GoF patterns may deviate from the original definitions of the patterns. In a prior study, we found intent deviations, and missing, incomplete, and erroneous participants. Depending on the pattern, some deviations are more critical than others. In this paper, we provide a more detailed analysis of the deviations that occur in the information we find on the Web for each pattern. The result of the study is a catalog of the most important deviations, which we call *smells*. The catalog includes smells that compromise the intent of the patterns, smells that violate the abstract coupling principle, and smells in which the patterns are not used in the right context.

KEYWORDS GoF design patterns, deviating defitions, deviating examples.

1. Introduction

The Gang of Four (GoF) patterns provide generic solutions to frequently encountered object-oriented design problems. Gamma, Helm, Johnson, and Vlissides introduced the patterns in a seminal book (Gamma et al. 1994), published in 1994. Over the years, the book has been translated into many languages and more than half a million copies of the book have been sold [1].

The impact of GoF patterns on software has been the subject of several empirical studies and experiments (Mayvan et al. 2017). The results of these efforts showed that the use of patterns improves flexibility, facilitates maintenance, and reduces proneness to code smells (Prechelt et al. 2001, 2002; Walter & Alkhaeir 2016; Alfadel et al. 2020; Almadi et al. 2021). However, the incorrect use of GoF patterns can negatively affect software quality. In previous studies, we observed several issues and mistakes that compromise the benefits of the patterns (A. V. Zarras 2020; A. Zarras 2021).

The correct use of GoF patterns involves understanding their purpose and structure. In the past, the only source of information about the patterns was the GoF book. Today, the situation is different; many popular Web sites provide information about the GoF patterns. The problem with this plenitude of information is that it does not always comply with the original pattern definitions, given in the GoF book.

In a previous effort, we studied a corpus of (1) *pattern definitions* and (2) *pattern instances* that *illustrate the use of GoF patterns*, collected from four popular Web sites (A. V. Zarras & Vassiliadis 2023). Our study revealed that the information provided by the Web sites deviates from the original definitions of the patterns. Specifically, we observed *intent deviations* in pattern definitions; i.e., the description of the intent of the patterns does not reflect their actual purpose. We further found *missing participants*; i.e., certain pattern definitions/instances do not include all participants specified in the original definitions of the patterns. We also found *incomplete participants*; i.e., participants that do not provide the complete set of methods described in the original definitions of the patterns. Finally, we observed *erroneous participants*; i.e., participants that do not behave as specified in the original definitions of the patterns. Regarding the *frequency of deviations*, we found that deviations in the definitions of the patterns are rare. On the other hand, deviating pattern instances are much more frequent. Missing participants are the most frequent deviations, followed by incomplete participants, erroneous participants, and intent deviations.

JOT reference format:
Apostolos Zarras and Panos Vassiliadis. *Smells to avoid when you seek information about GoF patterns on the Web.* Journal of Object Technology. Vol. 25, No. 02, 2026. http://dx.doi.org/10.5381/jot.2026.25.02.a3
[1] en.wikipedia.org/wiki/Design_Patterns

In summary, the above results provide a high-level view of the deviations that occur in the information we find on the Web about the Gof patterns. This high-level view is good enough to satisfy our scientific curiosity concerning the adherence of the information we find on the Web to the original GoF pattern definitions.

However, in practice, a developer looking for information on the Web about a pattern that he wants to use in his code would like a more detailed view of the deviations that concern this pattern. To address this issue, we study pattern instances that illustrate the use of the Gof patterns, found in five popular Web sites. Our data set extends the one we used in our previous effort (A. V. Zarras 2020; A. Zarras 2021). The output of our research is *a catalog of important deviations, which we call smells*. The catalog lists smells that *jeopardize the intent of the patterns*, smells that *violate abstract coupling* because they omit key abstractions, and smells in which the *patterns are not used in the appropriate context*. We believe that the smells we observed could very well appear in instances of patterns involved in real contexts. For this reason, we provide a detailed analysis of them, hoping that people, specifically young, inexperienced developers, will avoid them.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 discusses the setup of our study, defines the notion of smells, and introduces the template that we adopt for their specification. Sections 4, 5, and 6 discuss the smells that we observed in our study, grouped with respect to the different categories of GoF patterns, namely creational, structural and behavioral patterns. Finally, Section 7 summarizes the contribution of this paper.

2. Related Work

The related work on design patterns is extensive, with dedicated communities and forums for researchers and practitioners [2,3]. Mayvan et al. (Mayvan et al. 2017), provide a thorough systematic mapping of the state of the art in design patterns. According to their study, research in design patterns can be categorized into five major sub-areas: pattern development, specification, usage, mining, and quality evaluation.

Our research focuses on pattern quality evaluation. This area encompasses studies that investigate the impact of pattern usage on software quality, as well as efforts that evaluate the correctness of pattern instances.

The impact of design patterns on software quality has been investigated in several empirical studies. For example, Prechelt et al. (Prechelt et al. 2001) found that the design patterns improve flexibility, helping maintenance efforts. In a subsequent study, Prechelt et al. (Prechelt et al. 2002) observed that the use of design patterns in conjunction with specialized comments facilitates maintenance tasks. Vokac (Vokác 2004) found that the mere use of patterns does not guarantee a low defect rate. Bieman et al. (Bieman et al. 2003) observed that pattern classes are more susceptible to changes. According to Aversano et al. (Aversano et al. 2007), pattern classes are more susceptible to

[2] hillside.net
[3] www.europlop.net

Table 1 Corpus of patterns instances.

	Corpus	Source Making	Refactoring Guru	Tutorials Point	Java T Point	Wikipedia	ALL
Creational	Abstract Factory	7	10	1	1	1	20
	Builder	5	10	1	1	1	18
	Factory Method	6	10	1	1	5	23
	Prototype	7	10	1	1	1	20
	Singleton	4	10	1	1	1	17
Structural	Adapter	6	10	1	1	3	21
	Bridge	5	10	1	1	6	23
	Composite	9	10	1	1	1	22
	Decorator	8	10	1	1	8	28
	Façade	5	10	1	1	1	18
	Flyweight	7	10	1	1	3	22
	Proxy	6	10	1	1	0	18
Behavioral	Chain of Resp.	6	10	1	1	1	19
	Command	7	10	1	1	1	20
	Interpreter	5	0	1	1	1	8
	Iterator	6	10	1	1	1	19
	Mediator	6	10	1	1	3	21
	Memento	5	10	1	1	4	21
	Observer	8	10	1	1	8	28
	State	9	10	1	1	0	21
	Strategy	5	10	1	1	1	18
	Template Method	5	10	1	1	1	18
	Visitor	6	10	1	0	4	21
	Total	143	220	23	22	56	464

Table 2 Deviations of patterns instances and density (i.e., number of deviations over number of instances).

Patterns	# Instances	Missing Participants	Incomplete Participants	Erroneous Participants	Density
Abstract Factory	20	12	0	5	0.85
Builder	18	4	0	8	0.67
Factory Method	23	9	0	1	0.43
Prototype	20	13	0	0	0.65
Singleton	17	0	0	0	0.00
Adapter	21	17	0	0	0.81
Bridge	23	1	0	0	0.04
Composite	22	15	31	1	2.14
Decorator	28	5	0	0	0.18
Façade	18	0	0	0	0.00
Flyweight	22	40	1	0	1.86
Proxy	18	3	0	0	0.17
Chain of Responsibility	19	11	0	1	0.63
Command	20	11	0	0	0.55
Interpreter	8	11	0	0	1.38
Iterator	19	17	26	0	2.26
Mediator	21	18	0	6	1.14
Memento	21	1	2	15	0.86
Observer	28	22	22	0	1.57
State	21	0	0	6	0.29
Strategy	18	1	0	2	0.17
Template Method	18	0	0	0	0.00
Visitor	21	7	0	0	0.33
ALL	464	218	82	45	0.74

Table 3 Balanced Pattern Specification Language (BPSL) notation.

BPSL Domains	Intent
C	A (domain) set of all classifiers (classes and interfaces) involved in the pattern structure.
A	A (domain) set of all attributes defined in the classifiers that belong to **C**.
M	A (domain) set of all methods defined in the classifiers that belong to **C**.
O	A (domain) set of objects involved in the specification of actions.
BPSL Relations	**Intent**
Defined-in(m, c) **Defined-in(m, i)**	Denotes that a method m is defined in a class c (resp. interface i).
Defined-in(a, c)	Denotes that an attribute a is defined in class c.
Reference-to-one(c1, c2) **Reference-to-one(c1, i)**	Denotes that a class c1 defines a member that belongs to (resp. implements) a class c2 (resp. interface i).
Reference-to-many(c1, c2) **Reference-to-many(c1, i)**	Denotes that classifier c1 defines many members that belong to (resp. implement) a class c2 (resp. interface i).
Inheritance(c1, c2)	Indicates that a class c1 inherits from a class c2.
Creation(c1, c2)	Indicates that a class c1 creates instance(s) of class c2.
Creation(m, c)	Denotes that a method m creates instance(s) of class c2.
Invocation(m1, m2)	Denotes that a method m1 invokes a method m2.
Instance(o, c)	Denotes that object o belongs to class c.
Additional Permanent Relations	**Intent**
Implements(c, i)	Indicates that a class c implements an interface i.
Modifier(m, am)	Specifies the access modifier am for a method m. The possible values of am are public, private, or protected.
Additional Temporal Relations	**Intent**
Association(c1, c2) **Association(c1, i)**	This relation is weaker than Reference-to that denotes part-whole associations. Specifically, it specifies temporal associations in which instances of class c1 collaborate with instances that belong (resp. implement) class c1 (resp. interface i). The instances of of c2 (resp. i) may be parameters passed to methods defined in c1 or local variables used in methods of c1. Multiplicities are intentionally omitted because they depend on the context in which a pattern is used.

change, whereas the classes that depend on them are less so. Further research efforts investigate the relationship between design patterns and code smells. Specifically, Walter and Alkhaeir (Walter & Alkhaeir 2016) found that the classes involved in design patterns are less prone to smells than the ones not involved in design patterns. Alfadel et al. reported similar findings. (Alfadel et al. 2020). Almadi et al. (Almadi et al. 2021) conducted a comprehensive systematic review on the relationship between design patterns and code smells.

Regarding the correctness of design patterns, Izurieta and Bieman (Izurieta & Bieman 2013) introduced the concepts of rot and grime. Design rot refers to the degradation of a pattern's structural integrity as a result of changes over subsequent software releases that compromise the pattern's intent. Design grime refers to decay from unrelated features added to classes involved in a pattern, without compromising the pattern's intent. Dale and Izurieta (Dale & Izurieta 2014) highlighted that certain types of design grime result in a higher technical debt, while Reimanis and Izurieta (Reimanis & Izurieta 2019) looked for correlations between different types of grime. Feitosa et al. (Feitosa et al. 2017) observe a linear accumulation of design grime in five software systems, depending on patterns and developers. In a subsequent study (Feitosa et al. 2018), they observed correlations between grime accumulation and decreased performance, security, and correctness.

A study conducted in the context of a software engineering course revealed frequent mistakes in the use of the Command pattern (A. V. Zarras 2020). To deal with these errors, the author proposed a pattern for the proper configuration of command objects. Another similar effort concerned errors in the use of the Strategy pattern and the corresponding solutions (A. Zarras 2021). Moreover, in our previous study, we identified deviations in pattern definitions and instances found on the Web (A. V. Zarras & Vassiliadis 2023). Taking a step further, in this paper, we focus on the detailed analysis of the deviations that occur in the pattern instances.

Our work is also related to the area of design pattern specification. The specification of design patterns has been extensively studied as a means to reduce the ambiguity inherent in textual pattern descriptions and to enable automated pattern verification, detection, and model transformation. Two key references that summarize the state of the art in this area are the edited volume on design pattern formalization techniques by Taibi (Taibi 2007) and the survey on design pattern specification languages by Khwaja and Alshayeb (Khwaja & Alshayeb 2016).

