Analysis of typical design patterns in Java and C++

Abstract

Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages. This paper seeks to analyse typical design patterns common to Java and C++ and their usage statistics.

Keywords*: OOP, design patterns, Java, C++, software design.*

Introduction

Design is one of the most difficult task in software development [1] and Developers, who have eagerly adopted them over the past years [2], needed to understand not only design patterns [3] 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 [4]. 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 [5].

A design pattern is a general reusable solution to a commonly occurring[6] problem in software design [7]–[9]. 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 [10]. 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 [11], [12].

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 [13], [14]. 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) [15]. Design patterns, overall, helps to thoroughly and designed well implemented frameworks enabling a degree of software reusability that can significantly improve software quality [16], [17].

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.

Creational

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 [18], [19].

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.

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 [20] in a way such that it doesn't have tight coupling with the class hierarchy of the library [21]. Factory Method is one of the most used design patterns in Java [22]. It is widely used in C++ code and is very useful when you need to provide a high level of flexibility for your code [23].

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 [24]. 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 [24]. Bulajic and Jovanovic [24] 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 [25]. Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern [26].

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 [26], [27]. Fig. 1 shows a UML class diagram example for an Abstract Factory Design pattern.

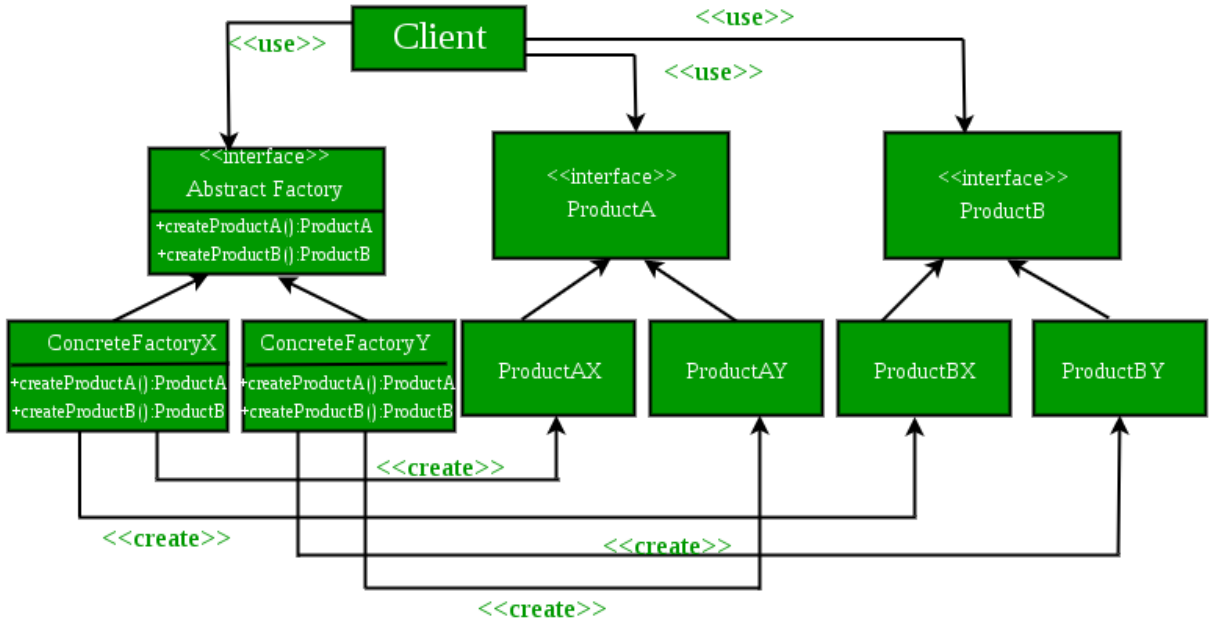


Fig. 1. UML class diagram example for the Abstract Factory Design Pattern

AbstractFactory: Declares an interface for operations that create abstract product objects.

ConcreteFactory: Implements the operations declared in the AbstractFactory to create concrete product objects.

Product: Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.

Client: Uses only interfaces declared by Abstract Factory and AbstractProduct classes.

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 isn't easy.

