

Analysis of Design Patterns

Stage V

HSE Team
hsalekh@hse.ru
HSE
Moscow, Russia

Abstract

Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. Eolang programming language is a new Object-Oriented Programming language that has been going through many phases of development in this research. In this phase, we analyze typical design patterns in Java and C++ by researching multiple popular open-source repositories, to detect common design patterns and their usage statistics and then suggest alternatives in Eolang that could replace such patterns which are not supported in the language. We additionally provide explanation why Eolang alternative could be better

Keywords: OOP, C++, Java, design patterns

1 Introduction

This report includes pattern analysis, usage statistics, comparison of design patterns implementation in C++ and Java, implementation of some popular design patterns in Eolang, explanation of why Eolang does not support some design patterns, description of alternatives and explanation of why Eolang alternatives could be better.

1.1 Background

Design is one of the most difficult task in software development [22] and Developers, who have eagerly adopted them over the past years [33], needed to understand not only design patterns [27] but the software systems before they can maintain them, even in cases where documentation and/or design models are missing or of a poor quality. In most cases only the source code as the basic form of documentation is available [30]. Maintenance is a

time-consuming activity within software development, and it requires a good understanding of the system in question. The knowledge about design patterns can help developers to understand the underlying architecture faster. Using design patterns is a widely accepted method to improve software development [21]. A design pattern is a general reusable solution to a commonly occurring [24] problem in software design [9, 17, 20?]. A design pattern isn't a finished design that can be transformed directly into code neither are they static entities, but evolving descriptions of best practices [23]. It is a description or template for how to solve a problem that can be used in many different situations. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints [13, 28]. Design patterns help to effectively speed up development and engineering processes by providing proven development patterns/paradigms. Quality software design requires considering issues that may not be visible until later in the implementation. Reusing design patterns helps to avoid subtle issues that may be catastrophic and help improve code reliability for programmers and architects familiar with the patterns. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem. They help software engineers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs [8, 29]. In short, the advantages of design

patterns include decoupling a request from specific operations (Chain of Responsibility and Command), making a system independent from software and hardware platforms (Abstract Factory and Bridge), independent from algorithmic solutions (Iterator, Strategy, Visitor), or avoid modifying implementations (Adapter, Decorator, Visitor) [18]. Design patterns, overall, helps to thoroughly and designed well implemented frameworks enabling a degree of software reusability that can significantly improve software quality [26, 34?].

In this paper, we analyse typical design patterns in Java and C++ and detect common patterns and further look at their usage statistics. There are many design patterns in software development and several of them are common to Java and C++. These design patterns come under three main types.

2 Design Patterns

2.1 Creational design Patterns

These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done. Creational design patterns are the Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype [25, 35].

2.1.1 Use case of creational design pattern.

1. Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what developer will do is create an instance of DBConnection class and use it for doing database operations wherever required. Which results in creating multiple connections from the database as each instance of DBConnection class will have a separate connection to the database. In order to deal with it, we create DBConnection class as a singleton class, so that only one instance of DBConnection is

created and a single connection is established. Because we can manage DB Connection via one instance, we can control load balance, unnecessary connections, etc.

2. Suppose you want to create multiple instances of similar kind and want to achieve loose coupling then you can go for Factory pattern. A class implementing factory design pattern works as a bridge between multiple classes. Consider an example of using multiple database servers like SQL Server and Oracle. If you are developing an application using SQL Server database as back end, but in future need to change database to oracle, you will need to modify all your code, so as factory design patterns maintain loose coupling and easy implementation, we should go for the factory design pattern in order to achieving loose coupling and the creation of a similar kind of object.

2.1.2 Factory Method: Factory Method, also known as virtual constructor, is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created [11] in a way such that it doesn't have tight coupling with the class hierarchy of the library [14]. Factory Method is one of the most used design patterns in Java [5]. It is widely used in C++ code and is very useful when you need to provide a high level of flexibility for your code [10].

2.1.3 Abstract Factory. Abstract Factory patterns work around a super-factory which creates other factories. Thus, it defines a new Abstract Product Factory for each family of products. This factory is also called as factory of factories. It provides one of the best ways to create an object. Abstract Factory design pattern covers the instantiation of the concrete classes behind two kinds of interfaces, where the first interface is responsible for creating a family of related and dependent products, and the second interface is responsible for creating concrete products. The client is using only the declared interfaces and is not aware which concrete families and products are created [19]. Adding a new family of products affects any

existing class that depends on it, and requires complex changes in the existing Abstract Factory code, as well as changes in the application or client code [19]. Bulajic and Jovanovic [19] demonstrates a solution where adding a new product class does not require complex changes in existing code, and the number of product classes is reduced to one product class per family of related or dependent products.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern [4]. Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern [15].

Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So, at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type, [2]. Fig. 1 shows a UML class diagram example for an Abstract Factory Design pattern.

This pattern is particularly useful when the client doesn't know exactly what type to create. The Abstract Factory pattern helps you control the classes of objects that an application creates by isolating concrete classes. The class of a concrete factory appears only once in an application, that is where it's instantiated. This makes it easy to change the concrete factory an application uses. Abstract Factory makes this easy for an application use object from only one family at a time when product objects in a family are designed to work together. Abstract Factory interface fixes the set of products that can be created. This serves as a disadvantage because extending abstract factories to produce new kinds of Products is not easy. The abstract factory design pattern can be implemented in both Java and C++ as demonstrated at [1, 15]. The Abstract Factory pattern is pretty common in C++ code. Many frameworks and libraries use it to provide a way to extend and customize their standard components

2.1.4 Builder. Builder pattern builds a complex object using simple objects and using a step-by-step approach and the final step will return the object. The builder is independent of other objects. Fig. 2 show a UML diagram of builder design pattern. Immutable objects can be build without much complex logic in object building process. Builder design pattern also helps in minimizing the number of parameters in constructor and thus there is no need to pass in null for optional parameters to the constructor. As a disadvantage, it requires creating a separate ConcreteBuilder for each different type of Product [7, 16].

2.1.5 Singleton. In software engineering, the term singleton implies a class which can only be instantiated once, and a global point of access to that instance is provided [2]. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. It one of the simplest design patterns in Java and C++.

2.1.6 Object Pool. Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high. An Object pool is a container which contains some number of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.

2.1.7 Prototype. Prototype pattern refers to creating duplicate object while keeping performance in mind. This pattern involves implementing a prototype interface which tells to create a clone of the current object.

2.2 Structural

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality. Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy [25, 35].

2.2.1 Use Case of Structural Design Pattern. When 2 interfaces are not compatible with each other and want to establish a relationship between them through an adapter it's called an adapter

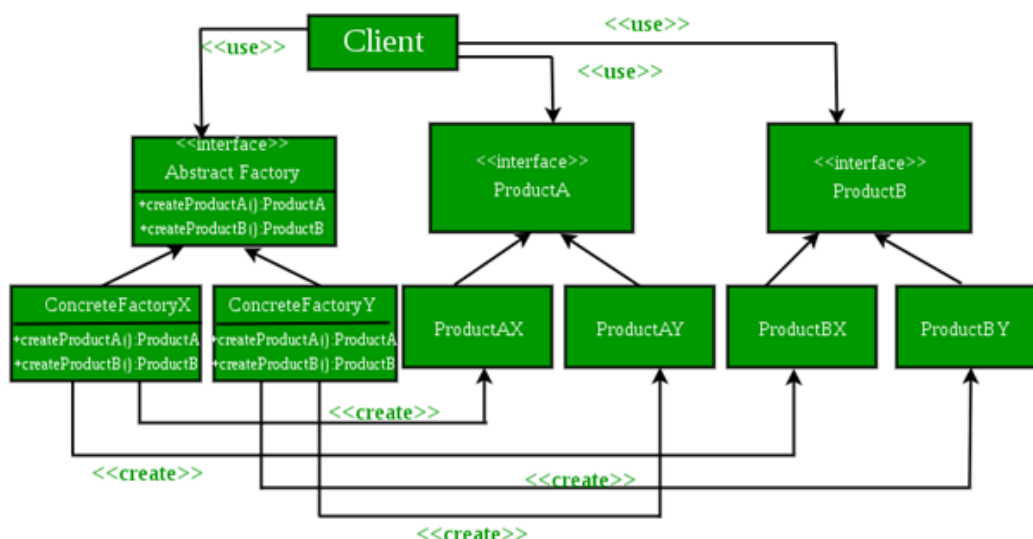


Figure 1. UML class diagram example for the Abstract Factory Design Pattern

design pattern. Adapter pattern converts the interface of a class into another interface or class that the client expects, i.e adapter lets classes work together that could not otherwise because of incompatibility. So, in these types of incompatible scenarios, we can go for the adapter pattern.

