# OBJECT ORIENTED MODELING FOR THE INDUSTRIAL APPLICATIONS

**T. Bekasiewicz, M. Gil, K. Szustakiewicz, K. Szustakiewicz**

Warsaw University of Technology
Department of Mechanical Engineering Design Fundamentals
e-mail: maciej.gil@hotmail.com, tomasz.bekasiewicz@fackelmann.pl,
k6.jim@wp.pl, jjezik6@wp.pl

**Keywords:** CAD, object-oriented modeling, design patterns

**Abstract**: *In this article we will describe our experiences in design of CAD software application. We have started from procedural concept of the application. Second attempt utilized object-oriented paradigm. Migration to this modern approach was rather difficult on its early steps, but afterwards it emerged very successful and we have picked all its benefits out. Object oriented modeling and design promoted us better understanding of the industrial design process supported by our application, lead us to cleaner software code as well as more maintainable system.*

## INTRODUCTION

In yr 2004 we have established good relationships and cooperation with our external industrial partner POLIMEX -MOSTOSTAL S.A., who is the second biggest civil engineering company in polish market. Our first joined project started in one of its departments - Zakład Krat Pomostowych (Platform Gratings Dept), where they design and construct grids, gratings, and wire meshes for platforms, light bridges, and stairs. They asked us to design and implement a dedicated CAD application for spiral stairs design support. Mostostal's previous efforts in this area and cooperation with other professional software houses were not satisfactory. Their legacy, dedicated solutions based on AutoCAD extensions were comfortless but first of all they do not speed up the design process of spiral stairs as expected [6].

After several meetings and discussion about Mostostal's requirements we have decided to take on this task. We of had to elaborate conceptual model of the system as well as provide ready application supporting customer needs. We have started with the team of five master theses –students, two consultants from our faculty and two of Mostostal Platform Gratings department. The first section will describe our first achievements.

The first project was completed and was accepted by our customer as successful. Mostostal was encouraged for further cooperation and we have decided to rewrite our original application in .net technology to add new features, to ease and enhance some design steps, add some functionality. Our system had to optimize stairs configuration, create all design documentation as well as technology documentation. This forced us to change the software development approach. Another point to consider was our team shrink to two students, two consultants with Mostostal part limited to only checkpoint control. With the second project we have started with object oriented approach. In the time this text is written we have bypassed all milestones of it but still some work to follow. Second part of this article describes our new architecture.

## 1. PROCEDURAL APPROACH

The first release of the application was developed by the team of 5 students working cooperatively on different aspects of the system [2,3,6]. The main target of their work was to design and provide complete system, giving to the user the ease and speed during CAD design process of spiral stairs. The system has enabled rapid prototyping of staircase configuration- number of stairs (with their key parameters, angle and level), platforms, balusters, etc. and automated generation of 2D and 3D technical documentation. Primary customer's requirement was to create easy and comfortable in use computer system that accelerates time consuming calculations and also provides error-free drawings of spiral stairs.

Due to customer's software policy requirements there was a limitation to use AutoCAD and/or MS Office tools. The system was then developed in AutoCAD VBA. Because we have the team of five students working in parallel on different parts of the system, we have established central data exchange

structure, which was implemented in MS Excel, also with its VBA extensions. Each student, the developer, was taking valuable data from the excel repository, then his aim was to develop a piece of code for dedicated part of staircase and afterwards, to put the results and some control data- back to the excel worksheet. We have then independent software modules dedicated for stair flights, for railings, for wall fasteners and CAD drawings. The excel worksheet took the role of the blackboard [4], from the blackboard architecture approach. Whilst the major tasks of MS Excel's worksheet was to store data and execute all necessary calculations, AutoCAD's VBA, performed all code related to graphical user interface as well as drawing generation. Figure 1 presents the concept of the first release application [3,6].