The abstract factory design pattern can be implemented in both Java and C++ as demonstrated at [26], [28]. 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

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 [29], [30].

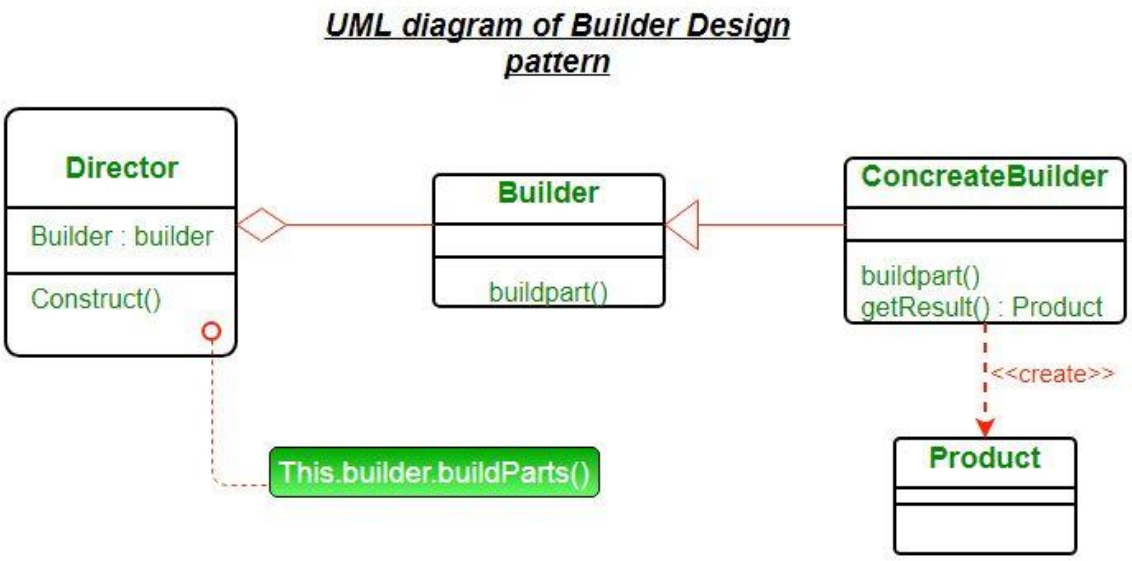


Fig. 2. UML Diagram of Builder Design Pattern

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 [31]. 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++.

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.

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.

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 [18], [19].

Use Case of Structural Design Pattern-

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

Adapter design pattern allows objects with incompatible interfaces to collaborate [32]. 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 [33]. The bridge pattern is used when we need to decouple [2] an abstraction from its implementation so that the two can vary independently [32]. 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. 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. 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. 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. 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.

Private Class Data is used to encapsulate class data and control write access to class attributes as it separates data from methods that us it. The Proxy pattern controls access to the original object [2], allowing you to perform something either before or after the request gets through to the original object. It represents functionality of another class.

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 [18], [19].

Use Case of Behavioral Design Pattern-

1) 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 [13] [34].

Sample implementation of these patterns [35] are available here in both Java and C++.

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.

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 [2] 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.

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 [32]. 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 [32]. It encapsulates how a set of objects interact. 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. 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.

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

Criticism

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

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*** [13].

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.

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 ⅔ of the "jurors" who attended the trial.

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 [13].

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 [13].

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

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

Hahsler [5] 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 of a common design language which developers use to communicate more efficiently. Fig. 3 shows the graph of the usage statistics according the approach of Hahsler [5].