2.2.2 Adapter. Adapter design pattern allows objects with incompatible interfaces to collaborate [3]. Adapter pattern works as a bridge between those two incompatible interfaces. A real-life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop [6].

2.2.3 Bridge. The bridge pattern is used when we need to decouple [33] an abstraction from its implementation so that the two can vary independently [3]. It helps you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

2.2.4 Composite. Composite design pattern helps in composing objects into tree structures and then work with these structures as if they were individual objects. It is used where a group of objects need to be treated in similar way as a single object.

This pattern creates a class that contains group of its own objects and provides ways to modify this group of same objects.

2.3 Decorator

Decorator design pattern allows a user to add new functionality to an existing object without altering its structure. It allows the attachment of new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours.

2.3.1 Facade. Facade design pattern, as the name goes, hides the complexities of the system and provides an interface to the client by which the client can access the system. It provides a simplified interface to a library, a framework, or any other complex set of classes. In essence, it provides methods required by the client and delegates calls to methods of existing system classes.

2.3.2 Flyweight. The Flyweight pattern enables you to fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object. It is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.

2.3.3 Private Class Data. Private Class Data is used to encapsulate class data and control write access to class attributes as it separates data from methods that use it.

2.3.4 Proxy. The Proxy pattern controls access to the original object [33], allowing you to perform something either before or after the request gets through to the original object. It represents functionality of another class.

2.4 Behavioural

Behavioral patterns are about identifying common communication patterns between objects and realizing these patterns. Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method and Visitor [25, 35].

2.4.1 Use Case of Behavioral Design Pattern. The template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes. The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. For example, in your project, you want the behavior of the module to be able to extend, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, no one is allowed to make source code changes to it, i.e. you can add but can't modify the structure in those scenarios a developer can approach template design pattern [8, 13].

Sample implementation of these patterns [12] are available here in both Java and C++.

2.4.2 Chain of responsibility. Chain of responsibility pattern suggests a chain of receiver objects for a request. It decouples sender and receiver of a request based on type of request. It allows you to pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

2.4.3 Command. Command pattern is also known as action or transaction pattern. This pattern turns a request into a stand-alone object that contains all

information about the request. The transformation allows you pass requests as a method argument, delay or queue a request's execution, and support undoable operations. It is data driven and it wraps an object under an object as command [33] and passes it to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

2.4.4 Interpreter. Interpreter pattern provides a way to evaluate language grammar or expression. Given a language, interpreter defines a representation for the language's grammar along with an interpreter that uses the representation to interpret sentences in the language. Then it maps a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design. This pattern is used in SQL parsing, symbol processing engine etc. The Iterator pattern is very commonly used design pattern in Java and .Net programming environment. It allows sequential traversal through a complex data structure without exposing its internal details [3]. This pattern is also common in C++ code. Mediator is used to reduce communication complexity between multiple objects or classes by providing a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling [3]. It encapsulates how a set of objects interact.

2.4.5 Memento. Memento pattern is used to restore state of an object to a previous state. Without violating encapsulation, you can capture and externalize an object's internal state so that the object can be returned to this state later.

2.4.6 Null Object. In Null Object pattern, a null object replaces check of NULL object instance. The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do-nothing behaviour in case data is not available.

2.4.7 Observer. Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically [33].

2.4.8 State. In State pattern, objects are created to represent various states and a context object whose behaviour varies as its state object changes. So, in State pattern, a class behaviour changes based on changes in its internal state.

2.4.9 Strategy. Strategy design pattern is the pattern where a class behaviour or its algorithm can be changed at run time.

2.4.10 Template. In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need, but the invocation is to be in the same way as defined by an abstract class. It allows the algorithm to vary independently from the clients that use it.

2.4.11 Visitor. In Visitor pattern, a visitor class changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. Visitor allows the definition of a new operation without changing the classes of the elements on which it operates.

3 Criticism

The concept of design patterns has been criticized by some in the field of computer science.

3.1 Targets the wrong problem

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay *Revenge of the Nerds* [8].

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the *Design Patterns* book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

3.2 Lacks formal foundations

The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with

their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by $\frac{2}{3}$ of the "jurors" who attended the trial.

3.3 Leads to inefficient solutions

The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern [8].

3.4 Does not differ significantly from other abstractions

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature [8].

According to [36], generality, precision, and understandability are the most important goals to consider in order to simplify software design pattern description.

4 Common design patterns in popular open-source repositories and their usage statistics

Hahsler [21] analyses the application of design patterns in Java by identifying patterns in projects using their log messages to look for names and descriptions. This attempt was done based on the idea that the names of design patterns become part

Patterns	Occurrences	%
No Pattern	183634	77.5%
Factory	20237	8.5%
Singleton	3331	1.4%
Observer	16061	6.8%
Template Method	5381	2.3%
Decorator	1513	0.6%
Factory + Observer + Singleton	485	0.2%
Observer + Template	953	0.4 %
Observer + Singleton	2390	1.0%
Factory + Observer	612	0.3%
Factory + Singleton	2279	1.0

Table 1. Frequencies of Pattern Occurrences and Percentage of Code Covered by the Patterns

of a common design language which developers use to communicate more efficiently. Fig. 2 shows the graph of the usage statistics according the approach of Hahsler [21].

Vokac [32] analysed the weekly evolution and maintenance of a large commercial product (C++, 500,000 LOC) over three years, comparing defect rates for classes that participated in selected design patterns to the code at large. He extracted design pattern information and concluded that Observer and Singleton patterns are correlated with larger code structures. The Template Method pattern was used in both simple and complex situations, leading to no clear tendency. The frequencies of pattern occurrence are shown in table 1.

5 Comparison of Design Patterns in C++ and Java

See Table 2.

6 Implementation of Some Design Patterns in Eolang

6.1 Abstract Factory

An abstract factory is a pattern that generates objects.

6.1.1 Purpose. Provides an interface for creating families of interconnected or interdependent objects without specifying their specific classes.

6.1.2 Structure. See Fig 3.

6.1.3 Participants.

1. AbstractFactory — abstract factory: declares an interface for operations that create abstract product objects.

2. ConcreteFactory — specific factory: implements operations that create specific objects-products.
3. AbstractProduct — abstract product: declares the interface for the type of object-product.
4. ConcreteProduct — specific product: defines the product object created by the corresponding particular factory, and implements the AbstractProductinterface.
5. Client: uses only interfaces that are declared in the AbstractFactory and AbstractProductclasses.