The state-of-the-art on design patterns specification spans a wide spectrum of approaches—ranging from UML-based notations to specification languages that rely on mathematical formalisms. For instance, the Role-Based Modeling Language (RBML) is a well-known approach to systematically specifying structural and behavioral aspects of design patterns (Kim et al. 2004). RBML provides precise conformance semantics grounded in the UML metamodel, allowing automated checking of whether a model correctly instantiates a given pattern. LePUS (Gasparis et al. 2008) relies on mathematical logic for the specification of design patterns. A design pattern specification in LePUS is a mathematical formula that may also be represented in a semantically equivalent graphical form. The Balanced Patterns Specification Language (BPSL) (Taibi & Ling 2003) is another well-known approach to formalizing design patterns. BPSL employs temporal logic of actions for the structural and behavioral specification of design patterns. In this paper, we employ BPSL to specify the GoF design patterns and illustrate the structural and behavioral deviations of the smells identified in our study.

3. Preliminaries

In this section, we provide details concerning the corpus of pattern instances that we consider in our study. In addition, we introduce the template that we use for the specification of the smells we discovered in our study.

3.1. Corpus of the study

Our study examines five well-known Web sites that offer information about the GoF patterns. Specifically, we focus on

Source Making[4], Refactoring Guru[5], Tutorials Point[6], Java T Point[7], and Wikipedia [8]. From the Web sites, we collected pattern instances that illustrate the use of the GoF patterns. The corpus of our study consists of 464 pattern instances. The data set is available online[9]

Table 1, gives additional details about the corpus. Refactoring Guru contributes the highest number of pattern instances to the corpus, followed by Source Making, Wikipedia, Tutorials Point, and Java T Point. In Refactoring Guru, Source Making, and Wikipedia we found pattern instances implemented in various programming languages, including Java, C#, C++, SmallTalk, PHP, Python, Ruby, Swift, TypeScript, Go, Javascript, and Delphi. In Refactoring Guru, we did not find instances of the Interpreter pattern, while in Tutorial Point, we did not find instances of the Visitor pattern. In Wikipedia, we found instances of all patterns except for Proxy and State.

Table 2 gives a detailed view of the different types of deviations we identified in the pattern instances of the corpus. In particular, the table provides the number of pattern instances per pattern, as well as the number of missing, incomplete, and erroneous participants found within these pattern instances. The table also reports the density of deviations for each pattern, measured as the number of deviations divided by the number of pattern instances.

In the detailed breakdown of the deviations, we observe that there are missing participants in the pattern instances of most patterns. The only exceptions are the instances of Singleton, Facade, State, and Template Method, in which we did not find any missing participants. In contrast, incomplete and erroneous participants occur in instances of fewer patterns. Specifically, we observe incomplete participants in instances of Composite, Flyweight, Iterator, Memento, and Observer. Erroneous participants occur in instances of Abstract Factory, Builder, Factory Method, Composite, Chain of Responsibility, Mediator, Memento, and Strategy.

Based on the detailed breakdown of the deviations, we conducted a thorough examination of the deviations occurring in the pattern instances of each pattern. Through this analysis, we observed that certain deviations are less critical than others when it comes to the correct use of the patterns. For instance, in some pattern instances, the missing participants are classes that simply use the core structure of the pattern. Moreover, we observed missing interface definitions in pattern instances implemented in dynamically typed languages such as Python, PHP, and Ruby. In such languages, it is not necessary to explicitly define an interface with multiple implementing classes to realize the notion of abstract coupling that underlies almost all GoF patterns. We also found incomplete participants in which the missing methods could be replaced by others that are already present, or by implementation language specific idioms. After the detailed examination of the deviations, we ended up with a catalog of smells; a smell may involve one or more different deviations, occurs in multiple pattern instances, and undermines the correctness of the pattern.

[4] sourcemaking.com
[5] refactoring.guru
[6] www.tutorialspoint.com/design_pattern
[7] www.javatpoint.com/design-patterns-in-java
[8] en.wikipedia.org/wiki/Design_Patterns
[9] https://drive.google.com/file/d/1vAn58ul7whaXM01TMFcj1er5UcCSzrYN/view?usp=sharing.

3.2. Smell description form

For the systematic specification of the smells that we identified in our study, we adopt a smell description template that consists of two parts.

The *first part* begins with a brief description of the **intent and original structure** of the pattern, specified in the GoF catalog (Gamma et al. 1994). We describe the structure of the pattern using a UML class diagram and a BPSL specification (Taibi & Ling 2003). At a glance, the BPSL specification defines a domain of participants (classes and interfaces) involved in the pattern structure, important methods and attributes, defined in participants, and permanent structural relations between participants (Table 3). BPSL supports the specification of different permanent relations between participants, including inheritance, part-of, creation, and invocation. BPSL relations are specified as predicates that should hold (e.g. Defined-in(m(), AClass) holds if m() is defined in AClass). BPSL also allows specifying temporal relations between participants and actions describing the participants' behavior.

For our study, we extend the standard set of BPSL relations with two additional permanent relations that allow us to specify classes implementing an interface and access modifiers for methods and attributes. Moreover, we define a temporal relation that facilitates the specification of associations between participants that occur when a participant is passed as a parameter or referenced by a local variable declared in a method. In the specification of associations, we deliberately omit multiplicities because they depend on the particular context in which a pattern structure is instantiated.

The *second part* of the smell specification begins with a smell evoking **name**. Then, we discuss the **symptoms of the smell**, i.e., the deviations, indicating the occurrence of the smell in pattern instances. The smell symptoms may be a combination of missing, incomplete, or erroneous participants. In this part, we also refer to the specific parts of the BPSL specification that are violated by the smelly structure. Finally, we elaborate on the **rationale** of the smell that explains why we consider the specific deviations harmful and report a concrete **example** from the corpus that illustrates the smell.

4. Creational Patterns

The GoF creational patterns deal with the object instantiation process. In this section, we discuss the smells we observed in instances of Abstract Factory, Builder, and Factory Method. We did not observe any noticeable smells in the instances of Prototype and Singleton.

4.1. Abstract Factory

Original Intent and Structure The intent of Abstract Factory, as specified in the GoF pattern catalog, is to *"provide an interface for creating families of related or dependent objects without specifying their concrete classes"* (Gamma et al. 1994)

Table 4 Abstract Factory BPSL specification.

Abstract Factory	
Exists Client, AbstractFactory, ConcreteFactory1, ConcreteFactory2, AbstractProductA, ProductA1, ProductA2, AbstractProductB, ProductB1, ProductB2 **in C;**	
Exists ProductA createProductA(), ProductB createProductB() **in M;**	
Defined-in(createProductA(), AbstractFactory)	**and**
Defined-in(createProductB(), AbstractFactory)	**and**
Defined-in(createProductA(), ConcreteFactory1)	**and**
Defined-in(createProductB(), ConcreteFactory1)	**and**
Defined-in(createProductA(), ConcreteFactory2)	**and**
Defined-in(createProductB(), ConcreteFactory2)	**and**
Implements(ConcreteFactory1, AbstractFactory)	**and**
Implements(ConcreteFactory2, AbstractFactory)	**and**
Implements(ProductA1, AbstractProductA)	**and**
Implements(ProductA2, AbstractProductA)	**and**
Implements(ProductB1, AbstractProductB)	**and**
Implements(ProductB1, AbstractProductB)	**and**
Creation(ConcreteFactory1, ProductA1)	**and**
Creation(ConcreteFactory1, ProductB1)	**and**
Creation(ConcreteFactory2, ProductA2)	**and**
Creation(ConcreteFactory2, ProductB2)	**and**
Association(Client, AbstractFactory)	**and**
Association(Client, AbstractProductA)	**and**
Association(Client, AbstractProductB)	**and**
Association(AbstractFactory, AbstractProductA)	**and**
Association(AbstractFactory, AbstractProductB)	

The original pattern structure (Figure 1(a), Table 4), includes different hierarchies of product objects. For each hierarchy, we have an interface definition (e.g. AbstractProductA) and concrete product classes (e.g., ProductA1, ProductA2) that implement the interface. A *family* of related product objects consists of objects that belong to the concrete classes of the different hierarchies (e.g., ProductA1, ProductB1). Typically, the objects of a family are related or dependent in the sense that they are designed to be used together.

The AbstractFactory interface defines operations for creating abstract product objects (e.g., createProductA(), createProductB()). AbstractFactory has different alternative implementations that correspond to families of related objects (e.g., ConcreteFactory1 creates objects that belong to ProductA1 and ProductB1, while ConcreteFactory2 creates objects that belong to ProductA2 and ProductB2). Client is parameterized with an AbstractFactory object. The Client uses the AbstractFactory object to create a family of related objects. The AbstractFactory object makes sure that the Client will not use product objects that are not meant to be used together.

Abstract Workshop

Symptoms and Smelly Structure: Abstract Factory is used to create product objects that belong to a single hierarchy of product objects (Figure 1(b)). The pattern structure comprises only one interface definition for the product objects. All objects that belong to a certain family provide the same interface.

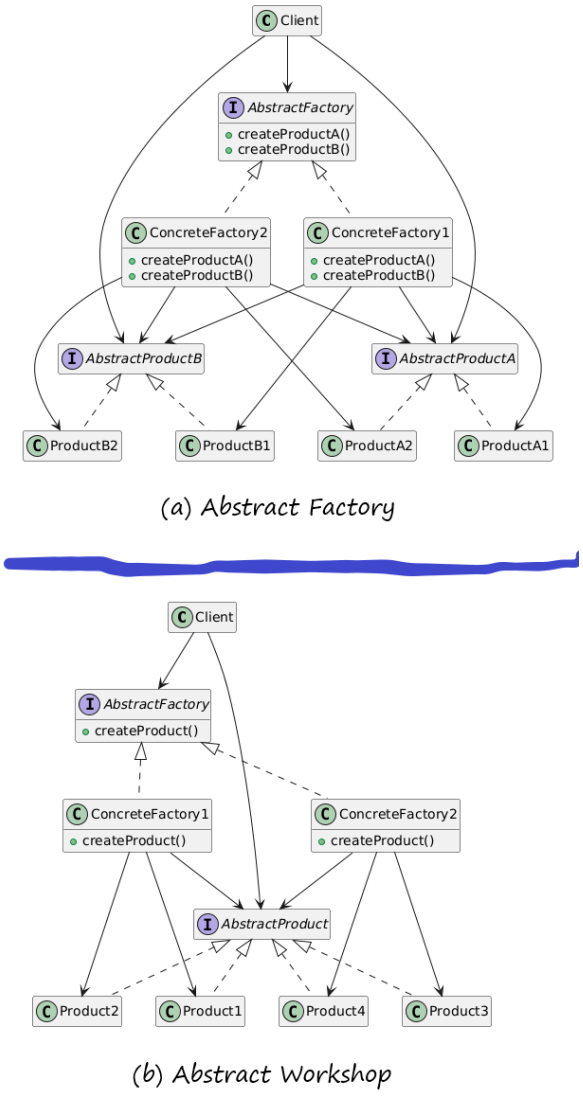

Figure 1 Abstract Factory original and smelly structures.

Abstract Factory vs. Abstract Workshop in BPSL: In the smelly structure, domain C does not include AbstractProductB, ProductB1 or ProductB2 (Table 4). All the predicates that involve them are false (e.g., Creation(ConcreteFactory1, ProductB1), Creation(ConcreteFactory2, ProductB2)).

Rationale: The goal of every pattern is to solve a specific design problem. *The problem solved by Abstract Factory is more complex than the problem solved by the smelly pattern instances.* To create objects that belong to a single hierarchy of product objects, we can use more suitable patterns such as the Factory Method, Builder, or Prototype. In fact, in the GoF book, there is a thorough discussion regarding the applicability of creational patterns at the end of the respective chapter where the authors explicitly state that *"Abstract Factory has the factory object producing objects of **several classes**".*

Example: Figure 2 gives an example of the Abstract Workshop smell, found in Source Making. The example shows a hierarchy of concrete classes that implement the Shape interface.

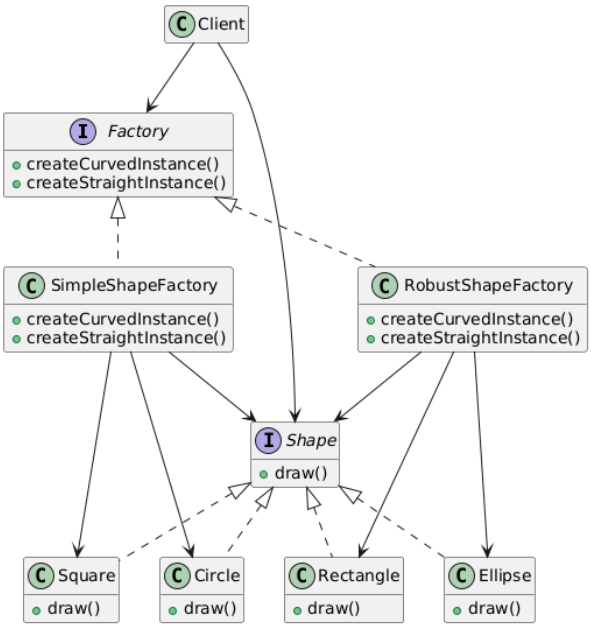

Figure 2 Abstract Workshop example - Source Making.