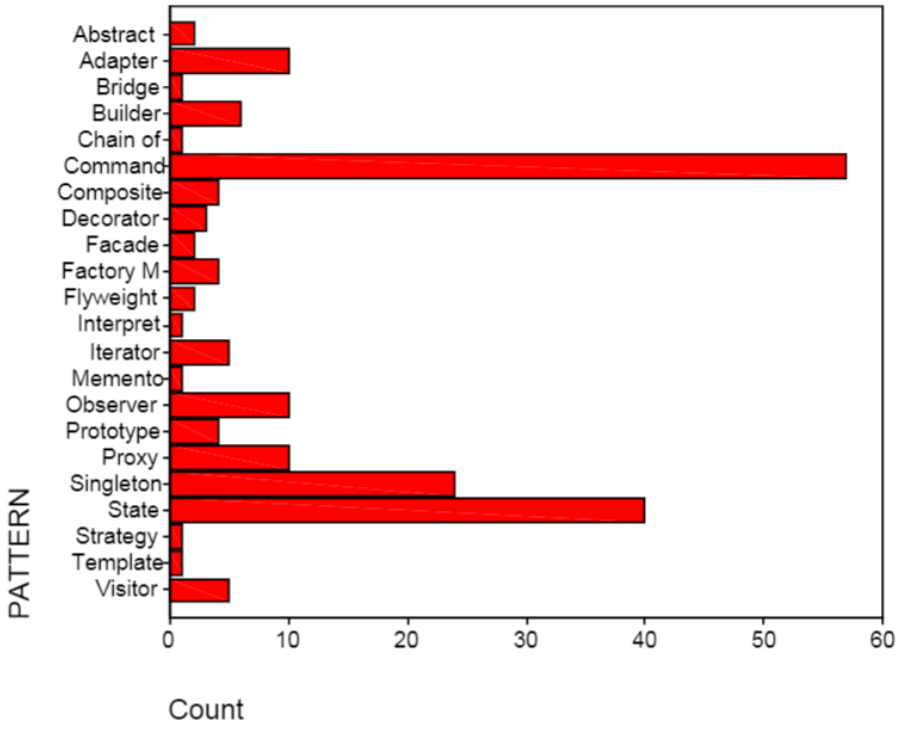


Fig. 3. Number of Java Projects using individual design patterns [5].

Vokac [37] 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.

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

Patterns	Occurrences	%
No Pattern	183 634	77.5 %
Factory	20 237	8.5 %
Singleton	3 331	1.4 %
Observer	16 061	6.8 %
Template Method	5 381	2.3 %
Decorator	1 513	0.6 %
Factory + Observer	612	0.3 %
Factory + Singleton	2 279	1.0 %
Observer + Singleton	2 390	1.0 %
Observer + Template	953	0.4 %
Factory + Observer + Singleton	485	0.2 %

Comparison of Design Patterns in C++ and Java

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	abstraction and implementation to decouple classes into several related hierarchies	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
Behavioural	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.
	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

Implementation of Some Design Patterns in Eolang

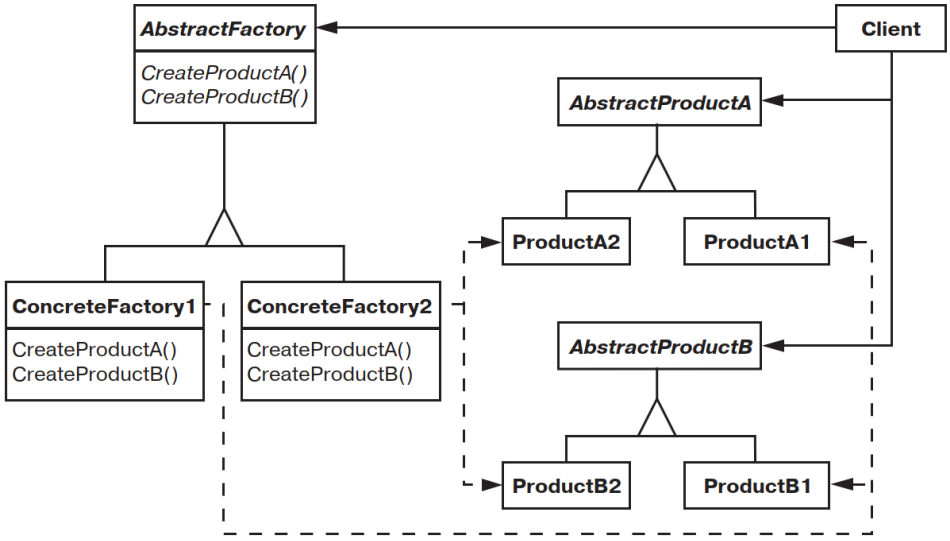
Abstract Factory

An abstract factory is a pattern that generates objects.