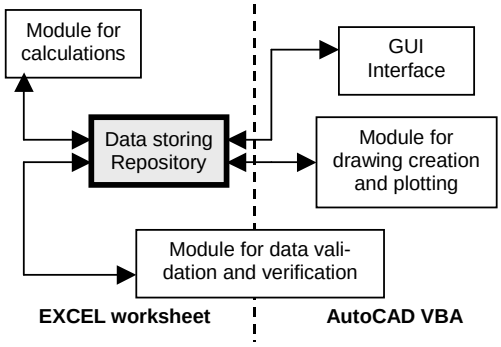


Fig.1. *Basic Application Architecture*

Figure 2 presents user interaction with the developed application. The basic scenario was as follows. First, user enters the entry data. Then he starts several calculations and afterwards, the effect (staircase layout) is being presented on the screen. If there are any errors or layout is not satisfactory then the designer needs to correct or change some data and start repeat calculations again. Finally, if the calculated data are acceptable, drawing module is initiated and CAD documentation is being generated.

The application was quite straightforward in the concept but it had some drawbacks. There was only one fixed algorithm that enabled user to go through the whole stairs layout design process. Any modification or update required repeating the process from the very beginning. Although the calculations were far quicker than legacy manual procedures, but any update in staircase layout required to restart the design process. This solution was not convenient for the user because in order to receive required values of output parameters, it required predicting, which input parameters should be modified and how. Moreover this inconvenience caused the application time-consuming in usage. Additionally, procedural programming in VBA caused some extra performance degradation.

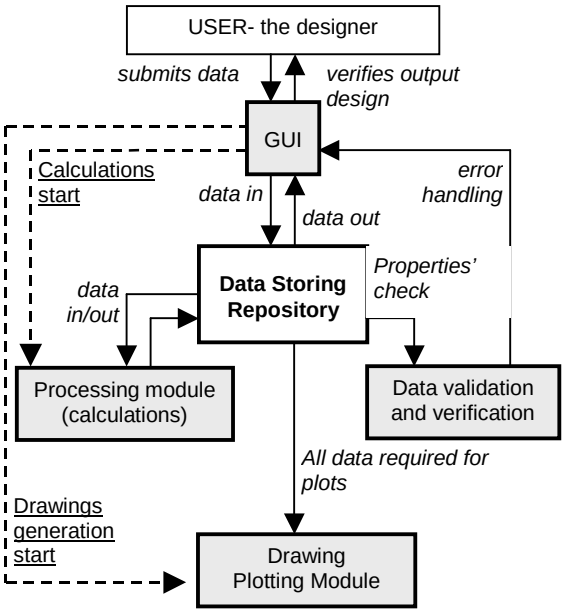


Fig.2. *User Interaction with the system*

Very long and complicated calculation and drawing subroutines with numerous code loops are one of the main reasons why the code was executing slowly and afterwards, is was also difficult to maintain.

Mostostal accepted the first release, as the application worked reliable and produced error-free documentation, although, the customer has identified all above disadvantages. To meet customer's growing needs and requirements we have decided to improve our application and add some new functionality that couldn't be done in the legacy, procedural approach. We have decided to redesign the system and implement it in .NET, object-oriented technology.

## 2. OBJECT ORIENTED APPROACH

The customer recognized the first implementation. The application fulfilled its main requirements – it has accelerated laborious calculations, stair steps calculations and afterwards it generated AutoCAD documentation. Although it had some limitations- the code was closed, difficult to maintain and extend. Any change in the future customer requirement, like change in the interoperability with next release of AutoCAD or addition of another step stair type required substantial coding. The code itself was not clear and practically required only its authors for reading and updating [6].

These weaknesses brought down the decision to begin next project on the reengineered version of the application. After several meetings we have decided to implement this application in object-oriented, .NET technology. We have started with the new project. Its code name was 'LaScala'. We have decided to utilize modern, object-oriented approach based on .NET. The next sections describe the software alternatives we utilize in the reengineered version of the application. We have found the notion

of the design patterns is applicable in the area of CAD software development.