The Factory interface provides two methods for creating Shape objects. The Factory interface has two implementations: the SimpleShapeFactory creates Square and Circle objects, while the RobustShapeFactory creates Ellipse and Rectangle objects. In this example, we can easily constrain the combinations of Shape objects that can be created and used together, without the Abstract Factory pattern. A simple way to do this is to use the prototype pattern (Gamma et al. 1994). Specifically, we can use two hash maps for creating Shape objects, one containing two prototypical Shape objects that belong to Square and Circle, and another containing two prototypical Shape objects that belong to Rectangle and Ellipse

4.2. Builder

Original Intent and Structure The Builder pattern allows to *"separate the construction of a complex object from its representation so that the same construction process can create different representations"* (Gamma et al. 1994). In the original pattern structure (Figure 3(a), Table 5), the Builder interface defines methods for creating parts of a complex Product object. ConcreteBuilder is a concrete implementation of the Builder interface that creates a specific representation of Product objects. We may have multiple concrete implementations of Builder for respective Product object representations. The Director encapsulates the construction process of complex Product objects and constructs them using a Builder object.

Lazy Builder

Symptoms and Smelly Structure: ConcreteBuilder does not build the parts of the Product objects. Instead, the Director class creates the parts during the construction process. The Director uses the Builder object only to set the parts of the Product object using simple setter methods (Figure 3(b)).

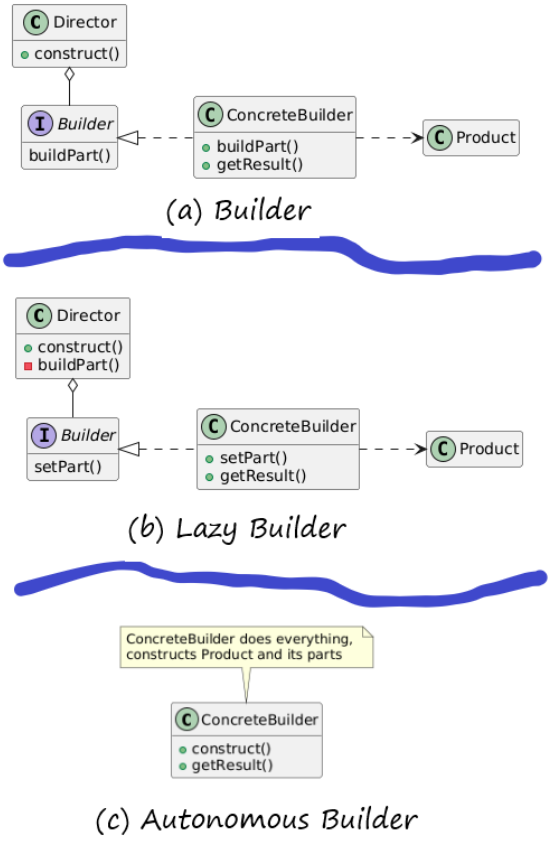

Figure 3 Builder original and smelly structures.

Table 5 Builder BPSL specification.

Builder	
Exists Director, Builder, ConcreteBuilder, Product **in C;** **Exists** construct(), buildPart(), getResult() **in M;**	
Defined-in(construct(), Director)	**and**
Defined-in(buildPart(), Builder)	**and**
Defined-in(buildPart(), ConcreteBuilder)	**and**
Defined-in(getResult(), ConcreteBuilder)	**and**
Implements(ConcreteBuilder, Builder)	**and**
Creation(ConcreteBuilder, Product)	**and**
Reference-to-one(Director, Builder)	**and**
Invocation(construct(), buildPart())	**and**
Invocation(construct(), buildPart())	

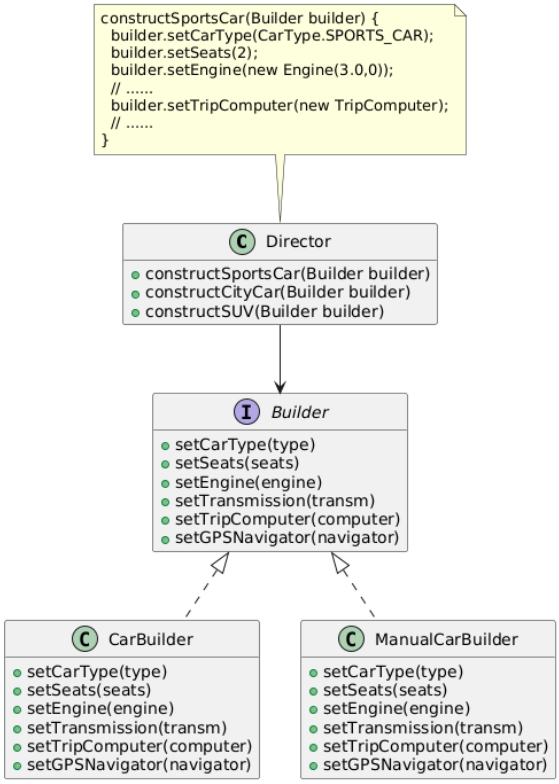

Figure 4 Lazy Builder example - Refactoring Guru.

Builder vs. Lazy Builder in BPSL: In the smelly structure Builder and ConcreteBuilder do not define a buildPart() method (in Table 5, Defined-in(buildPart(), Builder) and Defined-in(buildPart(), ConcreteBuilder) are false). Instead, the method is defined in Director.

Rationale: In the smelly pattern instances, the pattern intent is compromised because *the construction process of the Product object is not separated from the creation of its parts*.

Example: Figure 4, illustrates an example of the Lazy Builder smell found in Refactoring Guru. In this example, the Director realizes three construction processes for Car objects. The Director is coupled with the internal representation of the Car objects being built. In particular, each construction process (e.g., constructSportsCar()) creates the objects that make up a car (e.g., Engine object, TripComputer object) and gives these objects to Builder. The Builder sets the parts of the Car object, instead of creating them.

Autonomous Builder

Symptoms and Smelly Structure: ConcreteBuilder is responsible for the construction of the Product object and the creation of its parts. The pattern structure does not include a Director class (Figure 3(c)).

Builder vs. Autonomous Builder in BPSL: Director and Builder are missing from domain C, and all related predicates are false (e.g., in Table 5, Reference-to-one(Director, Builder) is false).

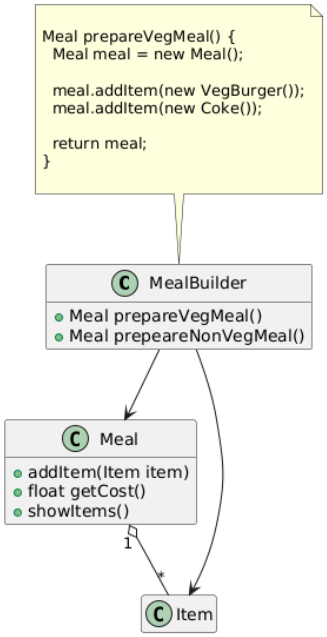

Figure 5 Autonomous Builder example - Tutorials Point.

Rationale: The Autonomous Builder jeopardizes the pattern intent because *the construction process of the Product object is not separated from the creation of its parts*.

Example: Figure 5, gives an example of the Autonomous Builder smell, found in Tutorials Point. In this example, the MealBuilder is responsible for creating and setting the objects that make up a Meal object.

4.3. Factory Method

Original Intent and Structure The key idea in the Factory Method pattern is to *"define an interface for creating an object, but let subclasses decide which class to instantiate"* (Gamma et al. 1994). In the original pattern structure (Figure 6, Table 6), Creator defines a factoryMethod() that returns a Product object. ConcreteCreator overrides factoryMethod(), defined in Creator, to create and return a ConcreteProduct object. According to this structure, it is possible to add further Product implementations, along with respective Creator subclasses that override the factoryMethod(), to create and return objects of these implementations.

Gamma et al. discuss **two major variants**. In the first variant, Creator is an abstract class that defines an abstract factoryMethod(). The Creator lets its subclasses decide the kind of Product objects that will be created, by implementing the abstract factoryMethod(). In the second variant, Creator is a concrete class that provides a default factoryMethod() implementation. In this variant, the Creator subclasses can change the kind of Product objects created by overriding the default factoryMethod() implementation. Gamma et al. further discuss a **third variant** of the pattern in which the Creator provides a parameterized factoryMethod() that creates different kinds of Product objects. The method decides which product object to construct based on a given parameter. In this variant, subclasses

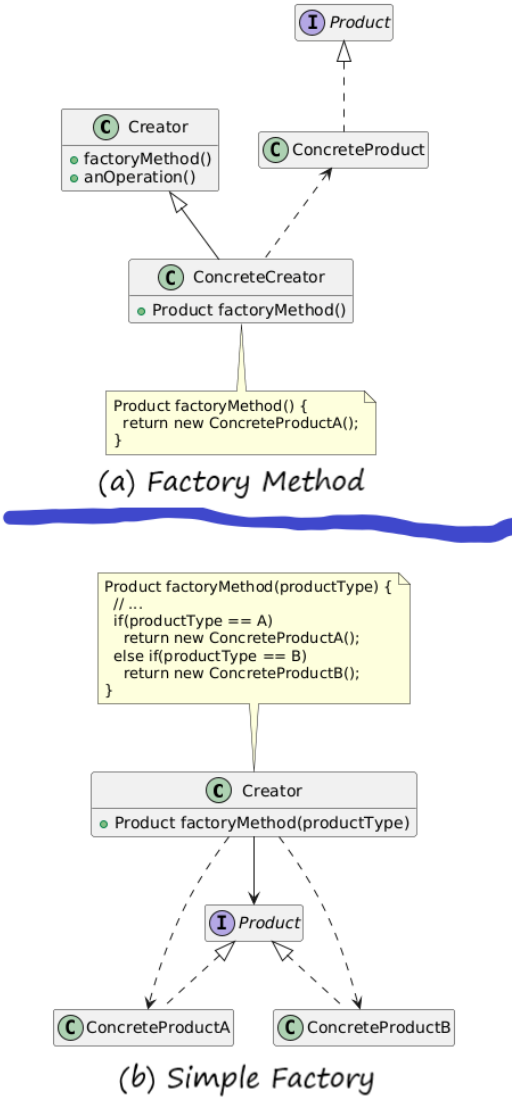

Figure 6 Factory Method original and smelly structure.

can override the creation method to change the kinds of Product objects that will be created or to add new kinds of Product objects. In the latter case, the factoryMethod() in the subclasses should invoke the factoryMethod() of the Creator.

Simple Factory

Symptoms and Smelly Structure: The Creator class defines a parameterized factoryMethod() that creates different kinds of Product objects (Figure 6(b)). The Creator class has no subclasses.

Factory Method vs. Simple Factory in BPSL: In the smelly structure, ConcreteCreator is missing from domain C (Table 6). Moreover, the factoryMethod() prototype is parameterized. Hence, it does not match the factoryMethod() that should be included in M. All related predicates are not valid (e.g., Inheritance(ConcreteCreator, Creator) is false).

Rationale: The smelly pattern instances *illustrate the third variant of the Factory Method pattern, instead of focusing on the key concept of deferring Product object creation to subclasses*. The third variant of the Factory Method pattern is also known as Factory (Cooper 1998) or Simple Factory[10] and aims to encapsulate the creation logic in one place, rather than distributing it to subclasses.

Table 6 Factory Method BPSL specification.