Purpose

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

Structure



Participants

AbstractFactory — abstract factory:

• declares an interface for operations that create abstract product objects;

ConcreteFactory — specific factory:

• implements operations that create specific objects-products;

AbstractProduct — abstract product:

• declares the interface for the type of object-product;

ConcreteProduct — specific product:

• defines the product object created by the corresponding particular factory;

• implements the AbstractProductinterface;

Client:

• uses only interfaces that are declared in the AbstractFactory and AbstractProductclasses.

Eolang 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
```

$ ./run.sh 1
ProductA: 1
ProductB: 2
$ ./run.sh 2
ProductA: one
ProductB: two

This program creates objects of integers or strings depending on the argsparameter [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 (whenadding anew implementation, you must make changes to the interface object).

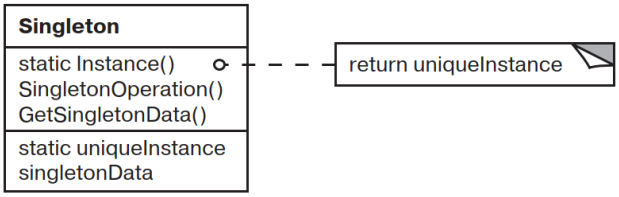
Singleton (singles)

A singleton is a pattern that generates objects

Purpose

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

Structure



Participants

Singleton Singleton:

• Defines an Instanceoperationthat allows clients to access a single instance. Instance is a class operation, that is, a static method of aclass;

• May be responsible for creating your own unique instance.

Relations

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

Eolang 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:

- 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.
- 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.

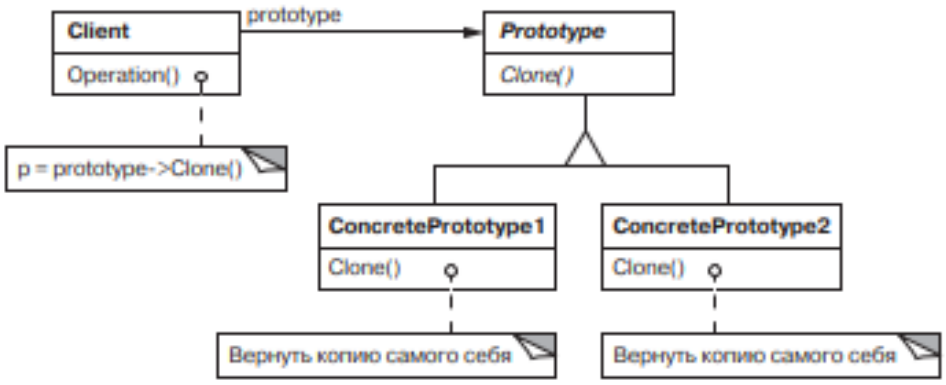
Prototype

A prototype is a pattern that generates objects.

Purpose

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

Structure



Participants

- Prototype — prototype:

• declares an interface for cloning itself;

— ConcretePrototype:

• implements the operation of cloning itself;

— Client — client:

• creates a new object by asking the prototype to clone itself.

Eolang Implementation

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

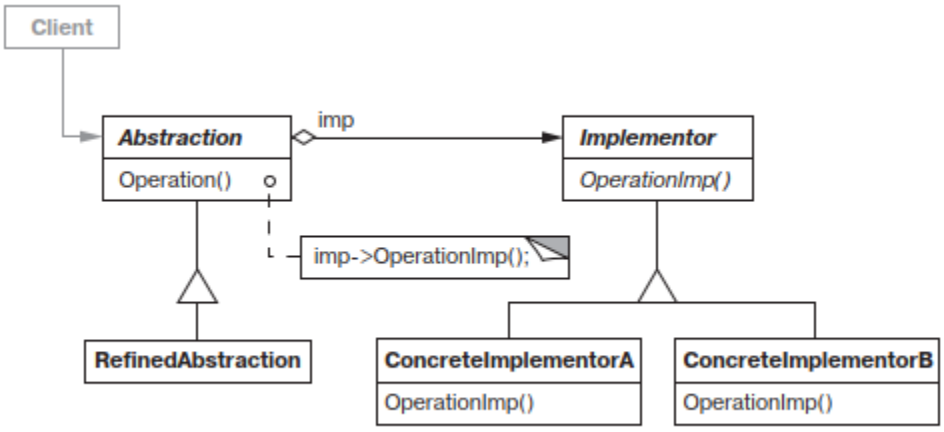
Bridge(BRIDGE)

A bridge is a pattern that structures objects.

Purpose

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

Structure



Participants

Abstraction — abstraction:

• defines the abstraction interface;

• stores a reference to an object of type Implementor;

RefinedAbstraction — refined abstraction:

• extends the interface defined by abstraction abstraction;

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

the Implementor class provides only primitive operations, and the Abstraction class defines higher-level operations based on these primitives;

ConcreteImplementor — specific implementer:

• implements the interface of the Implementor class and defines its specific implementation.

Relations

The Abstraction object redirects client requests to its Implementorobject.

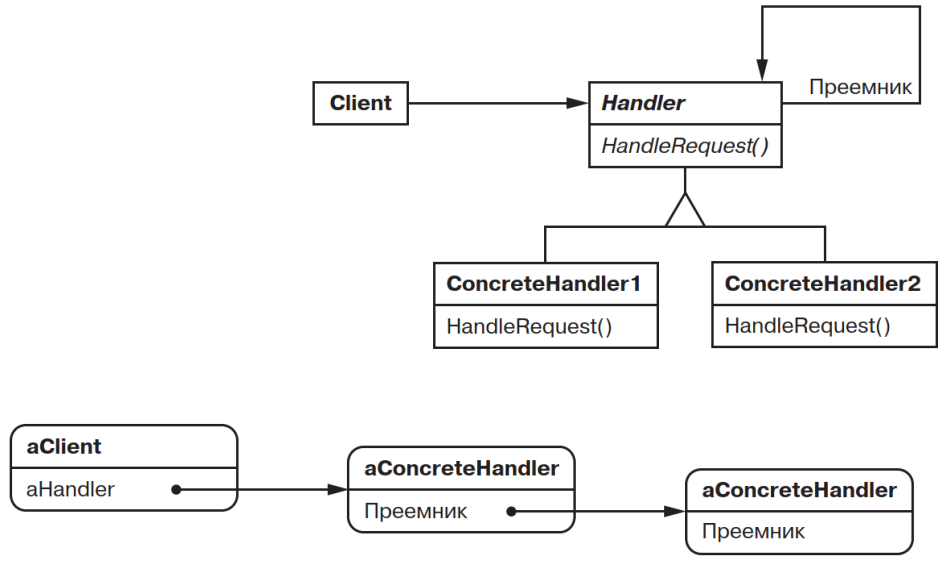
Chain of responsibility

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

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.

Structure



Participants

Handler — handler:

• defines the interface for processing requests;

• (optionally) implements communication with the successor;

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;

Client:

• Sends a request to some ConcreteHandler object in the chain.

Relations

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

Eolang 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"
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).

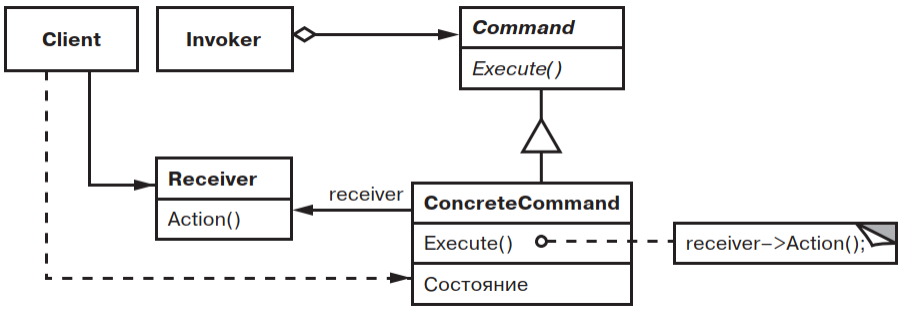
Command

Command — pattern of behavior of objects.

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.

Structure



Participants

- Command — command:

• declares the interface to perform the operation;

- 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;

— Client— client:

• Creates a ConcreteCommand class object and sets its recipient.

- Invoker— initiator:

• calls the command to execute the request;

- Receiver — recipient:

• has information about how to perform the operations necessary to meet the request. Any class can act as a recipient.

Relations

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

- The Invoker initiator saves the ConcreteCommandobject;

- 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;

- The ConcreteCommand object invokes the recipient's operations to execute the request.

Null Object Pattern

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.

EO Implementation

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

output:

run.cmd

null

Decorator design pattern

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.

Eolang Implementation