6.1.4 Implementation.

```

1. +package sandbox
2. +alias stdout org.eolang.io.stdout
3. +alias sprintf org.eolang.txt.sprintf
4.
5. [type] > abstractFactory
6.   if. > concreteFactory
7.     eq.
8.       type
9.       "1"
10.      concreteFactory1
11.      concreteFactory2
12.
13. [] > createProductA
14.   createProductA. > @
15.     ^.concreteFactory
16. [] > createProductB
17.   createProductB. > @
18.     ^.concreteFactory
19.
20. [] > concreteFactory1
21. [] > createProductA
22.   1 > @
23. [] > createProductB
24.   2 > @
25.
26. [] > concreteFactory2
27. [] > createProductA
28.   "one" > @
29. [] > createProductB
30.   "two" > @
31.
32. [args...] > appAbstractFactory
33.   abstractFactory > objFactory
34.   args.get 0
35.   stdout > @
36.   sprintf
37.     "ProductA: %s\nProductB: %s\n"
38.     objFactory.createProductA
39.     objFactory.createProductB

```

Output

```

$ ./run.sh 1
ProductA: 1

```

Category	Pattern	Java	C++
Creational	Factory method	Uses abstract keyword to declare factory methods in factory classes to be later implemented by subclasses	Uses static pointers to declare factory methods
	Abstract factory	Uses 'abstract' keyword to make abstract factories classes and interfaces	Uses 'class' keyword and pointers to create abstract factories and the 'new' keyword to create concrete factories which is later used to create concrete objects
	Builder	Defines classes with creation methods for creating or building complex objects.	An abstract base class declares the standard construction process, and concrete derived classes define the appropriate implementation for each step of the process. Uses struct and class keyword in process.
	Singleton	Uses the public keyword to define a single point of access method for classes	Make the class responsible for its own global pointer and "initialization on first use" (by using a private static pointer and a public static accessor method)
	Prototype	Uses the cloneable interface for implementing class prototypes	A superclass defines a clone method and subclasses implement this method to return an instance of the class
Structural	Adapter	The adapter class uses the extend keyword to extend another class to make it compatible with another class	An abstract base class is created that specifies the desired interface. An "adapter" class is defined that publicly inherits the interface of the abstract class, and privately inherits the implementation of the legacy component. This adapter class "maps" or "impedance matches" the new interface to the old implementation.
	Bridge	Use abstraction and implementation to decouple classes into several related hierarchies	Uses abstraction and implementation
	Composite	Uses inheritance to hierarchically implement an object tree	Uses inheritance and polymorphism to implement scalar/primitive classes and vector/container classes
	Decorator	Uses interface keyword to wrap subclasses and allow the subclasses to dynamically add new behaviours to objects	Uses the concept of wrapping-delegation which involves pointers to help add new behaviours to objects dynamically
	Chain of responsibility	Defines an abstract class with series of methods including abstract methods for other classes to implement and handle a chain of actions independently	Uses pointers and classes and defines a chain method in the base class for delegating to the next object.
Behavioural	Command	Applies the concept of inheritance and encapsulation to turn actions into objects	Applies the concept of inheritance and encapsulation to turn actions into objects
	Observer	Defines event listeners based on inheritance and encapsulation	Models the "independent" functionality with a "subject" abstraction and Models the "dependent" functionality with "observer" hierarchy
	Null object	Encapsulates the absence of an object by providing a substitutable alternative that offers suitable default do nothing behaviour	Similarly, provides a class that checks for null and returns a boolean
	Mediator	Uses 'interface' keyword to declare mediators which are later implemented by classes that intend to communicate with each other	Uses classes and pointers to implement mediators

Table 2. Comparison of Design Patterns in C++ and Java

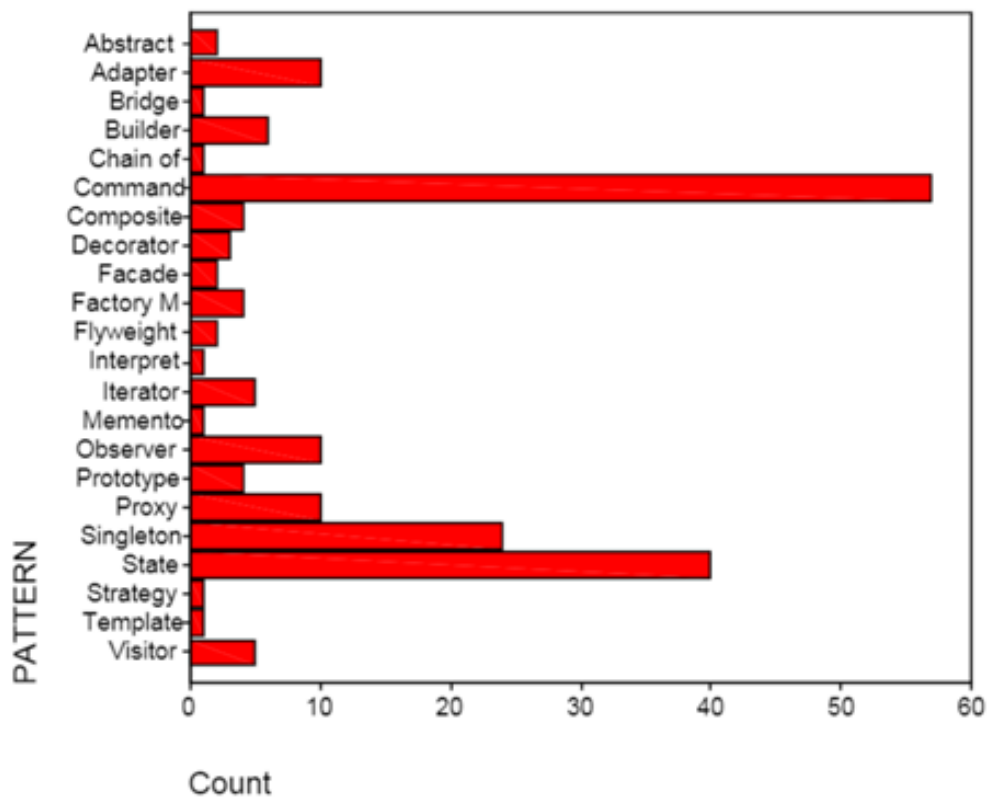


Figure 2. Number of Java Projects using individual design patterns [21]

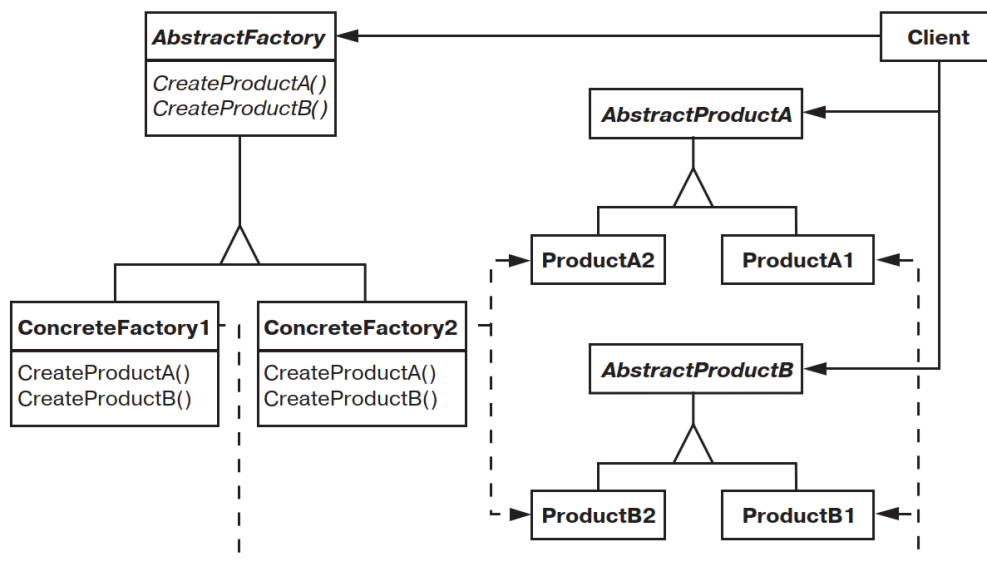


Figure 3. Abstract Factory

ProductB: 2
\$./run.sh 2

ProductA: one
ProductB: two

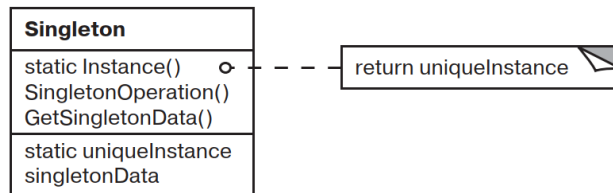


Figure 4. Singleton Design Pattern

This program creates objects of integers or strings depending on the args parameter [0]. If args[0] = 1, then objects 1 and 2 will be created, otherwise - "one" and "two". The template assumes the use of interfaces that are not present in the EO. In this case, an attempt was made to implement the interface through the EO object has a type parameter depending on which a specific implementation of the object factory is selected. This makes the interface object dependent on the set of implementations of this interface (when adding anew implementation, you must make changes to the interface object).