Factory Method	
Exists Creator, ConcreteCreator, Product, ConcreteProduct **in C;** **Exists** Product factoryMethod(), anOperation() **in M;**	
Defined-in(factoryMethod(), Creator) **Defined-in**(anOperation(), Creator) **Defined-in**(factoryMethod(), ConcreteCreator) **Inheritance**(ConcreteCreator, Creator) **Creation**(factoryMethod(), ConcreteProduct) **Creation**(ConcreteCreator, ConcreteProduct)	**and** **and** **and** **and** **and**

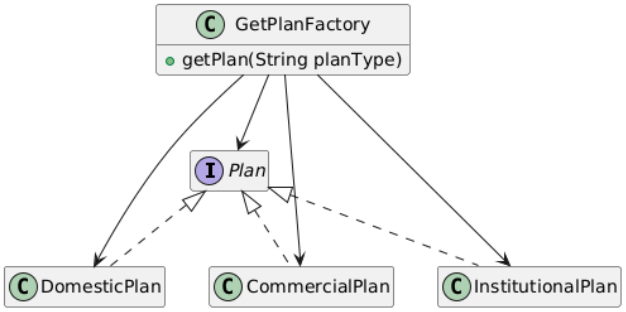

Figure 7 Simple Factory example - Java T Point.

Example: Figure 7, gives an example of the Simple Factory smell found at Java T Point. In the example, GetPlanFactory defines the getPlan() method that creates different kinds of Plan objects, based on a given parameter.

5. Structural Patterns

The GoF structural patterns deal with the composition of objects into more complex structures. In our study, we identified smells in instances of Composite, Flyweight, and Proxy. The instances of the Adapter, Decorator, Bridge, and Facade are smell-free.

5.1. Composite

Original Intent and Structure The Composite pattern *"lets clients treat individual objects and compositions of objects uniformly"* (Gamma et al. 1994).

In the original pattern structure, the Component class represents primitive and composite objects (Figure 8(a), Table 7). The Leaf and Composite classes are derived from Component. Leaf represents primitive objects, while Composite represents composite objects that consist of other primitive and/or composite objects. The goal of the pattern is to make the clients unaware of the specific types of primitive and composite objects they are

[10] refactoring.guru/design-patterns/factory-comparison

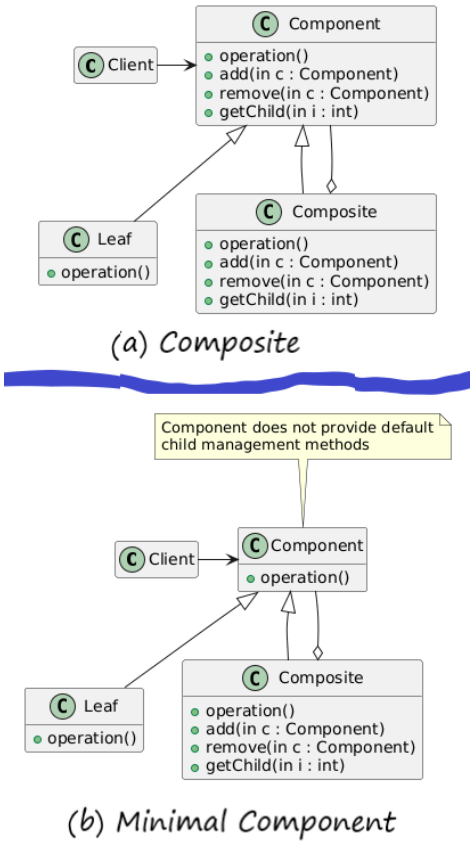

Figure 8 Composite original and smelly structure.

Table 7 Composite BPSL specification.

Composite	
Exists Client, Component, Leaf, Composite **in C;** **Exists** operation(), add(c: Component), remove(c: Component), getChild(int child) **in M;**	
Defined-in(operation(), Component)	**and**
Defined-in(add(), Component)	**and**
Defined-in(remove(), Component)	**and**
Defined-in(getChild(), Component)	**and**
Defined-in(operation(), Leaf)	**and**
Defined-in(operation(), Composite)	**and**
Defined-in(add(), Composite)	**and**
Defined-in(remove(), Composite)	**and**
Defined-in(getChild(), Composite)	**and**
Inheritance(Leaf, Component)	**and**
Inheritance(Composite, Component)	**and**
Reference-to-one(Composite, Component)	**and**
Association(Client, Component)	

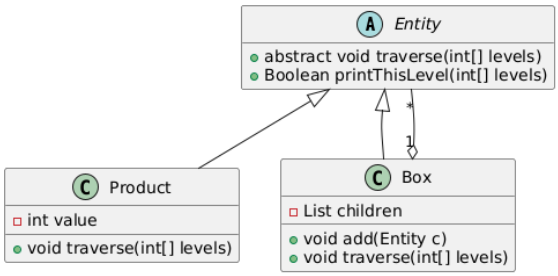

Figure 9 Minimal Component example - Source Making.

using. To this end, the Component class provides default methods for primitive and composite objects. This set of methods includes default child management operations for accessing the constituents of Composite objects. Child management methods may not be meaningful for Leaf objects. However, Gamma et al. point out that the definition of child management methods at the root of the class hierarchy is a deliberate design choice that favors uniform object treatment over type safety. The default methods, defined in Component, are overridden by the corresponding methods in Leaf and Composite, which define the specific behavior of Leaf and Composite objects.

Minimal Component

Symptoms and Smelly Structure: The default set of methods, defined in the Component class, is not maximal. In particular, the Component class does not define the default child management methods (Figure 8(b)).

In BPSL: In the smelly structure the predicates that concern the definition of the child management operations in the Component class (Defined-in(add(), Component), Defined-in(remove(), Component), and Defined-in(getChild(), Component)) are false (Table 7).

Rationale: In the smelly pattern structure, *type safety comes before uniform object treatment*. Consequently, the smelly pattern instances do not fully comply with the Composite pattern intent.

Example: Figure 9 illustrates a specific example found in Source Making. In this example, Element is a typical abstract class that defines an abstract traverse() method. Element has two concrete implementation classes, Product and Box. The Box objects are composite, consisting of other Product or Box objects. The Element class defines a minimal set of public methods, which does not include child management methods.

5.2. Flyweight

Original Intent and Structure The idea behind the Flyweight pattern is to *"use sharing to support large numbers of fine-grained objects efficiently"* (Gamma et al. 1994). In other words, Flyweight facilitates sharing a common state between objects, instead of wasting memory by storing the common state multiple times, one for each object. Two key concepts in Flyweight are the *intrinsic* and *extrinsic* state. The former is

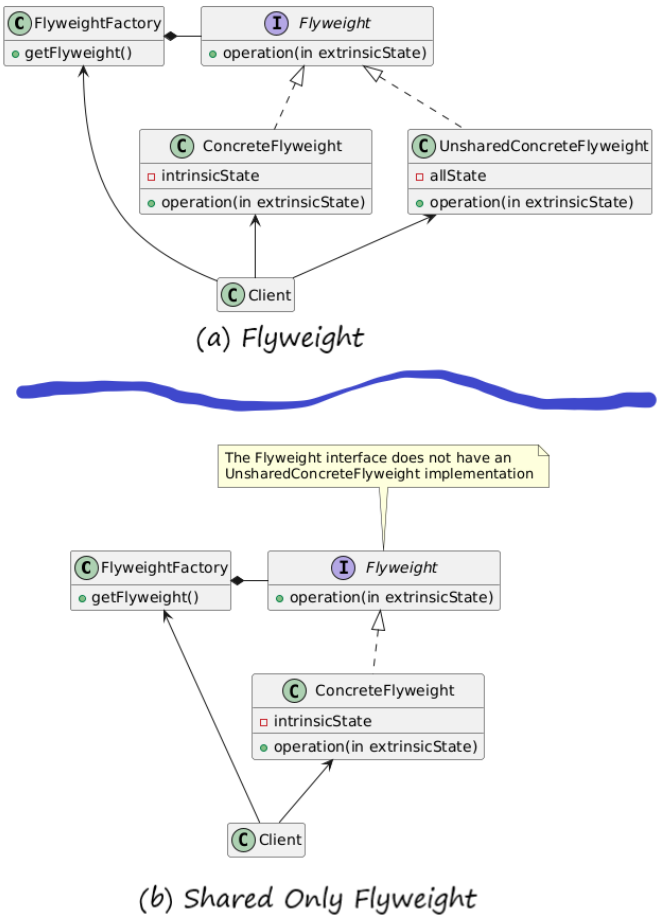

Figure 10 Flyweight original and smelly structure.

a state that can be shared between multiple objects, while the latter is a context-dependent state that cannot be shared.

The original pattern structure comprises the Flyweight interface that allows one to set the extrinsic state of the Flyweight objects (Figure 10(a), Table 8). ConcreteFlyweight is a concrete implementation of Flyweight that defines the intrinsic state of shareable objects. The original pattern structure also includes the UnsharedConcreteFlyweight implementation of Flyweight to highlight that certain Flyweight objects may not be shareable. FlyweightFactory enables the sharing of Flyweight objects. Upon a request, FlyweightFactory creates or provides a reference to an existing Flyweight object. The Client uses FlyweightFactory to obtain references to Flyweight objects. The Client is also responsible for setting the extrinsic state of the Flyweight objects.

Shared Only Flyweight

Symptoms and Smelly Structure: The smelly structure does not include UnsharedConcreteFlyweight (Figure 10(b)).

Flyweight vs. Shared Only Flyweight in BPSL: According to the smelly structure, UnsharedConcreteFlyweight is omitted in domain C (Table 8). All related predicates are invalid (e.g., Association(Client, UnsharedConcreteFlyweight), Creation(FlyweightFactory, UnsharedConcreteFlyweight) are invalid).

Table 8 Flyweight BPSL specification.

Flyweight	
Exists Client, FlyweightFactory, Flyweight, ConcreteFlyweight, UnsharedConcreteFlyweight **in C;** **Exists** operation(extrinsicState), Flyweight getFlyweight() **in M;** **Exists** intrinsicState, allState **in A;**	
Defined-in(operation(), Flyweight**)**	**and**
Defined-in(operation(), ConcreteFlyweight**)**	**and**
Defined-in(operation(), UnsharedConcreteFlyweight**)**	**and**
Defined-in(intrinsicState, ConcreteFlyweight**)**	**and**
Defined-in(allState, UnsharedConcreteFlyweight**)**	**and**
Implements(ConcreteFlyweight, Flyweight**)**	**and**
Implements(UnsharedConcreteFlyweight, Flyweight**)**	**and**
Creation(FlyweightFactory, ConcreteFlyweight**)**	**and**
Creation(FlyweightFactory,UnsharedConcreteFlyweight**)**	**and**
Reference-to-one(FlyweightFactory, Flyweight**)**	**and**
Association(Client, Flyweight**)**	**and**
Association(Client, FlyweightFactory**)**	**and**
Association(Client, ConcreteFlyweight**)**	**and**
Association(Client, UnsharedConcreteFlyweight**)**	

Rationale: According to Gamma et al., the Flyweight pattern is supposed to enable sharing, not to enforce it. Without UnsharedConcreteFlyweight, the option of not sharing a Flyweight object is unavailable.

Example: A Tutorials Point example that illustrates the Shared Only Flyweight smell is given in Figure 11. ShapeFactory is responsible for the creation and sharing of Shape objects. Circle is a concrete implementation of the Shape interface. The possibility of not sharing a Shape object is not available.

5.3. Proxy

Original Intent and Structure The intent of the Proxy pattern is to *"provide a surrogate or placeholder for another object to control access to it"* (Gamma et al. 1994). Gamma et al. refer to different kinds of proxies that serve various purposes, such as the remote proxy, the virtual proxy, the protection proxy, etc. The original structure of the pattern includes the Subject interface that is common for real objects and proxy objects (Figure 12(a), Table 9). The RealSubject class represents real objects, while the Proxy class represents proxy objects. Proxy has a reference to an encapsulated RealSubject object. This reference allows a Proxy object to control access to the encapsulated RealSubject object. The Client class has a reference to a Subject object that enables the Client objects to access either a RealSubject object or a Proxy object that encapsulates a RealSubject object.

Proxy Without Subject

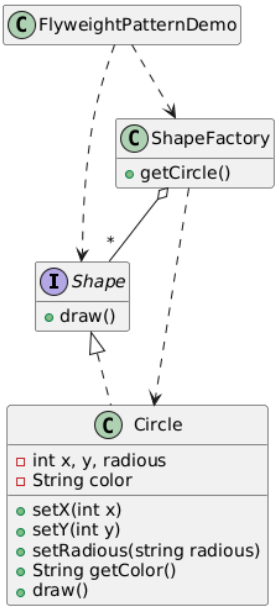

Figure 11 Shared Only Flyweight example - Tutorials Point.