During design process of the applications we meet with the similar problems in different parts of the designed applications. The same, suitable solutions emerged the in many places of the application. We could apply the solutions to these issues using the notion of design patterns [1,12,15].

The design patterns describe a commonly recurring design problem that occurs in a particular context and based on a set of guiding force or recommend a solution to it [1]. The solution is usually a simple mechanism, set of interrelations between two or more classes, objects, services, processes, threads, components, or nodes that work together to solve a general design problem within that particular context. We perceive design patterns as set of considerations built on the top of the object-oriented programming but they are the part of this technology. Design patterns provided us and helped to describe solutions to specific problems we met during software design. The design pattern is not a ready to use solution [14]. It is a template, mini-architecture, set of guidelines how to solve a problem that can be used in many different occasions. The design patterns are not algorithms because they don't solve the computational problems rather than design problems. Using the design patterns we have speed up the development process by providing those already tested, proven development paradigms. But the most valuable benefit was the design patterns made our own designs more flexible, modular, reusable, and understandable.

We can classify the design patterns based on multiple criteria, the most common of which is the basic underlying problem they solve. During years of research and practice in software development many design patterns has emerged and has been elaborated, the most common are:

- Fundamental patterns: delegation pattern, functional design, interface pattern, proxy pattern, and immutable pattern.
- Creational patterns: abstract factory pattern, anonymous subroutine objects pattern, builder pattern, factory method pattern, lazy initialization pattern, prototype pattern, and singleton pattern.
- Structural patterns: adapter pattern, bridge pattern, composite pattern, container pattern, decorator pattern, extensibility pattern, façade pattern, flyweight pattern, proxy pattern, pipes and filters, private class data pattern.
- Behavioral patterns: chain of responsibility pattern, command pattern, event listener, interpreter pattern, iterator pattern, mediator pattern, memento pattern, observer pattern, state pattern, strategy pattern, template method pattern, visitor pattern, single-serving visitor pattern, hierarchical visitor pattern.
- Concurrency patterns: active object, balking pattern, double checked locking pattern, guarded suspension, leaders/followers pattern, monitor object, read write lock pattern, scheduler pattern, thread pool pattern, thread-specific storage.
- Architectural patterns: model-view-controller pattern, presentation-abstraction-control pattern, and client-server pattern.

Software factories is a paradigm for automating software development that integrates component based and model driven development [5,7,12], software architecture, aspect oriented programming, and requirements, process and software product line engineering to increase agility, productivity and predictability across the software life cycle.

## 2.1. Model-view-controller pattern

The model-view-controller [1,14] pattern (named also MVC) is a software architectural pattern that separates an application's data model of the domain, the graphic user interface representation, and the actions based on user input (control logic) into three separate classes so that modifications to one class can be made with minimal impact to the others:

- **Model.** The model manages the behavior and domain-specific representation of the information, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).
- **View.** The view renders the domain-specific model into a form suitable for interaction. The view is typically a graphic user interface element. The view manages the display of information.
- **Controller.** The controller responds to events from the model and/or from the view. The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

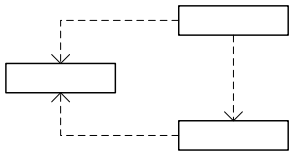

Fig.3. *Model-view-controller pattern class structure*

Model-view-controller, shown in figure 3, is one of the fundamental software architectural design patterns. Its main purpose is the separation of graphical user interface logic from the application's business logic.

### 2.1.1. Main MVC Application Models

In our application there are actually two project models that are corresponding to the MVC paradigm. The first one we called *Stairway Computation Model* and the second one *Stairway Detailed Model*. Both of the models consist of group

of objects represented in the programming code by classes, which are organized using encapsulation, inheritance and polymorphism.