```
1.  +package sandbox
2.  +alias stdout org.eolang.io.stdout
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 exits in Eolang.

Conclusion

To Do

References

[1] S. M. H. Hasheminejad and S. Jalili, 'Design patterns selection: An automatic two-phase method', *Journal of Systems and Software*, vol. 85, no. 2, pp. 408–424, Feb. 2012, doi: 10.1016/j.jss.2011.08.031.

[2] P. Wendorff, 'Assessment of design patterns during software reengineering: lessons learned from a large commercial project', in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, Mar. 2001, pp. 77–84. doi: 10.1109/CSMR.2001.914971.

[3] D. Riehle and H. Züllighoven, 'Understanding and using patterns in software development', *Theory and Practice of Object Systems*, vol. 2, no. 1, pp. 3–13, 1996, doi: 10.1002/(SICI)1096-9942(1996)2:1<3::AID-TAPO1>3.0.CO;2-#.

[4] D. Streitferdt, C. Heller, and I. Philippow, 'Searching design patterns in source code', in *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Jul. 2005, vol. 2, pp. 33-34 Vol. 1. doi: 10.1109/COMPSAC.2005.135.

[5] M. Hahsler, *A Quantitative Study of the Application of Design Patterns in Java*. 2003.

[6] S. Hussain, J. Keung, and A. A. Khan, 'Software design patterns classification and selection using text categorization approach', *Applied Soft Computing*, vol. 58, pp. 225–244, Sep. 2017, doi: 10.1016/j.asoc.2017.04.043.

[7] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem, 'Object-oriented design patterns recovery', *Journal of Systems and Software*, vol. 59, no. 2, pp. 181–196, Nov. 2001, doi: 10.1016/S0164-1212(01)00061-9.

[8] J. Coplien, 'Software design patterns: common questions and answers', *undefined*, 1998, Accessed: Jul. 28, 2021. [Online]. Available: https://www.semanticscholar.org/paper/Software-design-patterns%3A-common-questions-and-Coplien/9544fccfd09a9315b29ced5bc1e69f572114b7ec

[9] 'Design Patterns: Elements of Reusable Object-Oriented Software [Book]'. https://www.oreilly.com/library/view/design-patterns-elements/0201633612/ (accessed Aug. 02, 2021).

[10] J. Heer and M. Agrawala, 'Software Design Patterns for Information Visualization', *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 853–860, Sep. 2006, doi: 10.1109/TVCG.2006.178.

[11] 'Software Design Patterns - GeeksforGeeks'. https://www.geeksforgeeks.org/software-design-patterns/ (accessed Jul. 23, 2021).

[12] W. Schaffer and A. Zundorf, 'Round-trip engineering with design patterns, UML, java and C++', in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, May 1999, pp. 683–684. doi: 10.1145/302405.302956.

[13] 'Design Patterns and Refactoring'. https://sourcemaking.com (accessed Jul. 23, 2021).

[14] D. C. Schmidt, M. Fayad, and R. E. Johnson, 'Software patterns', *Commun. ACM*, vol. 39, no. 10, pp. 37–39, Oct. 1996, doi: 10.1145/236156.236164.

[15] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, '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*, New York, NY, USA, Sep. 2007, pp. 385–394. doi: 10.1145/1287624.1287680.

[16] W. Pree and H. Sikora, 'Design patterns for object-oriented software development (tutorial)', in *Proceedings of the 19th international conference on Software engineering*, New York, NY, USA, May 1997, pp. 663–664. doi: 10.1145/253228.253810.