6.2 Singleton (singles)

A singleton is a pattern that generates objects.

6.2.1 Purpose. Ensures that the class has only one instance and provides a global access point to it.

6.2.2 Structure. See fig. 4.

6.2.3 Participants. Singleton Singleton:

1. Defines an Instance operation that allows clients to access a single instance. Instance is a class operation, that is, a static method of a class
2. May be responsible for creating your own unique instance.

6.2.4 Relations. Clients access an instance of the Singleton class only through its Instance operation.

6.2.5 Implementation. There are no classes in the EO, so this pattern is not implemented in its pure form. If we define Singleton in terms of EO as an object that is guaranteed to have only one copy, then the implementation of this object is also impossible for the following reasons:

1. There are no references in the EO. Any use of an object in a location other than the place of definition is a copy of this object.
2. EO does not have the ability to restrict access to objects or prevent it from being copied. You cannot restrict the creation of copies of an object.

6.3 Prototype

A prototype is a pattern that generates objects.

6.3.1 Purpose. Specifies the types of objects to create using the prototype instance and creates new objects by copying the prototype.

6.3.2 Structure. See fig 5.

1. — prototype: declares an interface for cloning itself.
2. — Concrete Prototype: implements the operation of cloning itself.
3. — client: creates a new object by asking the prototype to clone itself.

6.3.3 Implementation. In Eolang, each object can be copied, and each object can perform template functions.

6.4 Observer

In EO, all objects have immutable state. Based on the purpose of the template, its use in EO is pointless.

6.5 Bridge

A bridge is a pattern that structures objects.

6.5.1 Purpose. Separate abstraction from its implementation so that both can be changed independently.

6.5.2 Structure. See fig. 6

6.5.3 Participants.

1. Abstraction — abstraction: defines the abstraction interface, and stores a reference to an object of type Implementor.
2. RefinedAbstraction — refined abstraction: extends the interface defined by abstraction abstraction.
3. Implementor — implementer: defines the interface for the implementation classes. it does not have to exactly match the interface of the abstraction class. In fact both interfaces can be completely different. usually the interface of the Implementor class provides only primitive operations, and the Abstraction class defines higher-level operations based on these primitives.
4. ConcreteImplementor — specific implementer: implements the interface of the Implementor class and defines its specific implementation.

6.5.4 Relations. The Abstraction object redirects client requests to its Implementor object.

6.6 Chain of responsibility

A chain of responsibilities is a pattern of behavior of objects.

6.6.1 Purpose. Avoids binding the sender of the request to its recipient by providing the ability to process the request to multiple objects. Binds the receiving objects to the chain and passes the request along that chain until it is processed.

6.6.2 Structure. See fig. 7.

6.6.3 Participants.

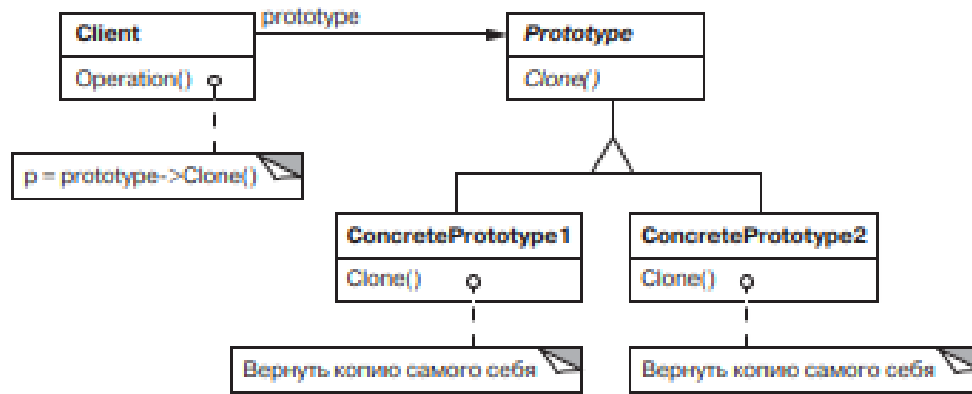


Figure 5. Prototype Design Pattern

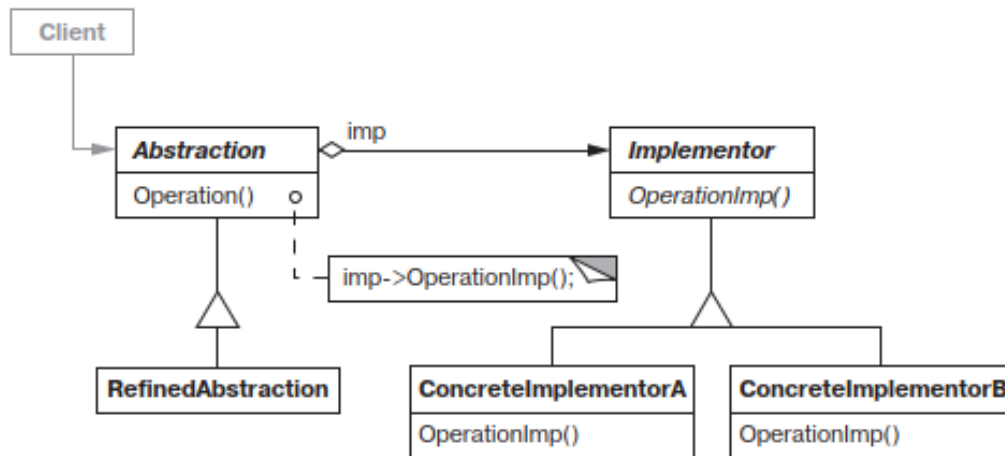


Figure 6. Bridge Design Pattern

1. Handler — handler: defines the interface for processing requests; (optionally) implements communication with the successor.
2. ConcreteHandler — specific handler: processes the request for which it is responsible; Has access to his successor; If ConcreteHandler is able to process the request, it does so, if it cannot, it sends it to its successor;
3. Client: Sends a request to some ConcreteHandler object in the chain.

6.6.4 Relation. A request initiated by a client is moved along the chain until some ConcreteHandler object takes responsibility for processing it.

6.6.5 Implementation.

1. `+package sandbox`
2. `+alias stdout org.eolang.io.stdout`
3. `+alias sprintf org.eolang.txt.sprintf`

- 4.
5. `[nextHandler] > defaultHandler`
6. `[message] > process`
7. `"" > @`
- 8.
9. `[] > handler1`
10. `[message] > process`
11. `if. > @`
12. `message.eq "1"`
13. `"one"`
14. `^.nextHandler.process message`
15. `defaultHandler > @`
16. `handler2`
- 17.
18. `[] > handler2`
19. `[message] > process`
20. `if. > @`
21. `message.eq "2"`

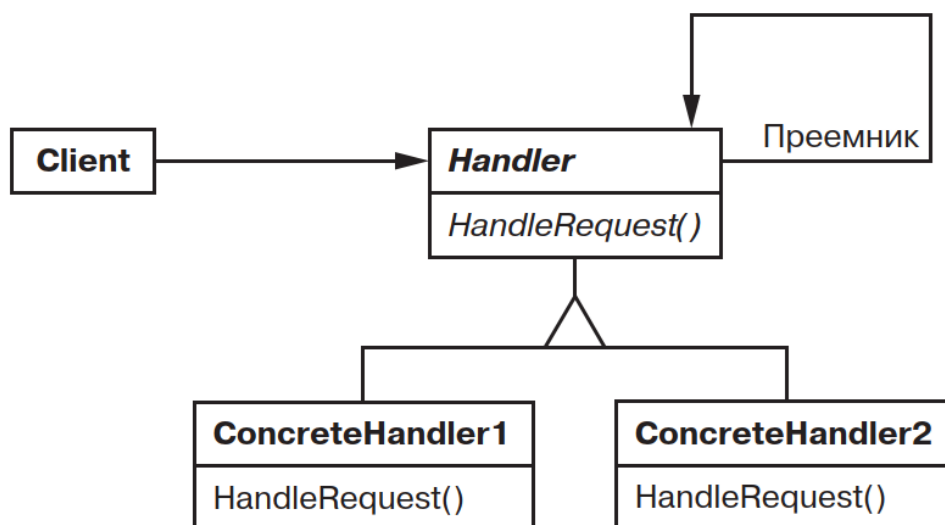


Figure 7. Chain of responsibility Design Pattern

```

22.     "two"
23.     ^.nextHandler.process message
24.     defaultHandler > @
25.     handler3
26.
27. [] > handler3
28. [message] > process
29.     if. > @
30.         message.eq "3"
31.         "three"
32.         ^.nextHandler.process message
33.     defaultHandler > @
34.     handler4
35.
36. [] > handler4
37. [message] > process
38.     if. > @
39.         message.eq "4"
40.         "four"
41.         ^.nextHandler.process message
42.     defaultHandler > @
43.     defaultHandler
44.
45. [args...] > appChain
46.     handler1 > hChain
47.     stdout > @
48.     sprintf
49.     "%s\n"
50. hChain. process
51.     args.get 0
  
```

The input parameter `args[0]` is passed sequentially to 4 handlers, each of which processes its value (numbers from 1 to 4 are converted into words if another parameter is entered, an empty string is returned).

