World of Computer Science and Information Technology Journal (WCSIT)
ISSN: 2221-0741
Vol. 6, No. 2, 12-20, 2016

The General Form of GoF Design Patterns

Siniša Vlajić
Department of software engineering
Faculty of organisational sciences, University of Belgrade
Belgrade, Serbia

Vojislav Stanojević, Dušan Savić, Miloš Milić,
Ilija Antović, Saša Lazarević
Department of software engineering
Faculty of organisational sciences, University of Belgrade
Belgrade, Serbia

Abstract— In this paper, we present a general form of *GoF Design Patterns* as a process that transforms the *BDPSP* (the Basic Design Pattern Structure of the Problem) to the *BDPSS* (the Basic Design Pattern Structure of the Solution), i.e. transforms the unstable structure of the program to the stable structure. The stability of the *BDPSP* and *BDPSS* is explained by using *Robert C. Martin's Stability metric*. The *BDPSP* and *BDPSS* are described by three elements: *Client*, *Abstract Server* and *Concrete Server*. The *Client* is an element of the pattern structure that uses the functionalities of the abstract server and concrete servers in order to carry out its own functionality. The *Abstract Server* is an element of the pattern structure that provides the client with an abstract functionalities that can be implemented with various specific functionalities. The *Concrete Server* is an element of the pattern structure that provide the client with concrete functionalities that realise abstract functionalities of the abstract server. In this sense, the *Design Pattern* is a process that relationship *Client has Concrete Server*, transform to relationships: a) *Client has Abstract Server* and b) *Concrete server is Abstract Server*. We believe the BDPSS is the key mechanism or essence of the GoF design patterns, which allow easy maintenance and upgrade of the program. We have showed that in 20 of 23 GoF design patterns the BDPSS completely describes a pattern or a particular part of the pattern. We are using the general form of GoF design patterns in the teaching process, as the first step in the overall understanding of the GoF design patterns, before a detailed explanation of the specific characteristics of GoF design patterns. We think that this paper can greatly help the students and developers to quickly and clearly understand the essence of GoF design patterns.

Keywords- Software design; Design patterns; Stability metric; Software stability; Software maintenance.

I. INTRODUCTION

The development of the modern object-oriented software systems involves the use of the **Design Patterns** [DP], that represent the best experiences and practices in the software design, that were discovered by numerous software developers over a long period of time. One of the basic definition of the design patterns is [DP]: "The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". They represent generic solutions that can be applied multiple times for different classes of problems. The design patterns are one of the most important mechanisms used for the development of the stable software systems because they provide great flexibility during the maintenance and upgrade of the software systems. The **software stability**, according to ISO 9126 [ISO1] "characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes". From perspective of Yau and Collofello software stability is resistance to propagation of changes (**ripple effect**) that the software would have when it is modified ([YAU1], [YAU2]), which is also known as **modular continuity** [MEY1]. E. Mohamed Fayad points the importance of identifying the areas in the project that are stable [MF1], during the development of a software project: This yields a stable core design and, thus, a stable software product. Changes introduced to the software project will then be in the periphery, since the core was based on something that remains and will remain stable". When talking about stability, Robert C. Martin stressed the significance of the dependencies that exist between components and classes for the stability of a software system. He has said [RM2]: "Depend in the direction of stability" and suggests that: "Any component that we expect to be volatile should not be depended on by a component that is difficult to change! Otherwise, the volatile component will also be difficult to change ". He has introduced the **Stable-Dependencies Principle (SDP)** to ensure that "modules that are intended to be easy to change are not depended on by modules that are more difficult to change than they are ". In our previous studies we have tried to define a formal basis for making the stable and sustainable software systems, explaining the design patterns by the symmetry concepts[VS2]. The starting point of this study was Rosen's definition of the symmetry [ROS1]: "Symmetry is immunity to a possible change". We also tried to explain the relationship that exists between the GoF design patterns and software entropy in the context of the software maintenance [VS1].

The software system (in a broader sense) or the software component (in a narrow sense) is being created on the basis of the user requirements. The first version of the software usually contains mixed a general and specific parts of the code. The general parts of the code can be used not only to solve current problems, but also to solve some other problems. They can be reused in the development of new software systems (code reuse). Code reuse is defined as: "*... use of existing software, or software knowledge, to build new software, following the reusability principles*" [FRAKES1]. Specific parts of the code are related to the current problem and they can't be used for other problems. They don't have *reuse* feature.

When software is being developed over the time, each new version of the software tends to separate general and specific parts of the programming code. Ideally, software system should contain only general parts of the code. In that case, the specific parts of the code could be "relocated" from the program and somewhere kept as program parameters. This way, a parameterized software system can be customized to different problems, by changing the values of the parameters of the software system for each new specific problem. M. Stark said [STARK1]: "*Parameterized software system is one that can be configured by selecting generalized models and providing specific parameter values to fit those models into a general design*."

The software system should be created so that it can be easily adapted to each new user requirement. Such software system is able to accept new or change the existing functionalities without major changes to its structure and behavior.