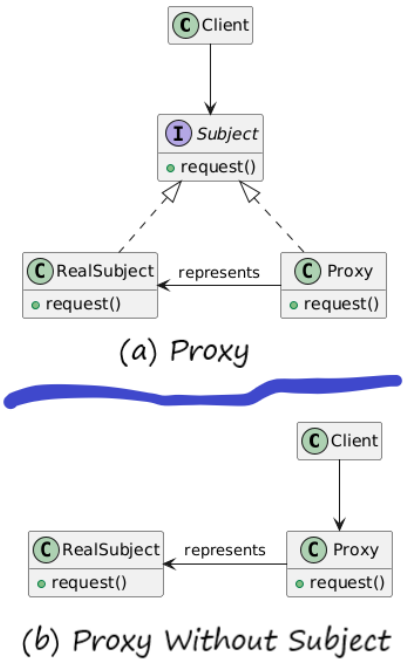

Figure 12 Proxy original and smelly structure.

Table 9 Proxy BPSL specification.

Proxy	
Exists Client, Subject, RealSubject, Proxy **in C;**	
Exists request() **in M;**	
Defined-in(request(), Subject**)**	**and**
Defined-in(request(), RealSubject**)**	**and**
Defined-in(request(), Proxy**)**	**and**
Implements(RealSubject, Subject**)**	**and**
Implements(Proxy, Subject**)**	**and**
Association(Proxy, RealSubject**)**	**and**
Association(Client, Subject**)**	

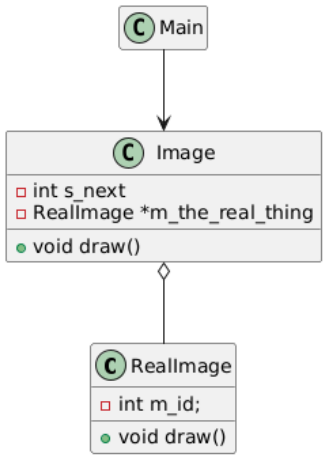

Figure 13 Proxy Without Subject example - Source Making.

Symptoms and Smelly Structure: The Client class directly refers to the Proxy object. The pattern structure does not include the definition of a Subject interface (Figure 12(b)).

Proxy vs. Proxy Without Subject in BPSL: In the smelly structure, Subject is missing from domain C. The predicates that involve the missing Subject are invalid (e.g., in Table 9, Implements(Proxy, Subject), Implements(RealSubject, Subject) are false).

Rationale: According to the pattern intent, the Proxy object is a surrogate that can be used anywhere a RealSubject object is expected, without code changes. However, this is impossible if the Client class directly refers to a RealSubject object or a Proxy object. Instead, the Client class should depend on a higher-level abstraction.

Example: Figure 13 gives an example of Proxy Without Subject, found in Source Making. The RealImage class represents real objects, while the Image class represents proxy objects. The Image class has a reference to an encapsulated RealImage object. Two classes do not implement a common interface. The Main class creates and uses a Proxy object.

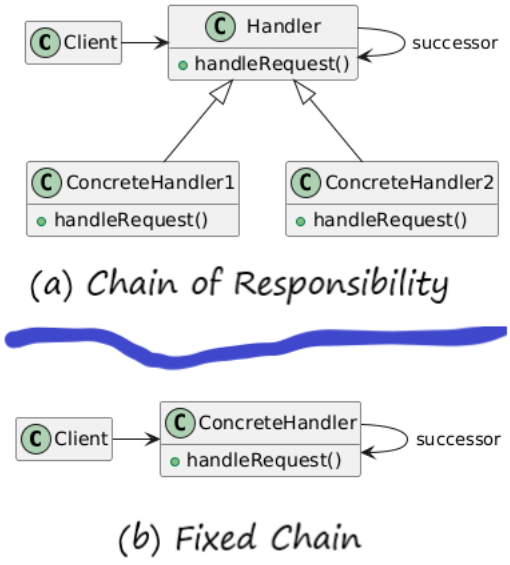

Figure 14 Chain of Responsibility original and smelly structure.

6. Behavioral Patterns

The behavioral patterns focus on issues related to assigning responsibilities to classes and the communication between objects. In the corpus, we identified smells in instances of Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Strategy, and State. On the other hand, we did not find any smells in the instances of Observer, Template Method, and Visitor.

6.1. Chain of Responsibility

Original Intent and Structure The intent of the Chain of Responsibility pattern is to *"avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request"* (Gamma et al. 1994). To do this, the sender passes the request along a chain of handlers that provide the same interface. According to the original pattern structure, Handler defines an interface for handling requests and maintains a Handler reference that facilitates the setup of the chain of Handler objects (Figure 14(a), Table 10). ConcreteHandler1, ConcreteHandler2 are concrete Handler implementations. Generally, a Handler object handles a request and/or passes it to its successor. The Client class is responsible for initiating a request to the chain of Handler objects.

Fixed Chain

Symptoms and Smelly Structure: The classes of the handling objects are strongly coupled (Figure 14(b), Table 10). The smelly pattern structure does not include a common Handler interface.

Chain of Responcibility vs. Fixed Chain: According to the smelly structure, Handler is missing from domain

Table 10 Chain of Responsibility BPSL specification.

Chain of Responsibility	
Exists Client, Handler, ConcreteHandler1, ConcreteHandler2 **in C;** **Exists** handleRequest() **in M;**	
Defined-in(handleRequest(), Handler**)** **Defined-in(**handleRequest(), ConcreteHandler1**)** **Defined-in(**handleRequest(), ConcreteHandler2**)** **Inheritance(**ConcreteHandler1, Handler**)** **Inheritance(**ConcreteHandler2, Handler**)** **Association(**Handler, Handler**)** **Association(**Client, Handler**)**	**and** **and** **and** **and** **and** **and**

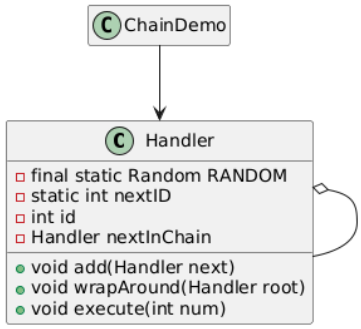

Figure 15 Fixed Chain example - Source Making.

C, and the respective predicates are false (e.g., in Table 10 Inheritance(ConcreteHandler1, Handler), Defined-in(handleRequest(), Handler) do not hold).

Rationale: Decoupling the senders from the receivers is a key idea of the pattern. This idea is compromised in the smelly pattern structure because the handling objects depend on the concrete classes of their successor objects. According to the smelly pattern structure, it is only possible to construct fixed chains that cannot be easily changed.

Example: An example of the Fixed Chain smell found in Source Making is given in Figure 15. Handler is a concrete class that has a self-reference that allows the construction of chains of Handler objects. Adding other types of objects to the chain is not possible.

6.2. Command

Original Intent and Structure The purpose of the Command pattern is to *"encapsulate a request as an object, thereby allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations"*. The structure of the pattern comprises the Command interface that defines the interface for executing an operation. ConcreteCommand implements the Command interface using a Receiver object, Invoker uses a Command object to perform a request. Client creates a ConcreteCommand object, sets its receiver and executes commands through Invoker (Figure 16(a), Table 11).

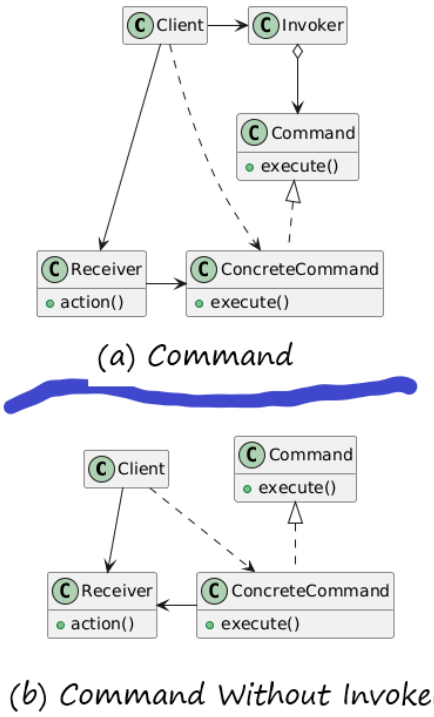

Figure 16 Command original and smelly structure.

Command Without Invoker:

Symptoms and Smelly Structure: The smelly pattern structure does not include an Invoker class. Often, the missing Invoker co-occurs with a missing Command interface (Figure 16(b)).

Command vs Command Without Invoker in BPSL: In the smelly structure, Invoker is not included in domain C. The predicates that involve the missing participant are invalid (e.g., in Table 11, Association(Client, Invoker), Reference-to-one(Invoker, Command) are false).

Table 11 Command BPSL specification.

Command	
Exists Client, Invoker, Command, ConcreteCommand, Receiver **in C;** **Exists** execute(), action() **in M;**	
Defined-in(execute(), Command**)** **Defined-in(**execute(), ConcreteCommand**)** **Defined-in(**action(), Receiver**)** **Implements(**ConcreteCommand, Command**)** **Creation(**Client, ConcreteCommand**)** **Reference-to-one(**Invoker, Command**)** **Association(**Client, Invoker**)** **Association(**Client, Receiver**)**	**and** **and** **and** **and** **and** **and** **and**

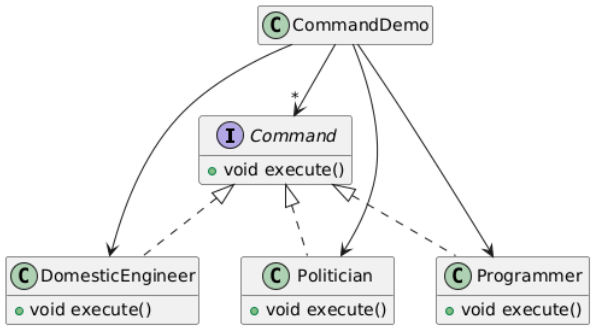

Figure 17 Command Without Invoker example - Source Making.

Rationale: The purpose of the Command pattern is to enable the parameterization of the Invoker with different types of Command objects. Hence, without the Invoker the use of the pattern is meaningless.

Example: Figure 17 gives an example of the smell, found in Source Making. In the example, we observe a hierarchy consisting of 3 different concrete implementations of the Command interface, representing 3 different commands. CommandDemo creates Command objects. The example does not include any other class that could play the role of the Invoker, which is supposed to be parameterized with different kinds of Command objects.

6.3. Interpreter

Original Intent and Structure The Interpreter pattern concerns the design of an interpreter for a given language. Specifically, Gamma et al. state that the pattern intends to *"define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language"* (Gamma et al. 1994).

The backbone of the pattern structure is the representation of the grammar of the language (Figure 18(a), Table 12). The language representation consists of the AbstractExpression class that defines an abstract interpret() method. AbstractExpression has two concrete implementations, namely TerminalExpression and NonTerminalExpression, representing, respectively, the terminal symbols and the non-terminal symbols of the language. The pattern structure further includes the Context class, which holds global information for the interpreter, and the Client class, which constructs and sets up the expressions and context.

Hard-coded Grammar

Symptoms and Smelly Structure: The smelly pattern structure does not explicitly define a grammar representation for the interpreted language, that is, the grammar representation classes (AbstractExpression, TerminalExpression, and NonTerminalExpression) are missing (Figure 18(b)).

Interpreter vs. Hard-coded Grammer in BPSL: In the smelly structure, except for the Client, all other participants are missing from domain C (Table 12). Due to the missing participants, all

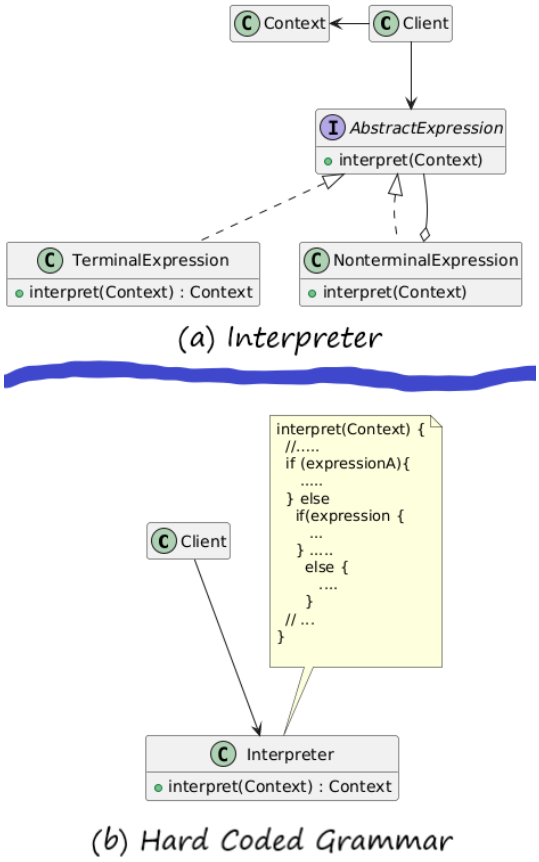

Figure 18 Interpreter original and smelly structure.