6.7 Command

Command pattern is a behavioral design pattern.

6.7.1 Purpose. Encapsulates a query in an object, thereby allowing clients to be parameterized for different requests, queued or logged requests, and supports cancellation of operations.

6.7.2 Structure. See fig. 8

6.7.3 Participants.

1. - Command – command: declares the interface to perform the operation.
2. - ConcreteCommand is a specific team: defines the relationship between the Receiver receiving object and the action; implements the Execute operation by calling the corresponding operations of the Receiverobject.
3. – Client— client: Creates a ConcreteCommand class object and sets its recipient.
4. - Invoker— initiator: calls the command to execute the request.
5. - Receiver — recipient: has information about how to perform the operations necessary to meet the request. Any class can act as a recipient.

6.7.4 Relations.

1. - The client creates a ConcreteCommand object and sets a recipient for it.

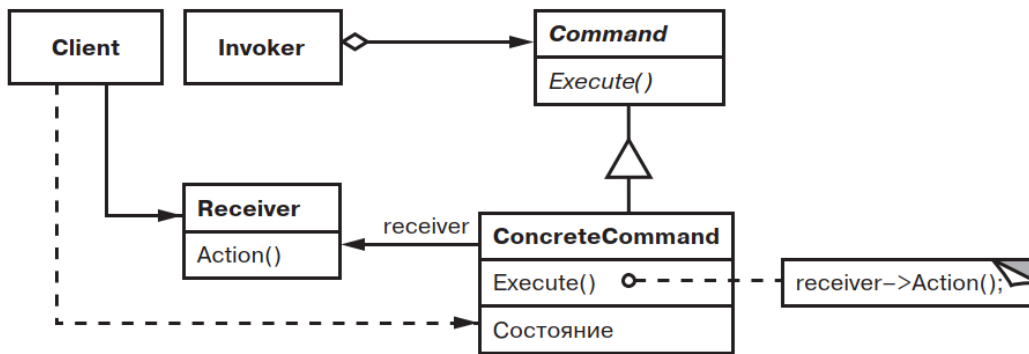


Figure 8. Command Design Pattern

2. - The Invoker initiator saves the ConcreteCommand object.
3. - The initiator sends a request by calling the ExecuteCommandOperation. If undoing of actions performed is supported, ConcreteCommand stores sufficient status information to perform the cancellation before calling Execute.
4. - The ConcreteCommand object invokes the recipient's operations to execute the request.

6.8 Null

The Null Object Pattern is one of the behavioral design patterns.

6.8.1 Purpose. Null object pattern is used to replace check of NULL object instance to simplify the use of dependencies that can be undefined.

6.8.2 Problem. In Null Object pattern, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do-nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available. The concept of null objects comes from the idea that some methods return null instead of real objects and may lead to having many checks for null in your code.

In Java and C++, the key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class. Because it conforms to the abstract class' interface, it can be used any place this type of object is needed. Null object should not have any state.

In Eolang, the concept of null does not exist as every object is meant to dataize to a value, and as such given a value on creation. As classes and interfaces do not exist here either, the closest implementation in Eolang would be to have every object implement a null attribute that either dataizes to a Boolean or a string representing a lack of value/data or whatever the default value of an object may be. In this

case, there may still be checks to see if null is true or false or contains the expected string.

6.8.3 Implementation.

```

1. +package sandbox
2. +alias sprintf org.eolang.io.sprintf
3.
4. [] > null
5. "null" > @
6.
7. [args...] > appNull
8. sprintf > @
9. if.
10.     args.isEmpty
11.     null
12.     args.get 0
    
```

output:

```
run.cmd null
```

6.9 Decorator

Decorator is a structural design pattern.

6.9.1 Purpose. Decorator pattern allows you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

6.9.2 Problem. Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

In Eolang, a copy of an object can be made, and new functionality be added. Here, the original object represents the decorator.

6.9.3 Implementation.

```

1. +package sandbox
2. +alias sprintf org.eolang.io.sprintf
3.
    
```

```

4. [] > carsDecorate
5.   8 > @
6.
7. [num] > someCars
8.   decorateWithMoreCars num > @
9.
10.  [number] > decorateWithMoreCars
11.    add. > @
12.      carsDecorate
13.      number
14.
15. [args...] > appDecorator
16.   stdout > @
17.     someCars (args.get 0)

```

In this example, the object “someCars” increases the number of cars in decarDecorator is for itself.

Output

```
run.cmd 5 13
```

It can be concluded that decorator design pattern naturally exists in Eolang.

6.10 Builder

Suppose, we have a class with a variety of input parameters. The input parameters are used to configure instances of the class. Some of the parameters may be optional, while some of them are mandatory to be set up. Hence, the following techniques of configuration of instances of the class may be applied:

1. Configuration of instance variables of an object directly in the user code through Setter Methods calls or by referencing variables straightforwardly. This practice may not be considered appropriate because it makes code instances more cohesive and interdependent while violating encapsulation of the inner state of objects (which may lead to breaking of the integrity of business logic of an application), and, hence, the usage of the practice is not encouraged. In addition, this technique may allow situations in which objects are being in an incomplete state, which also may break the logic of an application.
2. Creation of subclasses of the considered class when each successor has a slightly changed prototype of its constructors. This technique implies that prototypes of constructors of different subclasses have subsets of optional parameters in them while omitting some parameters, which makes it possible to create configurable instances of objects in a controlled manner. This practice is more encouraged to be applied in practice since it implies control over the creational process. However, it is not recommended for use when choosing the sole parent superclass is challenging or when the practice produces a wide or deep hierarchy of inheritance

3. Overloading of constructors or setting a single constructor with optional parameters. While this practice allows classes to create instances in a controlled way, it is undesirable in cases where the number of parameters or constructor overloads is too large to be manageable and understandable.

The Builder pattern may be considered a universal solution to the problem. The pattern defines the Builder class, which has methods (stages) for building objects. The user code can call the stages in any order, omitting some of them (optional configurations). Also, the Builder class may check that all the required parameters are set up. At the end of construction, user code is required to call a method that finishes the construction process and returns a ready-made object. The pattern encapsulates the creational sensitive logic inside the Builder class and makes the configuration process manageable to the user code.

6.10.1 Structure. See fig. 9.

6.10.2 Code Instances Involved. Builder is an abstract class that defines the contract of the creational steps of Products for its successors (concrete builders). Also, the Builder superclass defines the finalize method. Product is an interface for products (instances being created and configured through the Builder pattern). The interface defines the contract for all products so that these may be managed by Builders. (optional) Director is a class, which defines higher-level (that is, higher than the level of “understanding” of the builder itself, for example, rules for mandatory fields and compliance with business logic) scripts for building objects. The director can be used to reuse some high-level business logic for constructing objects based on various builder implementations.

6.10.3 Relations. Implementations of the Product interface are products. Inheritors of the Builder class provide concrete implementations for the building steps (or borrow some of those steps from the superclass). The Director (optional entity) class can manage builders in a general style (based on some configuration) in accordance with the higher-level logic of business rules. The client code can contact the Director, giving it the configuration, or build an object using the Builder directly.