Design Patterns provide separation of the general and specific parts of a software system, allowing the use of the general programming codes for some set of similar problems, i.e. a class of the problems.

When writing a program it is necessary to identify general and specific parts of the program, i.e. changeable and unchangeable parts of the program. If we recognize the places in the software system, which is constantly changing with the advent of new user requirements, then the pattern should be applied on those places to stem potential "chaos" that can be caused by the changeable places in the program.

These places, or these points, are by analogy equal to **bifurcation point** in the *Chaos Theory*. Steven Strogatz for bifurcation said the following [SS1]: " *The bifurcation is a qualitative change in the dynamics of a system as a parameter is varied. The value of the parameter at which the change occurs is called the bifurcation value, also sometimes known as the bifurcation point or point of bifurcation"*. The bifurcation point push system in chaos, because it changes equilibrium points of the system, i.e. the stability of the system.

Something similar can happen in the development of the software systems. If timely, the bifurcation points are not discovered, the orthogonal complexities could happen, i.e. the system could be orthogonally developed (at more places) in regard to the existing software system. This orthogonal complexity can completely "crash" the existing software system. This could be compared to "whirlpools" that pull the program into "the depth of nothingness".

Applying patterns on these points prevents the software system turns into chaos. The developer must have the knowledge, experience and intuition to recognize these "bifurcation points" during development of the software system. At these points the changes continuously occur. These points grow with time and become huge and difficult to maintain. Identifying "bifurcation point" accelerates establishing order in the system, allowing easiness in use and maintenance of the system.

"Spaghetti code", a term known from software engineering, which indicates the complex and complicated algorithm of some method, which grows with new user requirements, represents a potential case of "bifurcation point" of the software systems.

In this paper, we have analysed the **GoF (Gang of Four)** design patterns and found the structure of the solution that is common to most of the GoF design patterns. This has led us to hypothesize that this structure is a **key mechanism** of the GoF design patterns, which should be applied to the places where the bifurcation points occur. This structure of the solution contains "attractor" which introduces order in the software system.

Based on the previously mentioned structure of the solution, we have defined the structure of the problem. At the end, we have explained how the structure of the problem is transformed to the structure of the solution. This approach enables logical relationship between elements of the structure of the problem and solution of the GoF design patterns, allowing us to explain, in the general sense, ***what are design patterns and when and how they arise***.

II. THE GOF DESIGN PATTERNS – DESIGN PROBLEM AND SOLUTION

The GoF defines design patterns [DP] as: "The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". The definition points out the main parts of the GoF design patterns: problem and solution. **The problem**, describes when the pattern should be applyed and specific design problem that causes an inflexible design. **The solution** describes the elements of the design, their roles and relationships and explains how a general arrangement of elements solves design problem.

That means, the design patterns recognise a design problem and gives an appropriate design solution for it. Actually, the GoF design patterns transform (T) the design problem to the design solution (Figure 1).

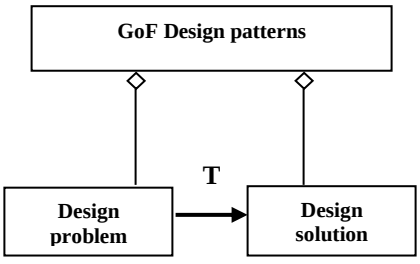


Figure 1: The GoF Design pattern – Design problem and solution

The GoF design patterns contains: a) Design problem, b) Design solution and c) Transformation (T) of the design problem to the design solution, which can be represented as:

Design problem --------T--> Design solution

III. THE BASIC DESIGN PATTERN STRUCTURE OF THE SOLUTION

In the book *Design Patterns: Elements of Reusable Object-Oriented Software* [DP], there are **23 GoF design patterns**. They are divided into three groups: creational patterns, structural patterns and behavioral patterns. We have discovered the structure that exist in 20 of these 23 patterns. This structure eighter completely describes a pattern or some particular part of the pattern. It is **key mechanism or essence** of GoF design patterns. We called (Figure 4) this structure: **"The Basic Design Pattern Structure of the Solution (BDPSS)** ". This structure is presented on figure 2:

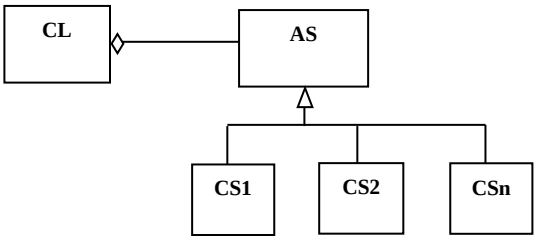


Figure 2: The Basic Design Patern Structure of the Solution

The **BDPSS** is a 3-tuple (***Client(Cl), Abstract Server (AS) and Concrete Server (CS))***, where can be an arbitrary number (***n***) of the concrete servers. The client can be normal or abstract class, abstract server can be interface, normal or abstract class, while concrete server can be normal class. Below we show the definitions of the elements of BDPSS:

The ***Client*** is an element of the pattern structure that uses the functionalities of the abstract server and concrete servers in order to carry out its own functionality.