Table 12 Interpreter BPSL specification.

Interpreter	
Exists Client, Context, AbstractExpression, TerminalExpression, NonTerminalExpression **in C;**	
Exists interpret(Context) **in M;**	
Defined-in(interpret(),AbstractExpression**)**	**and**
Defined-in(interpret(),TerminalExpression**)**	**and**
Defined-in(interpretContext(),NonTerminalExpression**)**	**and**
Implements(TerminalExpression, AbstractExpression**)**	**and**
Implements(NonTerminalExpression, AbstractExpression**)**	**and**
Reference-to-one(NonTerminalExpression, AbstractExpression**)**	**and**
Association(Client, AbstractExpression**)**	**and**
Association(Client, Context**)**	

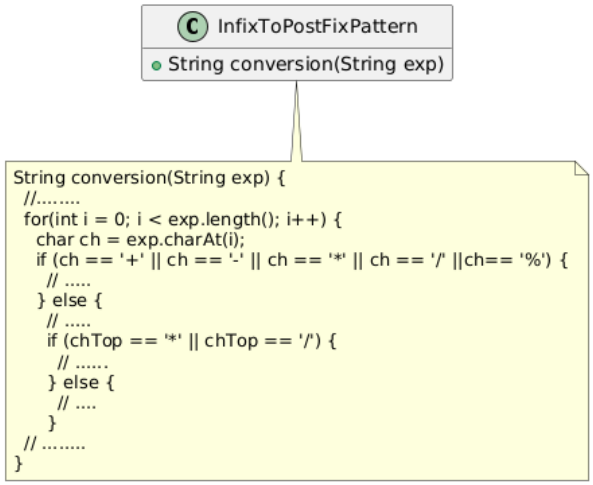

Figure 19 Hard-coded Grammar example - Java T Point.

the predicates are invalid (e.g., Defined-in(interpret(), AbstractExpression), Association(Client, AbstractExpression) are not valid).

Rationale: In the smelly pattern instances, the grammar representation is hard-coded in the interpreter, which parses and interprets the language expressions. Such designs and implementations do not conform to the intent of the Interpreter pattern.

Example: Figure 19, illustrates an occurrence of the Hard-coded Grammar smell in a pattern instance found in Java T Point. In this example, InfixToPostFixPattern is the interpreter that transforms infix mathematical expressions into corresponding postfix mathematical expressions. The convert() method is responsible for the transformation. The grammar for the interpreted infix expressions is hard-coded in the convert() method as a long chain of if-else statements.

6.4. Iterator

Original Intent and Structure The purpose of the Iterator pattern is to *"provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"* (Gamma et al. 1994). The pattern structure includes the Iterator interface that defines methods for accessing the elements of the aggregate object, the ConcreteIterator that implements the Iterator interface, the Aggregate interface that provides means to create an Iterator object, and the ConcreteAggregate that implements the Aggregate interface to return a ConcreteIterator object (Figure 20(a), Table 13). The Client class has a reference to an Aggregate object. Using this reference, the Client can obtain a reference to an Iterator object, which it will use to traverse the elements that make up the Aggregate object.

Monomorphic Iteration:

Symptoms and Smelly Structure: In the smelly structure, the Aggregate and the Iterator interfaces are missing (Figure 20(b)). The Client class directly refers to a ConcreteAggregate object. Using this reference, the Client can obtain a reference to a

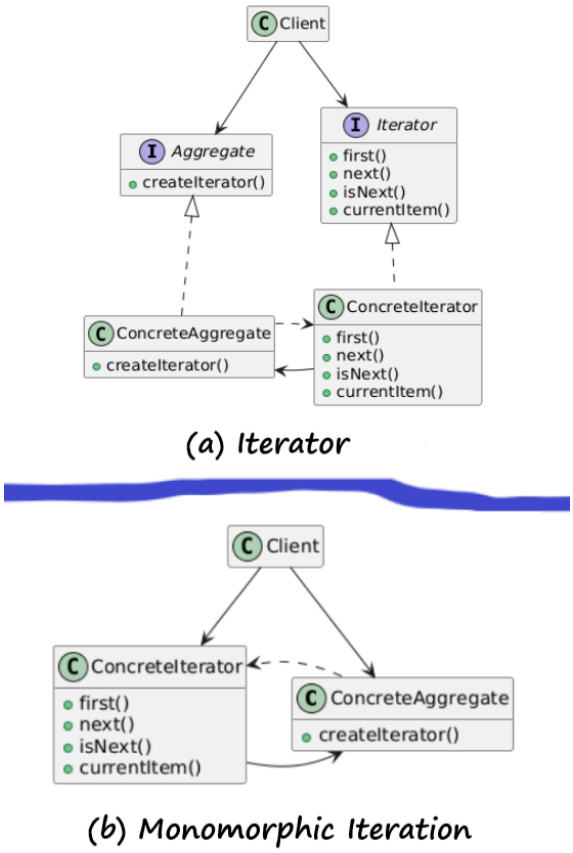

Figure 20 Iterator original and smelly structure.

Table 13 Iterator BPSL specification.

Iterator	
Exists Client, Iterator, ConcreteIterator, Aggregate, ConcreteAggregate **in C;** **Exists** first(), next(), isNext(), currentItem(), createIterator() **in M;**	
Defined-in(first(), Iterator**)**	**and**
Defined-in(next(), Iterator**)**	**and**
Defined-in(isNext(), Iterator**)**	**and**
Defined-in(currentItem(), Iterator**)**	**and**
Defined-in(first(), ConcreteIterator**)**	**and**
Defined-in(next(),ConcreteIterator**)**	**and**
Defined-in(isNext(),ConcreteIterator**)**	**and**
Defined-in(currentItem(),ConcreteIterator**)**	**and**
Defined-in(createIterator(), Aggregate**)**	**and**
Defined-in(createIterator(),ConcreteAggregate**)**	**and**
Inheritance(ConcreteIterator, Iterator**)**	**and**
Inheritance(ConcreteAggregate, Aggregate**)**	**and**
Creation(ConcreteAggregate,ConcreteIterator**)**	**and**
Association(ConcreteIterator,ConcreteAggregate**)**	

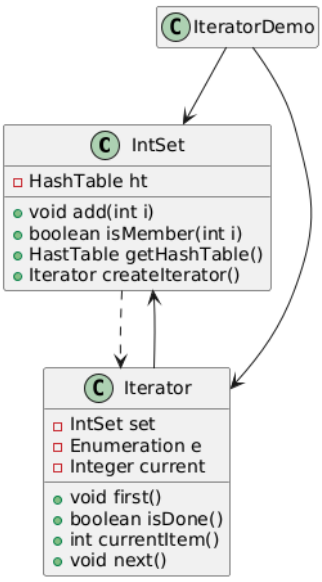

Figure 21 Monomorphic Iteration example - Source Making.

Table 14 Mediator BPSL specification.

Mediator	
Exists Mediator, ConcreteMediator, Colleague, Colleague1, Colleague2 **in C;**	
Implements(ConcreteMediator, Mediator**)**	**and**
Inheritance(Colleague1, Colleague**)**	**and**
Inheritance(Colleague2, Colleague**)**	**and**
Association(Colleague, Mediator**)**	**and**
Association(ConcreteMediator, Colleague1**)**	**and**
Association(ConcreteMediator, Colleague2**)**	

ConcreteIterator object which it will use to traverse the elements that make up the ConcreteAggregate object.

Iterator vs. Monomorphic Iteration in BPSL: The Iterator and Aggregate interfaces are not included in domain C (Table 13). The predicates that concern the missing participants are not valid (e.g., Inheritance(ConcreteIterator, Iterator), Inheritance(ConcreteAggregate, Aggregate) do not hold).

Rationale: The smelly pattern instances do not illustrate one of the prominent features of the Iterator pattern, the so-called *polymorphic iteration*, which allows clients to traverse various types of aggregate objects with respective iterators that realize different traversal algorithms. The concept of polymorphic iteration is possible thanks to the Aggregate and Iterator interfaces.

Example: Figure 21 gives a typical example of the Monomorphic Iteration smell, observed in Source Making. The example comprises only concrete classes. The IntSet class manages a set of integers, while the Iterator class allows traversing the elements of an IntSet object sequentially, without exposing its internal representation.

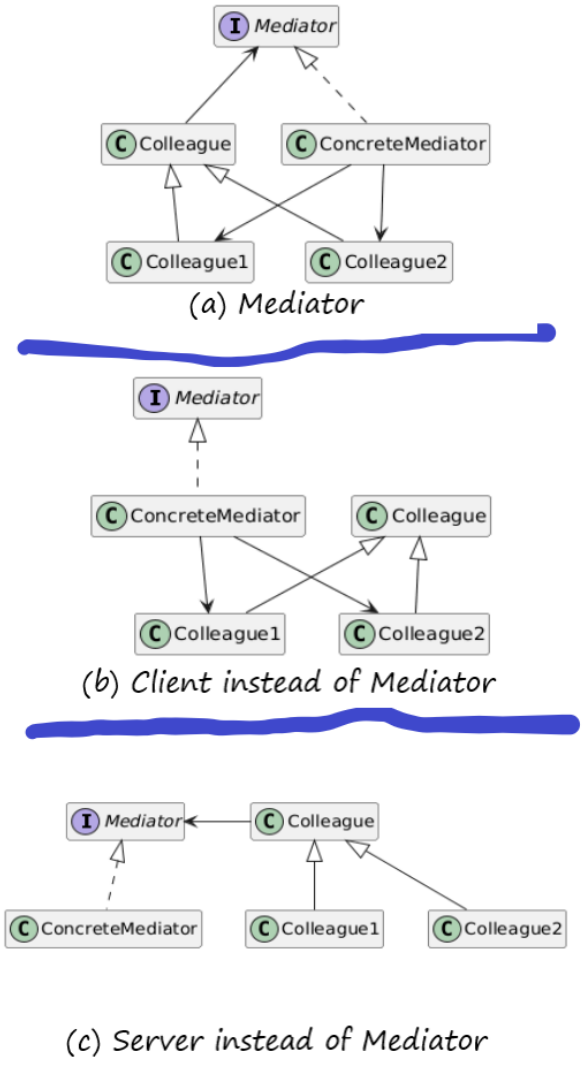

Figure 22 Mediator original and smelly structure.

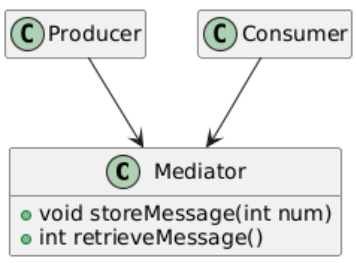

Figure 23 Mediator example - Source Making.

6.5. Mediator

Original Intent and Structure According to Gamma et al., the Mediator pattern allows to *"define an object that encapsulates how a set of objects interact"* (Gamma et al. 1994). In more detail, the pattern lets a set of objects communicate indirectly through a mediator object that encapsulates all of their interactions. The communication between the objects is done independently of these objects. In the pattern structure, the Mediator interface defines methods for interacting with Colleague objects (Figure 22(a), Table 14). The ConcreteMediator implementation of the Mediator interface coordinates the indirect interaction between the Colleague objects. The Colleague interface may have different implementation classes (Colleague1, Colleague2) representing various types of Colleague objects.

Mediator that Doesn't Mediate

Symptoms and Smelly Structure: The ConcreteMediator acts as a client that invokes methods on the Colleague objects, but the Colleague objects do not call the ConcreteMediator object (Figure 22(b)). Alternatively, the ConcreteMediator acts as a server, while the Colleague objects act as clients (Figure 22(c)).

Mediator vs. Mediator that Doesn't Mediate in BPSL: When ConcreteMediator acts as a server, Association(ConcreteMediator, Colleague1) and Association(ConcreteMediator, Colleague2) are false. Alternatively, when ConcreteMediator acts as a client, Association(Colleague, Mediator) is false.