6.10.4 Implementation. First, we should mention that the problem solved by the Builder pattern may be addressed by the partial application mechanism embedded into the language as one of its features. The partial application mechanism allows programmers to partially apply objects (i.e., create objects with some or all of the input attributes omitted and then, optionally, set unbound attributes after throughout the program). This technique may be utilized as a more concise alternative to constructor overloading. Here is an example:

```
[a b c name] > triangle
```

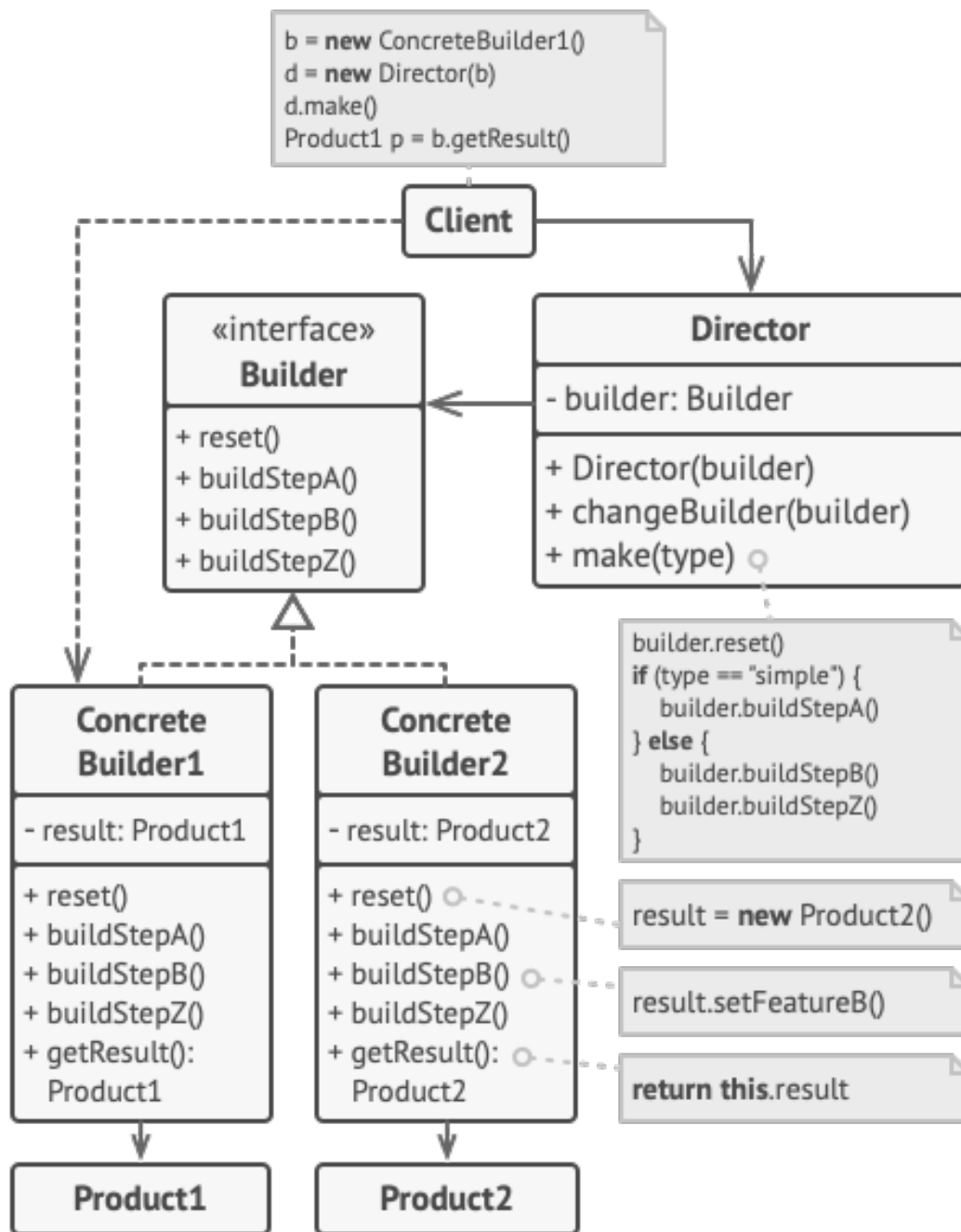


Figure 9. Builder Design Pattern

```

add. > perimeter
add.
  a
  b
  c

sprintf > toString
"The triangle is named '%s'."
    
```

```

name

[args...] > app
triangle > triangleA
10:a
triangleA > triangleABC
7:b
    
```

```

8:c
triangleABC > triangleABCNamed
  "My Triangle":name
triangle > triangleNamed
  "My Another Triangle":name

```

Here, we have the triangle object with input attributes a, b, c, and name. The triangle object has two bound attributes: perimeter (which relies on a, b, and c) and toString (which relies on name). Object app demonstrates the partial application mechanism. So, triangleA has only the a attribute bound, triangleABC has all the sides (a, b, c) set up, triangleABCNamed has all the sides and its name configured, and triangleNamed has the name only. All three triangles are constructed through partial application (meaning, some of the attributes are left unbound or were bound after). The above example demonstrates an alternative solution to the problem of optional configuration of objects. However, this solution does not encapsulate the creation process of objects. Hence, the Builder pattern still may have its place in the EO environment.

Consider the following example:

```

[] > builder
subbuilder triangle > @
[triangleEntity] > subbuilder
  # finalizes the construction process
  [] > finalize
    ^.^subbuilder > @
  # configures the a free attribute
  [aVal] > setA
    ^.^subbuilder > @
    ^.^triangleEntity
    (^validateSide aVal):a
  # configures the b free attribute
  [bVal] > setb
    ^.^subbuilder > @
    ^.^triangleEntity
    (^validateSide bVal):b
  # configures the c free attribute
  [cVal] > setc
    ^.^subbuilder > @
    ^.^triangleEntity
    (^validateSide cVal):c
  # configures the name free attribute
  [nameVal] > setname
    ^.^subbuilder > @
    ^.^triangleEntity
    (^validateName nameVal):name
  # validates side value
  [val] > validateSide
    if. > @
      val.greater 0
      val
      error

```

```

"The side of a triangle must not be less than 1!"
# validates name
[val] > validateName
if. > @
  val.length.neq 0
  val
  error
"The name of a triangle must not be empty!"

```

```

[args...] > app
builder > b
finalize. > triangleABC
  setC.
  setB.
  setA.
    builder
    10
    12
    0

```

Here we implemented the principles of the Builder pattern through measures supplied by the EO language. The builder object contains the subbuilder attribute object that implements the construction steps (setA, setB, setC, setName) as well as validation sub-steps (validateSide, validateName) and the finalize attribute that finishes the construction process and returns the resulting object. Initially, the instantiation of the builder object is substituted with a copy of the subbuilder object with an empty (meaning, all free attributes are unbound) copy of the constructing object. On each construction step, the subbuilder object returns itself by passing a changed version of the constructing object to its constructor. The construction steps have validation substeps that may implement some complex business logic. Validation steps may return an error or a validated object, which may lead to an interruption of the program execution and prevent inconsistency of the business logic. In conclusion, we would like to notice that the problem originally stated above (problem of optional configuration of objects with a lot of input parameters) and solved with the Builder pattern may not be actual to EO since it has the partial application mechanism that allows programmers to perform such configuration and, in addition, EO does not allow objects to have more than four free attributes (although, this restriction may be mitigated through passing complex object structures as free attributes). Nevertheless, the EO implementation of the Builder pattern may find its utilization in scopes where encapsulation of the creational process of objects is required. For instance, it may be needed when business logic validation of values passed for binding to free attributes is required.

6.11 Factory Method

The Factory Method Pattern is a creational object-oriented design pattern.

6.11.1 Purpose. Defines the creational method in the Creator superclass that defines a rule (that is, an interface or a contract) for creating an object (product) of some supertype Product. This method is used by the superclass or its more specific implementations, and the factory method can also be called from outside the class by other entities within the application. Concrete implementations of the Creator class with a factory method can return subtypes of the Product type, thereby "tailoring" a specific implementation of the product class to the one required by the factory method contract. Hence, the pattern allows programmers to implement seamless configuration of the architecture of the application.