The ***Abstract Server*** is an element of the pattern structure that provides the client with an abstract functionalities that can be implemented with various specific functionalities.

The ***Concrete Server*** is an element of the pattern structure that provide the client with concrete functionalities that realise abstract functionalities of the abstract server.

The Client and concrete servers are connected to abstract server. Between client and abstract server exist **aggregation, composition** or **dependency** UML relationship. Mentioned relationships are described by the term **"has"**. Between concrete servers and abstract server exist **inheritance** or **realization** UML relationship. Mentioned relationships are decribed by the term "**is**". BDPSS can be represented as follows:

(Cl has AS) and (CS_1 is AS) and (CS_2 is AS) and ... and (CS_n is AS)

In the general sense, we can describe it on the folowing way:

(Cl has AS) and (CS is AS), where can be an arbitrary number (***n***) of the concrete servers.

The client and concrete servers are dependent on abstract server. Between the client and concrete servers there is not dependence. The Abstract server in the BDPSS represents **"atractor"** of the software system.

If we look BDPSS as some object-oriented program:

```
interface AS { void request(); }

class CS1 implements AS { void request() { System.out.println("CS1
performs client's request!"); } }

class CS2 implements AS { void request() { System.out.println("CS2
performs client's request!"); } }
...
class CSn implements AS { void request() { System.out.println("CSn
performs client's request!"); } }

class CL
  { AS as;
    void makeConnection(AS as1)  {as = as1;}
    void request() {as.request();}
  }

class Main
  {   public static void main(String arg[])
       {    CS1 cs1 = new CS1();
            CS2 cs2 = new CS2();
            ...
            CS2 csn = new CSn();

            CL cl = new CL();
            if (arg[0].equals("cs1"))
               cl.makeConnection(cs1);
            if (arg[0].equals("cs2")
                 cl.makeConnection(cs2);
            ...
            if (arg[0].equals("csn")
                 cl.makeConnection(csn);
           cl.request();
       }
  }
```

we can say that, at compile time, the client (CL) knows that will be connected with some object of classes (CS1, C22, ..., CSn) that implement abstract server (AS), but in that moment, it does not know the exact object (cs1or cs2,... or cs_n). The Client makes connection, at run time, with concrete object. That means, at run time will be resolved what concrete server will realize the client's request. The relationship *Client has AS* gives the program flexibility during its maintenance and upgrade, because a client's request can be realized in many different ways by using different concrete servers. This is known as dynamic or late binding, that lets the mutual subtitution of the objects, that have same interface, at run-time. *This substitutability is known as polymorphism* [DP], which is one of the key concepts in object-oriented design. The advantage of BDPSS is reflected in the case, when we want to add new concrete server CS_{n+1} to BDPSS,

class **CSn+1** implements AS { void request() { System.out.println("CS**n+1** performs client's request!"); }}

then we don't change the client, because the client is not related to concrete servers than with the abstract server. Note that GoF said [DP]: "*Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change*". The BDPSS lets that some aspect of system structure (*Concrete servers*) vary independently of other aspects (*Client*). Also, BDPSS is in compliance to one of the most important principle of reusable object-oriented design [DP]: *Program to an interface, not an implementation.* The Client depends on abstract server (*interface*) instead of concrete servers *(implementation).*

Wholly – ***more BDPSS:*** More BDPSS are presented wholly in the structure of the solution of the design patterns (4 design patterns).

Partly – ***one BDPSS:*** One BDPSS is presented partly in the structure of the solution of the design patterns (7 design patterns).

Partly – ***more BDPSS:*** More BDPSS are presented partly in the structure of the solution of the design patterns (3 design patterns).

There is 5 categories BDPSS presence in GoF design patterns:

Wholly – ***one BDPSS***: One BDPSS is presented wholly in the structure of the solution of the design patterns (6 design patterns).

There isn't *BDPSS*: The BDPSS isn't presented in the structure of the solution of the design patterns (3 design patterns).

Table 1 shows presence of the BDPSS in the GOF design patterns. The Client cannot be seen directly in *Factory Method* and *Template Method* patterns, but it practically exists when the pattern is used. Once again we emphasize, the BDPSS occurs in **20 GOF design patterns**.