The main object of the newly developed system is the `StairCaseApplication`, which represents our application after run. It contains collection of document objects – projects of stairs that are created. Going further each model has its own structure of objects.

As far as *Detailed Model* is concerned it's a full representation of all real components that can exist in the spiral stairway structure, for example steps, newels, but also brackets, bushes and even bolts. The *Computation Model* is not so sophisticated. It has only objects, which are necessary for copulation of the stairway layout, i.e. three-dimensional configuration of stairs, such as tiers, platforms and steps. Classes represent all stairway objects and each class has its members referred to properties of objects and methods - actions that can be taken on them. According to the MVC, all the objects and their properties constitute the full model of data and processing, but at the same time they are completely independent from any user interface (View) or controller part.

The View part of the MVC for main models is located in the main application form. It is MDI form, so it can contain set of child forms. Child forms are representation of `document` objects. We have deployed the concept of showing different kinds of graphical presentations of stairway models on these forms, e.g. 3D view, 'lemon' layout and others. After choosing a document object, the hierarchical structures of staircase models objects are drawn in the tree view controls, which are also placed in the main form. The main window contains two property grids as well – they are to present particular properties of the model objects (fig.4).

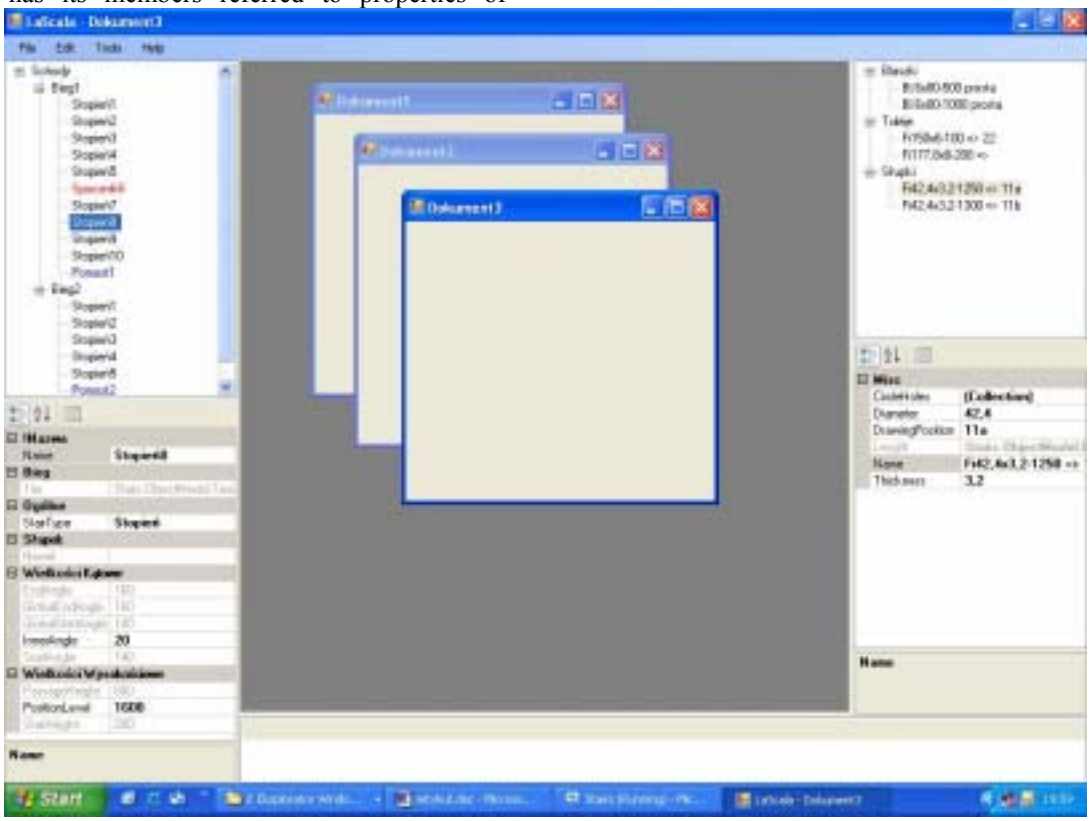

Fig.4. *LaScala Main Window*