6.11.2 Problem. The pattern addresses the problem of extending the architecture of an application. By specifying the product contract (Product Interface) and by defining the class contract with the Factory Method Class, the architect separates the responsibility for creating the product itself from other methods of the creator class. This can be useful when:

1. It is not known what types of the Product class may be used in the future, but it may be appropriate to leave a headroom for a potential extension of the application architecture. Otherwise, this can be interpreted as an implementation of the "Open / Closed" principle (O in SOLID).
2. Implementation of the principle of "Single Responsibility" (S in SOLID). The code responsible for setting (configuring) a specific version of the product can be placed in a single place, for example, in a class that configures the application based on the environment settings. Here, the Dependency Injection mechanism can also be used to perform such a configuration in an invisible manner.
3. The pattern allows programmers to separate the logic of product creation from other logic of the creator class. This facilitates the reuse of identical code.

6.11.3 Structure. See fig. 10.

6.11.4 Code Instances Involved. Creator is an abstract class that defines the contract of the reutilized steps (here, it is someOperation) and the creational step (createProduct) of Products for its successors (concrete creators). Product is an interface for products (instances being created and configured through the Factory Method pattern). The interface defines the contract for all products so that these may be managed by the pattern.

6.11.5 Relation. Implementations of the Product interface are products. Inheritors of the Creator class provide concrete implementations for the creational method and inherit the rest methods. The concrete implementation of the creational method may return different implementations of Product, which implies the substitution of logic (or configurability of the application).

6.11.6 Implementation. The EO programming language does not have interfaces, classes, and types. Because of it, we may omit defining the Product interface contract (since it would not impose any requirements). Consider the following implementation of the pattern in EO:

```
[] > creator
  # left to be redefined
>[] > createObject
  # operation over products
>[] > performOperation
  createObject.getWeight.add 1 > @

>[] > concreteCreatorA
  creator > @
>[] > createObject
  productA > @

>[] > concreteCreatorB
  creator > @
>[] > createObject
  productB > @

>[] > productA
  # let's suppose that this implementation
  # gets value from the production server
>[] > getWeight
  42 > @

>[] > productB
  # let's suppose that this implementation
  # gets value from the testing server
>[] > getWeight
  24 > @
```

Here, we have the creator object with the performOperation attribute (the logic to be reused) and the createObject attribute (the logic to be redefined for flexible substitution of objects). The concreteCreatorA and concreteCreatorB objects have the creator object as their decorator, so that these might inherit the reusable logic. Both objects define the createObject attribute that hides the original attribute of the same name from the decoration hierarchy. Objects productA and productB implement the attribute of interest (getWeight) differently. One of them may get the value from the production server, while another takes it from the testing environment. This example demonstrates the implementation of the classic version of the pattern in EO. However, we may consider a more EO-idiomatic example, free from additional structures (concrete creators) utilized in statically typed class-based object-oriented languages such as Java or C++. Consider the following example:

```
[@] > creator
  # operation over products
>[] > performOperation
```

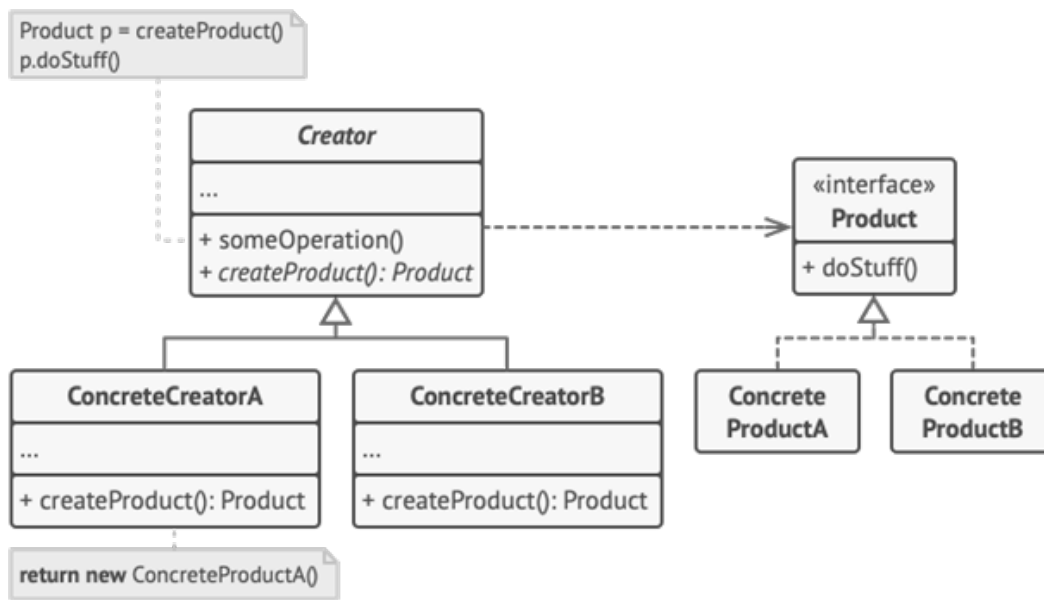


Figure 10. Factory Method Design Pattern

```

createObject.getWeight.add 1 > @

[] > productA
# let's suppose that this implementation
# gets value from the production server
[] > getWeight
42 > @

[] > productB
# let's suppose that this implementation
# gets value from the testing server
[] > getWeight
24 > @

[args...] > app
creator > creatorObject
[]
[] > createObject
if. > @
  (args.get 0).eq "test"
  productB
  productA

```

Here, we used the technique of passing decorator as a free attribute of the object creator. Its decorator is passed in the app object. The decorator has the only attribute createObject that the creator object inherits and relies on. The createObject attribute decides what version of a product should be chosen based on the environment configuration. This implementation of the pattern may be considered as more idiomatic and flexible from the EO perspective.

6.12 The Closures Functional Programming Technique

Since the EO programming language may be considered semi-functional, it might be useful to apply one of the widely adopted functional programming techniques, closures, in it. Simply put, the closures mechanism implies capturing outer lexical scope variables inside a function defined inside the scope with a consequent utilization of the function in other scopes. To support this technique, a language must operate over function as if they are first-class citizens (i.e., a language must return function or pass functions as parameters). Here is an example of this technique in JavaScript:

```

function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}
var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(2)); // 7
console.log(add10(2)); // 12

```

Here, we have the makeAdder function that returns an anonymous function capturing its outer state x. The state is then utilized when the returned function is applied with its own parameter y. In other words, the inner anonymous function remembers the value of x and uses it even when the actual value disappeared from the stack. This technique may be useful to emulate access modifiers in functional languages:

```

var counter = (function() {

```

```
var privateCounter = 0;
function changeBy(val) {
  privateCounter += val;
}

return {
  increment: function() {
    changeBy(1);
  },

  decrement: function() {
    changeBy(-1);
  },

  value: function() {
    return privateCounter;
  }
};
})();

console.log(counter.value()); // 0.

counter.increment();
counter.increment();
console.log(counter.value()); // 2.

counter.decrement();
console.log(counter.value()); // 1.
```

Here, the outer function counter returns a complex object-like structure containing functions that capture the state of the counter function. The state of the counter function is also complex: it has a mutable local variable privateCounter, and the changeBy function that mutates the value in the unified manner. The user code may not access the value and the mutating functions directly: both of them disappeared from the stack. However, the closures returned by the outer function still may do it. Hence, the technique allows functional programmers to simulate private state.

We surely may reproduce the similar technique of capturing the lexical scope in EO:

```
[] > counter
memory 0 > privateCounter
[val] > changeBy
  privateCounter.write > @
  privateCounter.add val
[] > @
[] > increment
  ^.^changeBy 1 > @
[] > decrement
  ^.^changeBy (-1) > @
[] > value
  ^.^privateCounter > @
```

7 Conclusion

It is possible to conclude that (1) EO is principally applicable to all the considered patterns; (2) For some patterns, EO is able to give a fairly concise and intuitively clear code, since the language combines the features of Functional Programming and OOP; (3) the issues of effective implementation of patterns on EO are largely determined by the characteristics of the environment (IDE + compiler) and today remain open.