TABLE 1: THE PRESENCE OF THE BDPSS IN GOF DESIGN PATTERNS

GOF design patterns	***Client***	***Abstract Server***	***Concrete Server***	***Category presence of BDPSS***
1. Abstract Factory	Client	AbstractFactory	ConcreteFactory	Wholly – more BDPSS
	Client	AbstractProductA	ProductA	
	Client	AbstractProductB	ProductB	
2. Builder	Director	Builder	ConcreteBuilder	Wholly – one BDPSS
3. Factory Method	Creator	Product	ConcreteProduct	Partly – one BDPSS
4. Prototype	Client	Prototype	ConcretePrototype	Wholly – one BDPSS
5. Singleton	Singleton			There isn't BDPSS
6. Adapter	Client	Target	Adapter	Partly – one BDPSS
7. Bridge	Client	Abstraction	RefinedAbstarction	Wholly – more BDPSS
	Abstraction	Implementor	ConcreteImplementor	
8. Composite	Client	Component	Leaf или Composite	Wholly – more BDPSS
	Composite	Component	Leaf или Composite	
9. Decorator	Decorator	Component	ConcreteComponent или Decorator	Partly – one BDPSS
10. Façade	main	Façade		There isn't BDPSS
11. Flyweight	FlyweightFactory	Flyweight	ConcreteFlyweighti или UnsharedConcreteFlyweight	Partly – one BDPSS
12. Proxy	Client	Subject	Proxy или RealSubject	Wholly – one BDPSS
13. Chain of Responsibility	Client	Handler	ConcreteHandler	Wholly – more BDPSS
	Handler	Handler	ConcreteHandler	
14. Command	Invoker	Command	ConcreteCommand	Partly – one BDPSS
15. Interpreter	Client	AbstractExpression	TerminalExpression или NonterminalExpression	Partly – more BDPSS
	NonterminalExpression	AbstractExpression	TerminalExpression или NonterminalExpression	
16. Iterator	Client	Aggregate	ConcreteAggregate	Partly – more BDPSS
	Client	Iterator	ConcreteIterator	
17. Mediator	Colleague	Mediator	ConcreteMediator	Partly – one BDPSS
18. Memento	CareTaker	Memento		There isn't BDPSS
19.Observer	Subject	Observer	ConcreteObserver	Partly – one BDPSS
20. State	Context	State	ConcreteState	Wholly – one BDPSS
21. Strategy	Context	Strategy	ConcreteStrategy	Wholly – one BDPSS
22. Template Method	Client	AbstractClass	ConcreteClass	Wholly – one BDPSS
23. Visitor	Client	Visitor	ConcreteVisitor	Partly – more BDPSS
	ObjectStructure	Element	ConcreteElementi	

Below we will give one example for each category of the BDPSS presence in GoF design patterns.

a) Wholly–one BDPSS category – Builder pattern

At the Builder pattern there is one BDPSS (Figure 3):

Client = Director, AbstractServer = Builder and *Concrete Sever = ConcreteBuilder.*

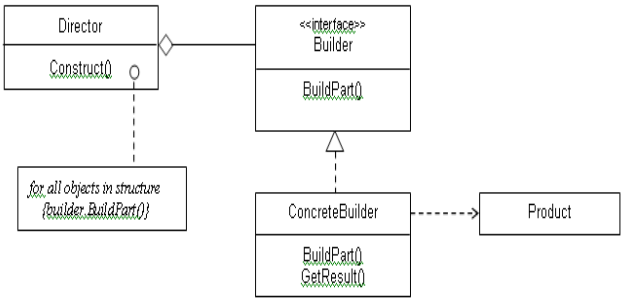


Figure 3: Buider pattern

b) Wholly–more BDPSS category – Bridge pattern

At the Bridge pattern there are two *BDPSS* (Figure 4):

Client = Client, AbstractServer = Abstraction and ConcreteSever = RefinedAbstraction

Client = Abstraction, AbstractServer = Implementor and ConcreteSevers = ConcreteImplementor.

ConcreteImplementor are *ConcreteImplementorA* and *ConcreteImplementorB.*

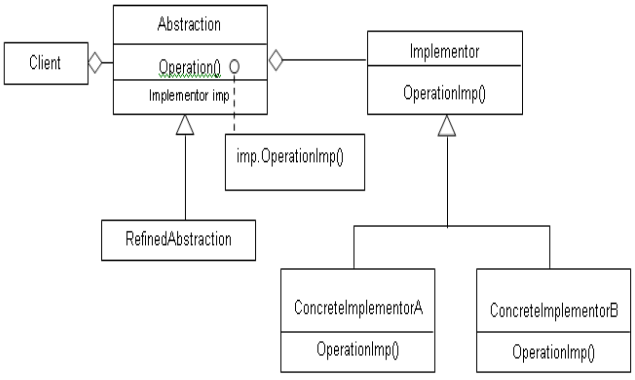


Figure 4: Bridge pattern

c) *Partly–one* BDPSS category – Mediator pattern

At the Mediator pattern there is one BDPSS (Figure 5):

Client = Colleague, AbstractServer = Mediator and *Concrete Sever = ConcreteMediator.*

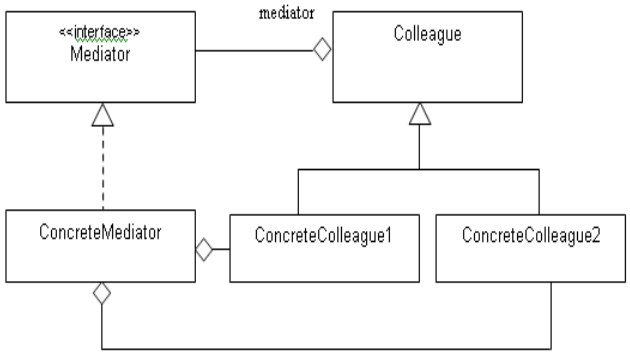


Figure 5: Mediator pattern

d) **Partly –more BDPSS category – Interpreter pattern**

At the Interpreter pattern there are two *BDPSS* (Figure 6):

Client = Client, AbstractServer = AbstractExpression and ConcreteSever = ConcreteExpression

Client = NoterminalExpression, AbstractServer = AbstractExpression and ConcreteSever = ConcreteExpression.