As far as Controller is concerned, it is nearly impossible to point one, single place where it is implemented. The Controller in MVC consists of the set of events and methods spread across the application code. These events trigger on users' actions. Controller's methods action the Models methods or change the View itself. For example `AddStair_MenuItem_Click` event handler responds to appropriate click event and guides to the `StairCollection Add` method in the Model causing its changes by adding new `StairItem`.

### 2.1.2. Three-dimensional layout MVC

There is also another, independent MVC model used for specific calculations supporting stairway three-dimensional layout. It contains collection of `result` objects, where each object represents different result in stairway layout computations. This project has its own view - graphical user interface that presents collection of result objects from the calculation process. Names of the result objects are presented in the tree view control. After choosing a result its properties are shown in the property grid control and the graphical representation is drawn in the proprietary developed graphical control (fig.5.)

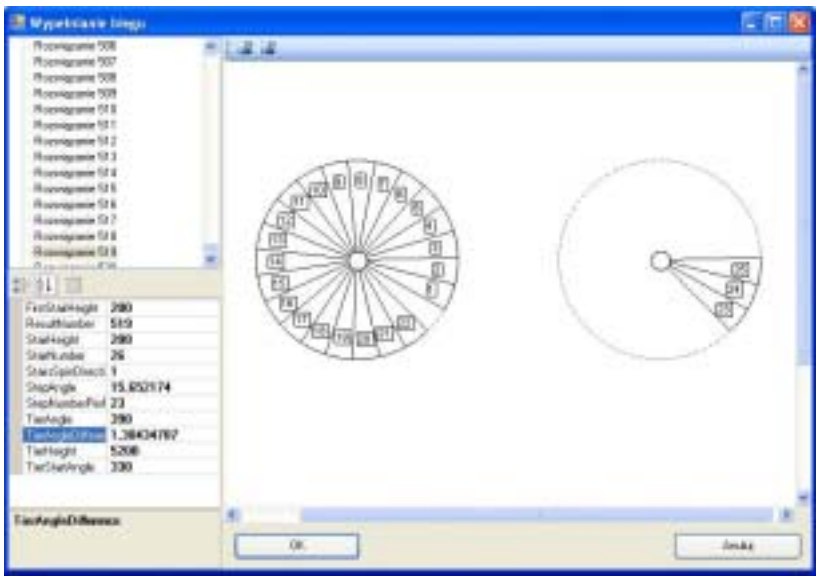

Fig.5. *Dialog for three-dimensional layout optimization*

## 2.2. Command pattern

The command pattern is a software behavioral pattern in which objects are used to represent actions [12]. Command pattern encapsulates a command request as an object, letting developers to parameterize clients with different requests, queue or log requests. Command pattern enables supporting undoable operations keeping a history stack of the recently executed commands. If the user wants to undo command, the program executes the most recent command object's undo() method.

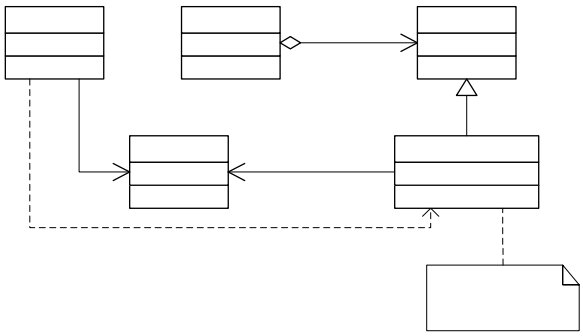

Fig. 6. *Command pattern class structure*

Command pattern enables also to prepare macro recording (macro command). If all actions are represented by command object then the program can keep a list of command objects as they are executed. Program can execute the same command objects again in sequence.