Also, EO has no local variables or any kind of stack-lifetime storage. Instead, any name refers to an object (stored in heap) that may be accessed through the scope of any other object via the dot-notation mechanism. Even anonymous objects may allow programmers to access its local scope (including parent and decoration hierarchies) freely. In addition, EO has no access modification instruments. This makes closures technique almost similar to the partial application mechanism. Moreover, the publicity of any attribute of any object makes encapsulation impossible in the language. This differentiates EO from functional programming languages and, also, from object-oriented languages. Absence of instruments of access modification (or simulation of it) may be a severe violation of object-oriented principle of encapsulation, which may lead to insecure environments breaking business logic of problem domains.

EO is fundamentally applicable to all the patterns considered. 2) For some patterns, EO is able to give a fairly concise and intuitively clear code, since the language combines the features of Functional Program (FP) and OOP. 3) the issues of effective implementation of patterns on EO are largely determined by the characteristics of the environment (IDE + compiler).

The implementation of the design patterns in Eolang is available at [31].

References

- [1] [n.d.]. Abstract Factory. <https://refactoring.guru/design-patterns/abstract-factory>
- [2] [n.d.]. Abstract Factory Design Pattern. <https://springframework.guru/gang-of-four-design-patterns/abstract-factory-design-pattern/>
- [3] [n.d.]. A catalogue of general-purpose software design patterns. 330–339. <https://doi.org/10.1109/TOOLS.1997.654742>
- [4] [n.d.]. Design Pattern - Abstract Factory Pattern - Tutorialspoint. https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- [5] [n.d.]. Design Pattern - Factory Pattern - Tutorialspoint. https://www.tutorialspoint.com/design_pattern/factory_pattern.htm
- [6] [n.d.]. Design Patterns - Adapter Pattern - Tutorialspoint. https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

- [7] [n.d.]. Design Patterns - Builder Pattern - Tutorialspoint. https://www.tutorialspoint.com/design_pattern/builder_pattern.htm
- [8] [n.d.]. Design Patterns and Refactoring. <https://sourcemaking.com>
- [9] [n.d.]. Design Patterns: Elements of Reusable Object-Oriented Software [Book]. <https://www.oreilly.com/library/view/design-patterns-elements/0201633612/> ISBN: 9780201633610.
- [10] [n.d.]. Design Patterns: Factory Method in C++. <https://refactoring.guru/design-patterns/factory-method/cpp/example>
- [11] [n.d.]. Factory Method. <https://refactoring.guru/design-patterns/factory-method>
- [12] [n.d.]. Huston Design Patterns. <http://www.vincehuston.org/dp/>
- [13] 2015. Design Patterns | Set 1 (Introduction). <https://www.geeksforgeeks.org/design-patterns-set-1-introduction/> Section: Design Pattern.
- [14] 2015. Design Patterns | Set 2 (Factory Method). <https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/> Section: Design Pattern.
- [15] 2017. Abstract Factory Pattern. <https://www.geeksforgeeks.org/abstract-factory-pattern/> Section: Design Pattern.
- [16] 2017. Builder Design Pattern. <https://www.geeksforgeeks.org/builder-design-pattern/> Section: Design Pattern.
- [17] G. Antonioli, G. Casazza, M. Di Penta, and R. Fiutem. 2001. Object-oriented design patterns recovery. *Journal of Systems and Software* 59, 2 (Nov. 2001), 181–196. [https://doi.org/10.1016/S0164-1212\(01\)00061-9](https://doi.org/10.1016/S0164-1212(01)00061-9)
- [18] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. 2007. An empirical study on the evolution of design patterns. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 385–394. <https://doi.org/10.1145/1287624.1287680>
- [19] Aleksandar Bulajic and Slobodan Jovanovic. 2012. An Approach to Reducing Complexity in Abstract Factory Design Pattern. *Journal of Emerging Trends in Computing and Information Sciences* 3, 10 (2012). Publisher: Citeseer.
- [20] J. Coplien. 1998. Software design patterns: common questions and answers. *undefined* (1998). <https://www.semanticscholar.org/paper/Software-design-patterns%3A-common-questions-and-Coplien/9544fccfd09a9315b29ced5bc1e69f572114b7ec>
- [21] Michael Hahsler. 2003. *A Quantitative Study of the Application of Design Patterns in Java*.
- [22] Seyed Mohammad Hossein Hasheminejad and Saeed Jalili. 2012. Design patterns selection: An automatic two-phase method. *Journal of Systems and Software* 85, 2 (Feb. 2012), 408–424. <https://doi.org/10.1016/j.jss.2011.08.031>
- [23] Jeffrey Heer and Maneesh Agrawala. 2006. Software Design Patterns for Information Visualization. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (Sept. 2006), 853–860. <https://doi.org/10.1109/TVCG.2006.178> Conference Name: IEEE Transactions on Visualization and Computer Graphics.
- [24] Shahid Hussain, Jacky Keung, and Arif Ali Khan. 2017. Software design patterns classification and selection using text categorization approach. *Applied Soft Computing* 58 (Sept. 2017), 225–244. <https://doi.org/10.1016/j.asoc.2017.04.043>
- [25] Partha Kuchana. 2004. *Software Architecture Design Patterns in Java*. Auerbach Publications, New York. <https://doi.org/10.1201/9780203496213>
- [26] Wolfgang Pree and Hermann Sikora. 1997. Design patterns for object-oriented software development (tutorial). In *Proceedings of the 19th international conference on Software engineering (ICSE '97)*. Association for Computing Machinery, New York, NY, USA, 663–664. <https://doi.org/10.1145/253228.253810>
- [27] Dirk Riehle and Heinz Züllighoven. 1996. Understanding and using patterns in software development. *Theory and Practice of Object Systems* 2, 1 (1996), 3–13. [https://doi.org/10.1002/\(SICI\)1096-9942\(1996\)2:1<3::AID-TAPO1>3.0.CO;2-#_eprint](https://doi.org/10.1002/(SICI)1096-9942(1996)2:1<3::AID-TAPO1>3.0.CO;2-#_eprint): <https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291096-9942%281996%292%3A1%3C3%3A%3AAID-TAPO1%3E3.0.CO%3B2-%23>.
- [28] W. Schaffer and A. Zundorf. 1999. Round-trip engineering with design patterns, UML, java and C++. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 683–684. <https://doi.org/10.1145/302405.302956> ISSN: 0270-5257.
- [29] Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. 1996. Software patterns. *Commun. ACM* 39, 10 (Oct. 1996), 37–39. <https://doi.org/10.1145/236156.236164>
- [30] D. Streitferdt, C. Heller, and I. Philippow. 2005. Searching design patterns in source code. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Vol. 2. 33–34 Vol. 1. <https://doi.org/10.1109/COMPSAC.2005.135> ISSN: 0730-3157.
- [31] HSE Team. 2021. Eo Design Patterns. Retrieved September 14, 2021 from <https://github.com/HSE-Eolang/eodesignpatterns>

- [32] Marek Vokac. 2004. Defect Frequency and Design Patterns: An Empirical Study of Industrial Code. *IEEE Transactions on Software Engineering* 30, 12 (Dec. 2004), 904–917. <https://doi.org/10.1109/TSE.2004.99>
- [33] P. Wendorff. 2001. Assessment of design patterns during software reengineering: lessons learned from a large commercial project. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. 77–84. <https://doi.org/10.1109/CSMR.2001.914971>
- [34] Cheng Zhang and David Budgen. 2012. What Do We Know about the Effectiveness of Software Design Patterns? *IEEE Transactions on Software Engineering* 38, 5 (Sept. 2012), 1213–1231. <https://doi.org/10.1109/TSE.2011.79> Conference Name: IEEE Transactions on Software Engineering.
- [35] W. Zimmer. 1995. Relationships between design patterns. *undefined* (1995). <https://www.semanticscholar.org/paper/Relationships-between-design-patterns-Zimmer/b7fd68d166ca62fc05fe267b69ac78c279c6ea4f>