ConcreteExpression are *TerminalExpression* and *NoterminalExpression.*

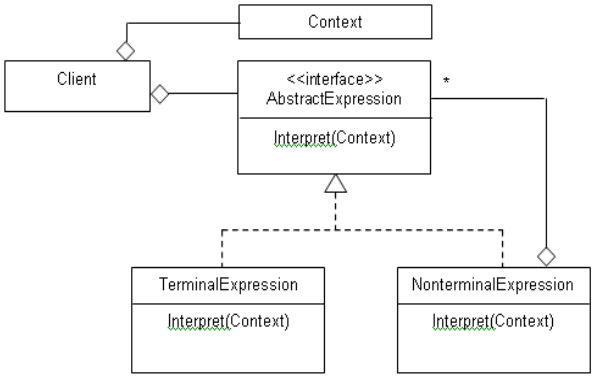


Figure 6: Interpreter pattern

e) **There isn't BDPSS category – Memento pattern**

At the Memento pattern there isn't the BDPSS (Figure 7).

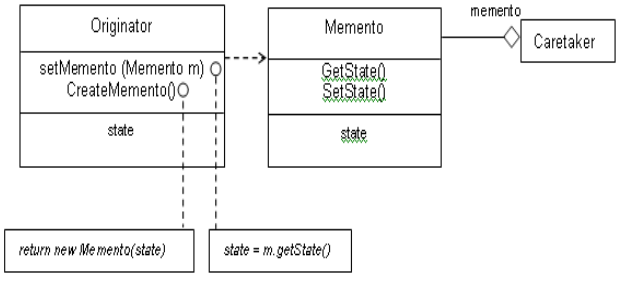


Figure 7: Memento pattern

IV. THE BASIC DESIGN PATTERN STRUCTURE OF THE PROBLEM

The BDPSS is the result of the transformation of the structure of the problem of the GOF design patterns, in which the client is directly connected to the concrete servers (*Figure 8*). We called this structure: **"The Basic Design Pattern Structure of the Problem (BDPSP)"**.

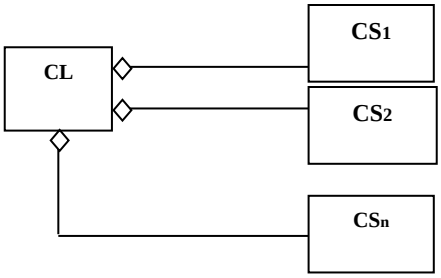


Figure 8: The Basic Design Pattern Structure of the Problem

The **BDPSP** is 2-tuple (***Client (Cl), Concrete Server$_n$(CS$_n$)***), where can be an arbitrary number (***n***) of the concrete servers. Below we show the definitions of the elements of BDPSP:

The **Client** is an element of the pattern structure that uses the functionalities of the concrete servers in order to carry out its own functionality. The Client is connected to many different concrete servers and dependent of them.

The **Concrete Server** is an element of the pattern structure that provides the client with concrete functionalities.

Between the client and the concrete server can be **aggregation, composition** or **dependency UML** relationship. The BDPSS is difficult to maintain because of the changeability of the client (its structure) when new concrete server CS_{n+1} is added to the BDPSP. The Client in BDPSP represents **"bifurcation point"** of the software system. *The problem of the BDPSP is the strong dependence between the client and concrete servers.* The BDPSP can be represented as follows:

(Cl has CS$_1$)and(Cl has CS$_2$) and ... and (Cl has CS$_n$)

In the general sense, we can describe it on the folowing way:

Cl has CS, where can be an arbitrary number (***n***) of the concrete servers.

From the perspective of the object-oriented program, client during compile time make connection to a concrete servers.

```
class CL
  { CS1 cs1; Cs2 cs2;   ...  CSn csn; ...
    CL( ) { cs1 = new CS1(); cs2 = new CS2();   ...
            csn= new CSn(); ... }
     void request (int cs)
        {
           if (cs==1) {cs1.request();}
           if (cs==2) {cs2.request();}
           ...
           if (cs==n) {csn.request();}
        }
    ... }
```

```
class CS1 { void request() { System.out.println("CS1 performs client's request!"); } }
class CS2 { void request() { System.out.println("CS2 performs client's request!"); } }
    ...
class CSn { void request() { System.out.println("CSn performs client's request!"); } }

class Main
      {  public static void main(String arg[])
          {   CL cl = new CL();
              int cs;
              if (arg[0].equals("cs1"))
                 cs = 1;
              if (arg[0].equals("cs2")
                 cs = 2;
              ...
              if (arg[0].equals("csn")
                  cs = n;
              cl.request(cs);
          }
   }
```

This connection remains for all the time of the program execution and *it disables the flexibility of the program during its execution*.