Rationale: The ConcreteMediator does not realize an indirect coordination protocol between the Colleague objects. Hence, the smelly pattern instances compromise the intent of the pattern.

Example: Figure 23 gives an example of the Mediator that Doesn't Mediate, found in Source Making. In this example, the Producer objects invoke a Mediator object to store messages, and the Consumer objects call the Mediator object to retrieve messages. The Mediator object is a passive server instead of a coordinator that *routes* messages between Producer and Consumer objects.

6.6. Memento

Original Intent and Structure *"Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later"* (Gamma et al. 1994). According to the pattern structure, Memento is responsible for storing the state of the Originator (Figure 24(a), Table 15). Memento is also responsible for protecting the state of the Originator against access by other objects. The Originator creates a Memento object that stores its current state. The Caretaker is responsible for keeping the Memento object. The Caretaker never modifies or accesses the contents of a Memento object. These contents should only be available to the Originator.

Exposed Memento

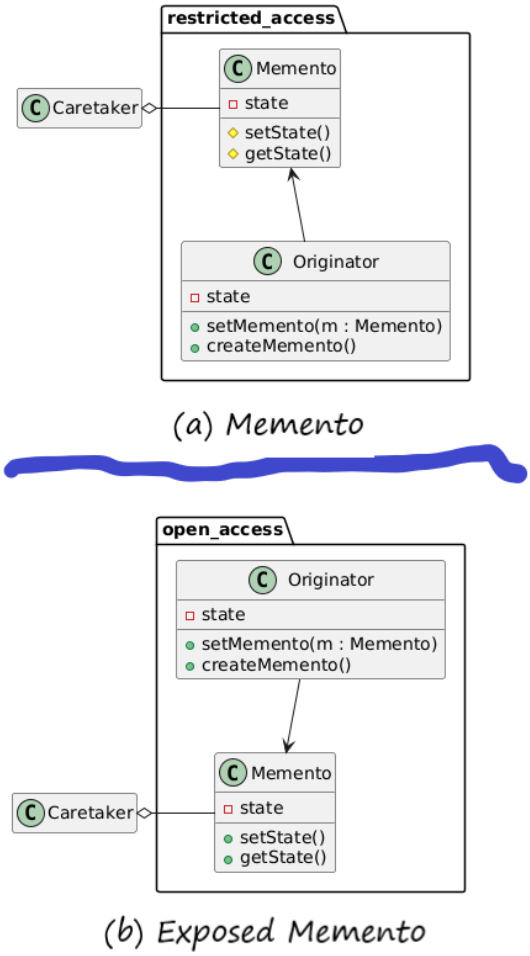

Figure 24 Memento original and smelly structure.

Table 15 Memento BPSL specification.

Memento	
Exists Memento, Originator, Caretaker **in C;**	
Exists getState(), setState(state), setMemento(m: Memento), Memento createMemento(), operation() **in M;**	
Exists state **in A;**	
Defined-in(getState(), Memento**)**	**and**
Defined-in(setState(), Memento**)**	**and**
Defined-in(state, Memento**)**	**and**
Defined-in(setMemento(), Originator**)**	**and**
Defined-in(createMement0(), Originator**)**	**and**
Defined-in(operation(), Originator**)**	**and**
Modifier(getState(), protected**)**	**and**
Modifier(setState(), protected**)**	**and**
Reference-to-one(Caretaker, Memento**)**	**and**
Creation(Originator, Memento**)**	**and**
Invocation(operation(), createMemento()**)**	**and**
Invocation(operation(), setMemento()**)**	
Association(Originator, Memento**)**	

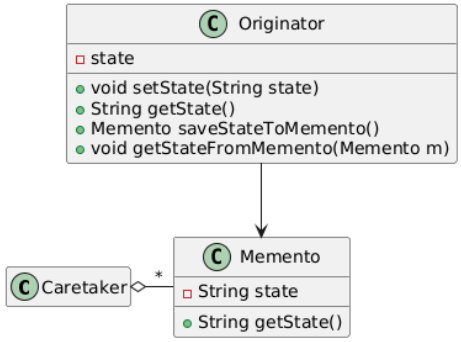

Figure 25 Exposed Memento example - Tutorials Point.

Symptoms and Smelly Structure: The state accessors and mutators defined in the Memento class are public (Figure 24(b)).

Memento vs. Exposed Memento in BPSL: Modifier(getState(), protected) and Modifier(setState(), protected) denote that only the Originator that belongs to the same package as Memento can access the state information stored in Memento. In the smelly structure, these predicates are false (Table 15).

Rational: The pattern should let the Originator store its state in an external object and restore it later, without violating encapsulation. Allowing objects other than the Originator to access the state stored in Memento is against the pattern intent.

Example: A typical example of Exposed Memento is illustrated in Figure 25. In this example, the Memento class provides the getState() method that returns the value of the state attribute. The getState() method is declared public. Therefore, access to the value of the state attribute is unrestricted.

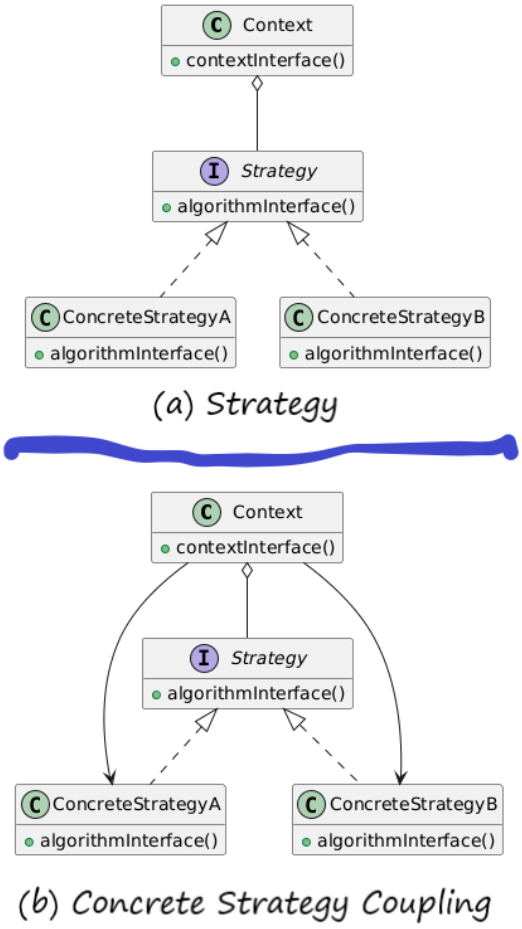

Figure 26 Strategy original and smelly structure.

6.7. Strategy

Original Intent and Structure The idea behind the Strategy pattern is to *"define a family of algorithms, encapsulate each one, and make them interchangeable. In other words, the pattern allows the algorithm to vary independently of the clients that use it"* (Gamma et al. 1994). Based on the original pattern structure, Strategy defines the interface common to all algorithms (Figure 26(a), Table 16). ConcreteStrategyA, ConcreteStrategyB, ConcreteStrategyC are algorithms that implement the Strategy interface. Context has a reference to a Strategy object that can be (re)configured to refer to any of the alternative algorithms.

Concrete Strategy Coupling

Symptoms and Smelly Structure: The Context class depends on the concrete classes that implement the algorithms, rather than relying only on the Strategy interface (Figure 26(b)).

Strategy vs. Concrete Strategy Coupling in BPSL: In the smelly structure, all the relations of the original pattern structure are still valid (Table 16). However, the smelly structure includes two additional associations, Association(Context, ConcreteStrategyA) and Association(Context, ConcreteStrategyB), between Context, ConcreteStrategyA and ConcreteStrategyB that should not be present.

Table 16 Strategy BPSL specification.

Strategy	
Exists Context, Strategy, ConcreteStrategyA, ConcreteStrategyB **in C;** **Exists** contextInterface(), algorithmInterface() **in M;**	
Defined-in(contextInterface(), Context**)** **Defined-in(**algorithmInterface(), Strategy**)** **Defined-in(**algorithmInterface(), ConcreteStrategyA**)** **Defined-in(**algorithmInterface(), ConcreteStrategyB**)** **Implements(**ConcreteStrategyA, Strategy**)** **Implements(**ConcreteStrategyB, Strategy**)** **Reference-to-one(**Context, Strategy**)**	**and** **and** **and** **and** **and** **and**

Rationale: Coupling the Context class with the concrete classes that implement the Strategy interface makes it impossible to vary the algorithms used by the Context class, independently from the Context class. This approach is against the intent of the pattern.

Example: Figure 27, illustrates an example of Concrete Strategy Coupling found in Source Making. In this example, the StrategyContext class is responsible for creating the appropriate Strategy object that may belong to three alternative implementations of the Strategy interface, that is, StrategyCaps, StrategyExclaim, and StrategyStars. Based on this design, the addition/removal of classes that implement the Strategy interface involves changing the StrategyContext implementation.

6.8. State

Original Intent and Structure The intent State is to *"allow an object to alter its behavior when its internal state changes. The object will appear to change its class"*. The original pattern structure comprises a Context class (Figure 26(a), Table 17). State, is an interface with different concrete implementation classes (e.g., ConcreteStateA and ConcreteStateB), each encapsulating the behavior, associated with a respective Context state. The Context class refers to a State object that may belong to any concrete class that implements the State interface (e.g., ConcreteStateA or ConcreteStateB). A Context object delegates state-specific requests to the referenced State object that serves them. The Context object decides which state succeeds another and changes the referenced State object for another that encapsulates the behavior associated with the successive state.

Context Without Transitions

Symptoms and Smelly Structure: The state transitions and the reconfiguration of the referenced State object are not triggered by the Context object The referenced State object remains the same for the lifetime of the Context object, or it is altered to another by an external client using the Context object (Figure 28(b)).

State vs. Context Without Transitions in BPSL: According to the Request(client, context) action specification (Table 17), a

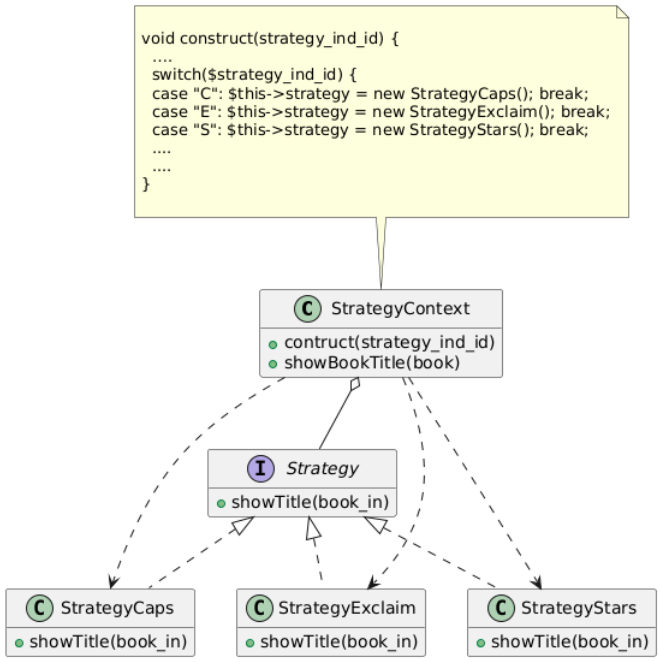

Figure 27 Concrete Strategy Coupling example - Source Making.