[17] C. Zhang and D. Budgen, ‘What Do We Know about the Effectiveness of Software Design Patterns?’, *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1213–1231, Sep. 2012, doi: 10.1109/TSE.2011.79.
[18] P. Kuchana, *Software Architecture Design Patterns in Java*. New York: Auerbach Publications, 2004. doi: 10.1201/9780203496213.
[19] W. Zimmer, ‘Relationships between design patterns’, *undefined*, 1995, Accessed: Jul. 29, 2021. [Online]. Available: https://www.semanticscholar.org/paper/Relationships-between-design-patterns-Zimmer/b7fd68d166ca62fc05fe267b69ac78c279c6ea4f
[20] ‘Factory Method’. https://refactoring.guru/design-patterns/factory-method (accessed Jul. 29, 2021).
[21] ‘Design Patterns | Set 2 (Factory Method)’, *GeeksforGeeks*, Aug. 17, 2015. https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/ (accessed Jul. 29, 2021).
[22] ‘Design Pattern - Factory Pattern - Tutorialspoint’. https://www.tutorialspoint.com/design_pattern/factory_pattern.htm (accessed Jul. 29, 2021).
[23] ‘Design Patterns: Factory Method in C++’. https://refactoring.guru/design-patterns/factory-method/cpp/example (accessed Jul. 29, 2021).
[24] A. Bulajic and S. Jovanovic, ‘An Approach to Reducing Complexity in Abstract Factory Design Pattern’, *Journal of Emerging Trends in Computing and Information Sciences*, vol. 3, no. 10, 2012.
[25] ‘Design Pattern - Abstract Factory Pattern - Tutorialspoint’. https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm (accessed Jul. 29, 2021).
[26] ‘Abstract Factory Pattern’, *GeeksforGeeks*, Jul. 14, 2017. https://www.geeksforgeeks.org/abstract-factory-pattern/ (accessed Jul. 29, 2021).
[27] ‘Abstract Factory Design Pattern’, *Spring Framework Guru*. https://springframework.guru/gang-of-four-design-patterns/abstract-factory-design-pattern/ (accessed Jul. 30, 2021).
[28] ‘Abstract Factory’. https://refactoring.guru/design-patterns/abstract-factory (accessed Jul. 29, 2021).
[29] ‘Design Patterns - Builder Pattern - Tutorialspoint’. https://www.tutorialspoint.com/design_pattern/builder_pattern.htm (accessed Jul. 30, 2021).
[30] ‘Builder Design Pattern’, *GeeksforGeeks*, Jul. 25, 2017. https://www.geeksforgeeks.org/builder-design-pattern/ (accessed Jul. 30, 2021).
[31] ‘Singleton Design Pattern in Java’, *Spring Framework Guru*. https://springframework.guru/gang-of-four-design-patterns/singleton-design-pattern/ (accessed Jul. 30, 2021).
[32] W. F. Tichy, ‘A catalogue of general-purpose software design patterns’, in *Proceedings of TOOLS USA 97. International Conference on Technology of Object Oriented Systems and Languages*, Aug. 1997, pp. 330–339. doi: 10.1109/TOOLS.1997.654742.
[33] ‘Design Patterns - Adapter Pattern - Tutorialspoint’. https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm (accessed Aug. 01, 2021).
[34] ‘Design Patterns | Set 1 (Introduction)’, *GeeksforGeeks*, Aug. 06, 2015. https://www.geeksforgeeks.org/design-patterns-set-1-introduction/ (accessed Jul. 23, 2021).
[35] ‘Huston Design Patterns’. http://www.vincehuston.org/dp/ (accessed Jul. 26, 2021).

[36] S. Khwaja and M. Alshayeb, ‘Survey On Software Design-Pattern Specification Languages’, *ACM Comput. Surv.*, vol. 49, no. 1, p. 21:1-21:35, Jun. 2016, doi: 10.1145/2926966.
[37] M. Vokac, ‘Defect Frequency and Design Patterns: An Empirical Study of Industrial Code’, *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 904–917, Dec. 2004, doi: 10.1109/TSE.2004.99.