In the book [DP] is said that: *"The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly. To design the system so that it's **robust to such changes**, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and reimplementation, **client modification**, and retesting"*.

The **BDPSP** is example of a structure that isn't robust on changes, where the Client must be modified every time when the new concrete server is added to the software system:

```
class CSn+1 { void request() {
                System.out.println("CSn+1 performs client's request!"); } }
class CL
    { CS1 cs1; Cs2 cs2;   ...  CSn csn; ...
      CSn+1 csn+1;  // change of the Client
      CL( ) { cs1 = new CS1(); cs2 = new CS2();  ...
              csn= new CSn(); ... }
       void request (int cs)
          {  if (cs==1) {cs1.request();}
             if (cs==2) {cs2.request();}
             ...
             if (cs==n) {csn.request();}
             if (cs==n+1) {csn+1.request();}// change of the Client
          }
       ... }
```

This structure (BDPSP) is cause of formation BDPSS.

V. THE GENERAL FORM OF GOF DESIGN PATTERNS

The general form of GoF design patterns could be presented as follows (*Figure 9*):

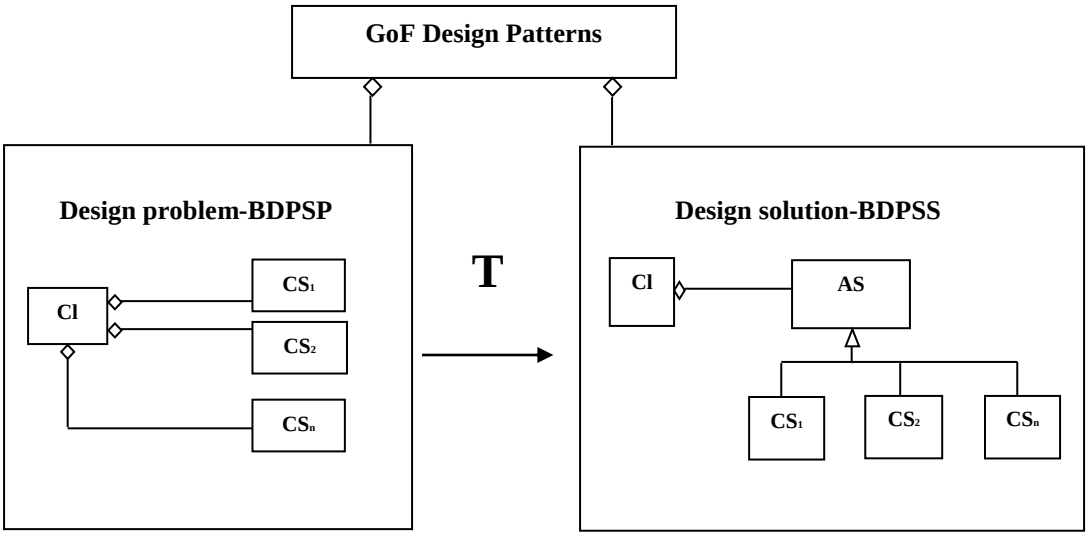


Figure 9: The general form of GOF Design Patterns

The general form of GoF design patterns contains:

a) The Basic Design Pattern Structure of the Problem (BDPSP), b) The Basic Design Pattern Structure of the Solution (BDPSS) and c) The Transformation (T) of BDPSP to BDPSS, which can be represented as:

T

BDPSP --------> BDPSS

The transformation ***T*** of the BDPSP to the BDPSS, indicates to the transformation of the 2-tuple (Client, Concrete server) to 3-tuple (Client, Abstract server, Concrete server), on the folowing way:

Cl has CS -> (Cl has AS) and (CS is AS)

The transformation T occurs when new user requirements often happens and increases the number of new concrete servers in the BDPSP. Then the client frequent changes its structure and the BDPSP with changeable client is difficult to maintain and upgrade. The BDPSP has **"unstable"** structure.

This can be demonstrated by using the *Robert C. Martin's Stability metric* [RM1, RM2]:

I = Ce/(Ca+Ce),

where is I mark for **instability.** *Ca is* afferent couplings, i.e. the number of classes outside this component that depend on classes within this component. *Ce is* efferent couplings, i.e. the number of classes inside this component that depend on classes outside this component. Stability metric has the range [0,1].

if I = 0 a component has a maximally stability. If I = 1 a component has a maximally instability.