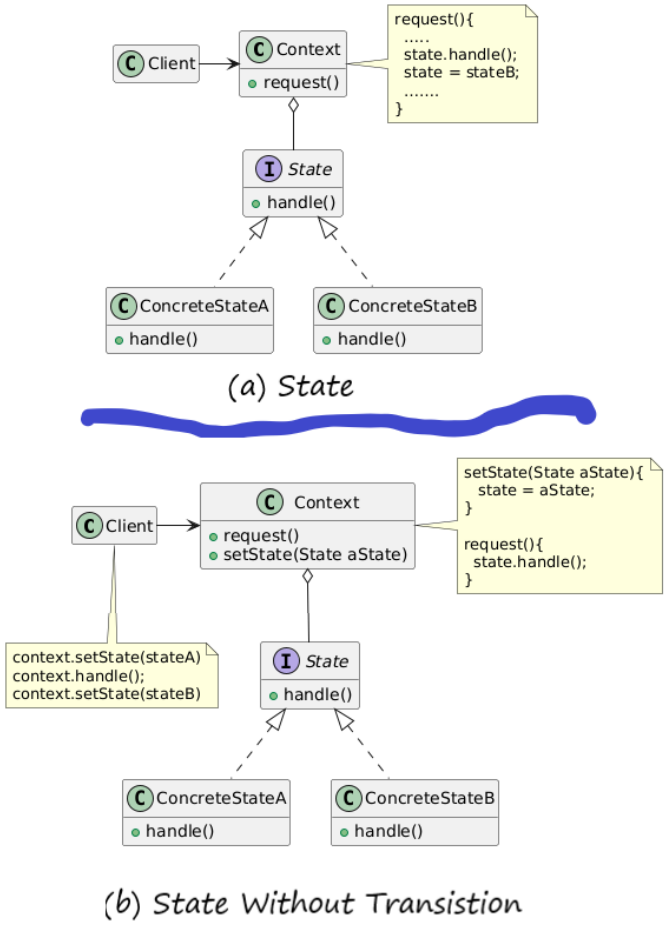

Figure 28 State original and smelly structure.

Table 17 State BPSL specification; Context Without Transitions violations are underlined and highlighted in red.

State	
Exists Client, Context, State, ConcreteStateA, ConcreteStateB **in C;** **Exists** request(), handle() **in M;** **Exists** state **in A;** **Exists** client, context, stateA, stateB **in O;**	
Defined-in(request(), Context) **Defined-in**(state, Context) **Defined-in**(handle(), State) **Defined-in**(handle(), ConcreteStateA) **Defined-in**(handle(), ConcreteStateB) **Implements**(ConcreteStateA, State) **Implements**(ConcreteStateB, State) **Reference-to-one**(Context, State) **Invocation**(request(), handle()) **Instance**(client, Client) **Instance**(context, Context) **Instance**(stateA, ConcreteStateA) **Instance**(stateB, ConcreteStateB) **Association**(Client, Context)	**and** **and** **and** **and** **and** **and** **and** **and** **and** **and** **and** **and** **and**
Request(client, context): **Association**(client, context) **and** context.state = stateA → **Handled**(context, stateA) **and** context.state' = stateB.	

request issued by a Client object that is associated with a Context object is handled by a ConcreteStateA object that is referenced by the Context object. The Context object then alters its own behavior by changing its state to refer to a ConcreteStateB object. In the smelly structure, the state change part of the action (context.state' = stateB) is not valid.

Rationale: In the smelly pattern structure, the Context object does not alter its behavior when its state changes, as it should according to the original intent of the pattern.

Example: Figure 29 shows an occurrence of the Content Without Transitions smell, found in Source Making. In this example, the main() function creates a ConcreteStateA object. Then, the main() function creates a Context object that refers to the ConcreteStateA object. The referenced State object remains unchanged for the lifetime of the Context object.

7. Summary of Findings & Next Steps

Is the information we find on the Web about the GoF patterns reliable?

Searching on the Web for information on GoF patterns should be done with caution. Often, the information we find on the Web about GoF patterns deviates from the original GoF pattern definitions. In this paper, we studied in detail a corpus of pattern instances that illustrate the usage of GoF patterns, found in

			Source Making		Refactoring		Tutorials Point		Java T Point		Wikipedia		TOTAL		
Pattern	Smell	Issue	# smells	freq.	# smells	freq.	# smells	freq.	# smells	freq.	# smells	freq.	# smells	# instances	freq.
Abstract Factory	Abstract Workshop	Context	3	0.43	0	0.00	1	1.00	0	0.00	1	1.00	5	20	0.25
Builder	Lazy Builder	Intent	0	0.00	1	0.10	0	0.00	0	0.00	1	1.00	2	18	0.11
	Autonomous Builder	Intent	0	0.00	0	0.00	1	1.00	1	1.00	0	0.00	2		0.11
Factory Method	Simple Factory	Intent	0	0.00	1	0.10	1	1.00	1	1.00	2	0.40	5	23	0.22
Composite	Minimal Component	Intent	9	1.00	3	0.30	0	0.00	0	0.00	0	0.00	12	22	0.55
Flyweight	Shared Only Flyweight	Intent	7	1.00	10	1.00	1	1.00	0		3	1.00	21	21	1.00
Proxy	Proxy Without Subject	Coupling	3	0.43	0	0.00	0	0.00	0	0.00	0		3	19	0.16
Chain of Responsibility	Fixed Chain	Coupling	2	0.33	0	0.00	0	0.00	0	0.00	0	0.00	2	19	0.11
Command	Command Without Invoker	Context	6	0.29	1	0.10	0	0.00	0	0.00	1	1.00	8	20	0.40
Interpreter	Hard-coded Grammar	Intent	2	1.00	0		0	0.00	1	1.00	0	0.00	3	8	0.38
Iterator	Monomorphic Iteration	Coupling	5	0.67	0	0.00	0	0.00	1	1.00	0	0.00	6	18	0.33
Mediator	Mediator that Doesn't Mediate	Intent	4	0.33	0	0.00	1	1.00	1	1.00	1	0.33	7	21	0.33
Memento	Exposed Memento	Intent	2	0.40	9	0.90	1	1.00	1	1.00	2	0.50	15	21	0.71
Strategy	Concrete Strategy Coupling	Coupling	2	0.40	0	0.00	0	0.00	0	0.00	0	0.00	2	18	0.11
State	Context Without Transitions	Intent	2	0.22	0	0.00	0	0.00	1	1.00	0	0.00	3	22	0.14
			47	0.52	25	0.19	6	0.43	7	0.54	11	0.46	96	270	0.36

Table 18 Summary of results.

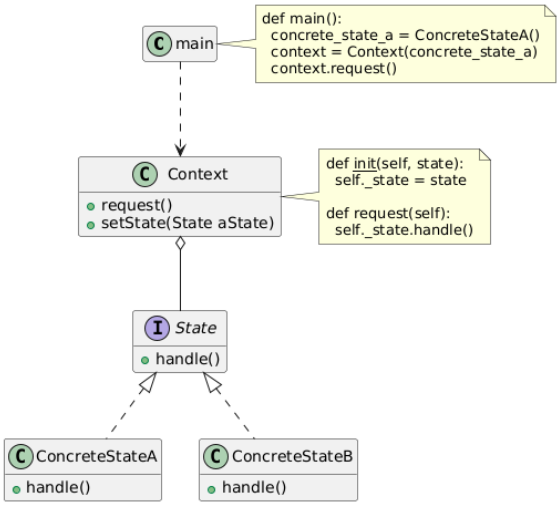

Figure 29 Context Without Transitions occurrence in Source Making.

five popular Web sites. Our result is a catalog of smells that comprises the most important deviations that we observed in the corpus. In general, we found 15 different smells in 14 of the 23 GoF patterns, that is, in 60% of the GoF patterns.

Specifically, 9 of the smells jeopardize the intent of the patterns. Then, we have 4 smells that concern pattern instances that violate the abstract coupling principle, that is, the pattern participants are strongly coupled because certain key abstractions are missing. Moreover, we have 2 smells related to pattern instances that are not used in the right context.

How likely is it to encounter smells in pattern instances we find on the Web?

In the overall corpus, the frequency of smells, that is, the number of smells over the number of pattern instances, is 0.36, which roughly means that we have one smell for every three pattern instances (Table 18). The frequency of smells found in the corpus varies from 0.11 to 1 for the different patterns. On average, the frequency of smells is 0.33 with a standard deviation of 0.25. The median for the number of smells in the corpus is 0.25. Practically, this means that the frequency of smells is low for most patterns. Flyweight, Memento, and Composite are the patterns with the highest frequency of smells. The frequency of smells in the pattern instances of each site ranges from 0.19 to 0.54. Refactoring Guru is the site with the lowest frequency of smells, i.e. 0.19. In the rest of the sites, the frequency of smells is more than double, while the variance between them is quite small.

What can we do to improve the current situation? Making a smell catalog is the least we can do to help developers recognize

these problematic cases when looking for information about GoF patterns on the Web. However, the detection of smells is up to the developers. Automating the detection of smelly pattern instances is a challenging issue for future research that could assist developers and address scalability. Investigating the role that AI technologies could play in smell detection is also an interesting issue. In this direction, we are developing a tool prototype that employs a graph-based approach to detect smelly GoF pattern structures in Java code and to compare the smelly structures with the original GoF pattern structures.

Acknowledgments

We would like to thank the reviewers of this paper for their helpful comments and suggestions.

References

Alfadel, M., Aljasser, K., & Alshayeb, M. (2020). Empirical study of the relationship between design patterns and code smells. *PLoS ONE*, *15*.

Almadi, S. H. S., Hooshyar, D., & Ahmad, R. B. (2021). Bad smells of gang of four design patterns: A decade systematic literature review. *Sustainability*, *13*.

Aversano, L., Canfora, G., Cerulo, L., Grosso, C. D., & Penta, M. D. (2007). An empirical study on the evolution of design patterns. In *Proceedings of the 6th joint meeting of the european software engineering conference and the acm sigsoft international symposium on foundations of software engineering (esec-fse)* (pp. 385–394). ACM.

Bieman, J. M., Straw, G., Wang, H., Munger, P. W., & Alexander, R. T. (2003). Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the 9th IEEE international software metrics symposium (metrics)* (pp. 40–49).

Cooper, J. W. (1998). *The design patterns java companion*. Addison-Wesley.

Dale, M. R., & Izurieta, C. (2014). Impacts of design pattern decay on system quality. In *Proceedings of the 8th acm/ieee international symposium on empirical software engineering and measurement ESEM* (pp. 37:1–37:4).

Feitosa, D., Ampatzoglou, A., Avgeriou, P., & Nakagawa, E. Y. (2018). Correlating pattern grime and quality attributes. *IEEE Access*, *6*, 23065–23078.

Feitosa, D., Avgeriou, P., Ampatzoglou, A., & Nakagawa, E. Y. (2017). The evolution of design pattern grime: An industrial case study. In *Proceedings of the 18th international conference on product-focused software process improvement (profes)* (Vol. 10611, pp. 165–181).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns - elements of reusable object-oriented software*. Addison-Wesley.

Gasparis, E., Nicholson, J., & Eden, A. H. (2008). Lepus3: An object-oriented design description language. In *Proceedings of the 5th international conference on diagrammatic representation and inference* (Vol. 5223, pp. 364–367). Springer.

Izurieta, C., & Bieman, J. M. (2013). A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, *21*(2), 289–323.

Khwaja, S., & Alshayeb, M. (2016). Survey on software design-pattern specification languages. *ACM Computing Surveys*, *49*(1).

Kim, D., France, R. B., Ghosh, S., & Song, E. (2004). A role-based metamodeling approach to specifying design patterns. *Software and Systems Modeling*, *3*(2), 163–189.

Mayvan, B. B., Rasoolzadegan, A., & Yazdi, Z. G. (2017). The state of the art on design patterns: A systematic mapping of the literature. *Journal of Systems and Software*, *125*, 93–118.

Prechelt, L., Unger, B., Philippsen, M., & Tichy, W. F. (2002). Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, *28*(6), 595–606.

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., & Votta, L. G. (2001). A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, *27*(12), 1134–1144.

Reimanis, D., & Izurieta, C. (2019). Behavioral evolution of design patterns: Understanding software reuse through the evolution of pattern behavior. In *Proceedings of the international conference on software reuse (icsr)* (Vol. 11602, pp. 77–93).

Taibi, T. (2007). *Design pattern formalization techniques*. IGI Global.

Taibi, T., & Ling, D. N. C. (2003). Formal specification of design patterns - A balanced approach. *Journal of Object Technology*, *2*(4), 127–140.

Vokác, M. (2004). Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, *30*(12), 904–917.

Walter, B., & Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, *74*, 127–142.

Zarras, A. (2021). The strategy configuration problem and how to solve it. In *Proceedings of the acm european conference on pattern languages of programs (EuroPLoP)* (pp. 9:1–9:11). ACM.

Zarras, A. V. (2020). Common mistakes when using the command pattern and how to avoid them. In *Proceedings of the acm european conference on pattern languages of programs (EuroPLoP)* (pp. 4:1–4:9). ACM.

Zarras, A. V., & Vassiliadis, P. (2023). A safari for deviating gof pattern definitions and examples on the web. In *Proceedings of the 42nd international conference on conceptual modeling (ER)* (pp. 181–197). Springer.