If we assume that each class of the BDPSP is in a separate component, and if we examine the stability of the class Client, from which depend m classes, and that depends on n classes (the number of the Concrete servers), then is Ce = n and Ca = m. if we assume m = 1 and n = 3 then the instability is:

I = n/m+n => I = 3/3+1 => I = 0.75

That means that the Client class is unstable 0.75. If n increases, then the expression n / (n + 1) tends to 1 and maximum instability. We can conclude, the instability of the class Client will grow with the increasing number of the concrete servers.

On the other hand a program that has the BDPSS in which number of the concrete servers does not affect the client. The BDPSS has "**stable**" structure.

If the stability of the Client class of the BDPSS is tested , when m = 1 (as in the previous examples), then is Ce = 1 a Ca = 1.

In this case instability is:

I = n/m+n => I = 1/1+1 => I = 0.5

Here is the Client class unstable 0.5 which is less than the Client class of the BDPSP. With the increasing number of the concrete servers the Client class does not change the stability, which means that the stability of the client is independent of the number of the concrete servers.

It means the patterns by the transformation T provides the mechanism that the unstable structure transforms to stable structure. The unstable structure has shorter life cycle than stable structure, because the unstable structure relatively

quickly lead a program to chaotic state which is very difficult to maintain.

Based on the above analysis we can give our definitions of design pattern:

Definition 1: ***The Design Pattern is a process that transforms the BDPSP to the BDPSS, i.e. transforms the unstable structure of the program to the stable structure.***

Definition 2: ***The Design Pattern is a process that relationship Client has Concrete Server, transform to relationships: a) Client has Abstract Server and b) Concrete Server is Abstract Server.***

VI. CONCLUSION

At the beginning of this paper, we have explained that the GoF design patterns, in the most general sense, represent the transformation of the problem to the solution. Then we pointed to the structure of the solution that is common to most of GoF Design Patterns. We called (Figure 4) this structure: "The Basic Design Pattern Structure of the Solution (BDPSS)". We believe that this structure represents a **key mechanism, essence or property** of the GoF design patterns, which allows easy maintenance and upgrade of the program.

The BDPSS lets some aspect of the system structure (*Concrete servers*) varies independently of other aspects (*Client*). After that, we have mentioned 5 categories the BDPSS presence in GoF design patterns. For each of these categories we gave one example. We showed that in 20 of 23 GoF design patterns the BDPSS completely describes the pattern or some particular part of the pattern, as shown in Table 1. The problem of GoF design patterns is explained by the structure that we called "The Basic Design Pattern Structure of the Problem (BDPSP)". The BDPSP is example a structure that isn't robust on changes. This structure is changeable (*Client*) when the new functionality is added to the software system and it is the cause of formation of the BDPSS. The main part of the paper considers the general form of the GoF design patterns, i.e. explains how the elements of the BDPSP are transformed to the elements of the BDPSS. Justification of the transformation is explained by using the Stability metric. Based on the above considerations are derived 2 definitions of the GoF design patterns:

Definition 1: ***The Design Pattern is a process that transforms BDPSP to BDPSS, i.e. transforms the unstable structure to the stable structure.***

Definition 2: ***The Design Pattern is a process that relationship Client has Concrete Server, transform to relationships: a) Client has Abstract Server and b) Concrete Server is Abstract server.***

At the end, we gave two examples of using general form of GoF design patterns. We believe that this paper can greatly help all those who want to understand, in general sense, how the structure of the problem is transformed to the structure of the solution of the GoF design patterns, i.e. how from the unstable structure of the program becomes the stable structure of the program which is invariant to changes.

This paper is the result of many years of teaching experience that we have had in working with our students on Department for Software Engineering, Faculty of Organizational Sciences, University of Belgrade. We are using the general form of GoF design patterns in the teaching process, as the first step in the overall understanding of GoF design patterns, before a detailed explanation of the specific characteristics of GoF design patterns. We think that this paper can greatly help the students and developers to quickly and clearly understand the essence of GoF design patterns.

REFERENCES

[1] [DP] Gamma E, Helm R, Johnson R, Vlissides J.: Design Patterns. Addison-Wesley: Reading MA, 1995.

[2] [ISO1] ISO 9126 "Information Technology: Software product evaluation, quality characteristics and guidelines for their use", International Organisation for Standardization, Geneva, 1992.

[3] [MEY1] B. Meyer: "Object-Oriented Software Construction", 2nd Edition, Prentice Hall, Englewood Cliffs, New Jersey, USA, 1997.

[4] [YAU1] S. Yau and J. Collofello:"Some stability measures for software maintenance," Transactions on Software Engineering, IEEE Computer Society, 6 (6), pp. 545-552, November 1980.

[5] [YAU2] S. Yau and J. Collofello, "Design stability measures for software maintenance," Transactions on Software Engineering, IEEE Computer Society, 11 (9), pp. 849-856, September 1985.

[6] [6] [MF1] Mohamed E. Fayad and Adam Altman: "Thinking objectively An Introduction to Software Stability", Communications of the ACM, Volume 44, Issue 9, Spet. 2001.

[7] [RM1] Robert . C. Martin , "Design Principles and Design patterns", www.objectmentor.com.

[8] [RM2] Robert . C. Martin, Micah Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall,

[9] ISBN: 978-0-13-185725-4, 2006.

[10] [AL1] Christopher Alexander,Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel: A Pattern Language. Oxford University Press, New York, 1977.

[11] [COP1] James O. Coplien: Software Patterns, Bell Labs, The Hillside Group, 2000.

[12] [ROS1] Joe Rosen: Symmetry rules – how science and nature are founded on symmetry, Springer-Verlag Berlin Heidelberg, 2008, ISBN 978-3-540-75972-0.

[13] [VS1] Vlajic Sinisa: Explanation of Software Entropy by Golden Ratio and Logarithmic Spiral, Proceedings of the 2007 International Conference on Software Engineering Theory and Practice (SETP-07) , Pages 12-20, ISBN: 978-0-9727412-6-2, July 9-12, 2007, Orlando, Florida, USA.

[14] [VS2] Siniša Vlajić, Dušan Savić, Ilija Antović: The Explanation of the Design Patterns by the Symmetry Concepts, The Fourteenth IASTED International Symposium on Artificial Intelligence and Soft Computing (ASC 2011), Proceedings of the IASTED International Conference on Signal and Image Processing and Application, Pages: 363-372, ISBN: 978-0-88986-885-4, June 22 – 24, 2011, Crete, Greece.

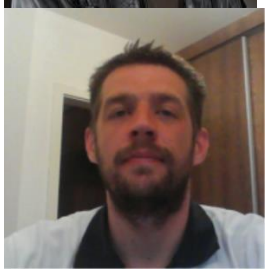
[15] [STARK1] M. Stark, "On Designing Parameterized Systems Using Ada", Proceedings of the Seventh Washington Ada Symposium, June 1990

[16] [FRAKES 1] Frakes, W.B. and Kyo Kang: "Software Reuse Research: Status and Future", IEEE Transactions on Software Engineering, 31(7), July, pp. 529-536., 2005.

[17] [SS1] Steven H.Strogatz: Nonlinear Dynamics and Chaos, with Applications to Physics, Biology, Chemistry and Engineering, Addison-Wesley, 1994.

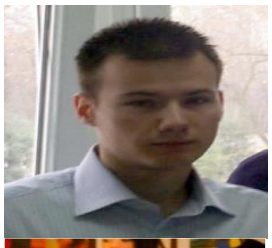
AUTHORS PROFILE

PhD Siniša Vlajić, is an associate professor of software engineering at University of Belgrade, Facult y of Organizational Sciences, Department of Information Systems. He has taught undergraduate and graduate level courses: introduction to programming, introduction to information system, software design, software patterns, programming methodology and Java programming language. He wrote many books, scripts and publications about C++, Java, software design, software patterns, database and information systems. His main research interests include: software process, software design, software maintenance, software pattern formalization and programming methodology. He is one of the founders of the Laboratory and Department of the Software Engineering at Faculty of Organizational Sciences.



Stanojević Vojislav, is an teaching assistant of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems. He has taught undergraduate and graduate level courses: introduction to programming, introduction to information system, software design, software patterns, programming methodology and Java programming language. He wrote publications about Java, software design, software patterns, application frameworks and domain specific languages. His main research interests include: software design, application frameworks, business rules, domain specific languages.

Dušan Savić received the Magistar degree in Information system and technologies from the Faculty of Organization Sciences, University of Belgrade, in 2010. He is currently postgraduate student and teaching assistant on Faculty of Organizational Sciences at the Software Engineering Department. He has interests in the following areas: Modeling and Meta-modeling, Model Driven Engineering, Requirement Engineering, Software Development, Software Design, Domain Specific Languages, Automation of User Interface development. He has taught undergraduate and graduate level courses in his area. He is the author or co-author of several publications on national and international conference and workshop and journal papers.



Miloš Milić is teaching assistant at Faculty of Organizational Sciences, University of Belgrade, Serbia. He has taught undergraduate and graduate level courses: introduction to programming, software design, software patterns and Java programming language. His research interests include Software Quality, Software Design and Software Testing. He holds BSc in Information Systems and MSc in Software Engineering. He is PhD student at University of Belgrade

PhD Ilija Antović is an assistant professor of software engineering at University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems. His research interests are: Automation of User Interface development, Modeling and Meta-modeling, Model Driven Engineering, Requirements Engineering, Software patterns, Code Generation. He lectures at undergraduate and graduate level courses in his area. He is the author or co-author of several publications on national and international conferences and journal papers.

PhD Saša D. Lazarević, is an associate professor at the University of Belgrade, Faculty of Organizational Sciences, Department of Software Engineering. He has taught undergraduate and graduate level courses: Introduction to programming, Software design, Software construction, Software testing, Software quality, Database systems. His main research interests include: software process, software design, software testing, software quality, universe of database systems and software construction on .NET platform. He is co-founder of the Department of Software Engineering at FOS, UB.