The classes and/or objects participating in this pattern are:

- Command: declares an interface for executing an operation.
- ConcreteCommand: defines a binding between a Receiver object and an action, implements Execute method by invoking the corresponding operation(s) on Receiver.
- Client: creates a ConcreteCommand object and sets its receiver.
- Invoker: asks the command to carry out the request.
- Receiver: knows how to perform the operations associated with carrying out the request.

Currently in our application, the Command Pattern paradigm is mainly used to support operations like adding or removing objects within collections. Our main form – `frmApplication` (Client) has a set of methods like 'Add…' and 'Remove…' where new instances of appropriate 'Add…' and 'Remove…' commands (ConcreteCommands) are created and then executed by suitable instance of the Invoker class. Suitable instance of the invoker class is meant here as that invoker which is a private field of activate document (project in our application). Because in our case of `StairCaseApplication` user can work on multiple projects (documents) at the same time, we have decided to provide each of them with it's own invoker. Each invoker, except of invoking commands (method `ExecuteCommand`), keeps also a history of commands executed when working with specified project. This feature gave us straightforward way to implement Undoing and Redoing functionality of last executed commands. Every instance of the invoker class has `Undo` and `Redo` methods that executes or reverse-executes particular command from the history stack of executed commands.

This history is stored in an array type variable in every instance of the invoker class, so user can work on multiple documents, switching between them and the history of completed actions is always remembered separately for each project. Current command is always kept in a private variable in the invoker so the application knows which command should be undone or redone from the commands history.

Depending on the particular command, different instances of the Receivers can be passed to the command as its parameter. As long as we are

concerning adding or/and removing objects in collection, the Receiver is the collection itself or the object which contains this collection. Receiver is passed to the ConcreteCommand and Receiver's Add or Remove action is then executed via ConcreteCommand's `Execute` method.

There is a special concept of rearranging implementation of Command Pattern in our application. We are considering representation of all simple activities by basic commands and more complicated activities by so called macro commands, which are certain sequences of basic commands. We can see this makes our code clearer, better organized and thus easier to maintain. Additionally, undoing and redoing should work more smoothly as we enable undoing and redoing of every single action that can be performed in the application's run time and not only adding and removing items in collections.

## 2.3. Abstract factory pattern

Abstract factory pattern is a software creational pattern, which provides an interface for creating families of related or dependent objects without specifying their concrete classes [12,15]. This pattern enables separating the details of implementation of a set of objects from its general usage.

The classes and/or objects participating in this pattern are:

- AbstractFactory: declares an interface for operations that create abstract products.
- ConcreateFactory: implements the operations to create concrete product objects.
- AbstractProduct: declares an interface for a type of product object.
- Product: defines a product object to be created by the corresponding concrete factory, implements the AbstractProduct interface.
- Client: uses interfaces declared by classes: AbstractFactory and AbstractProdut.

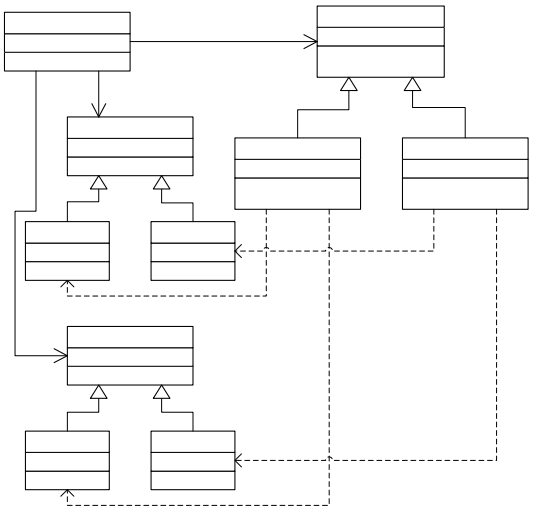

Fig. 7. *General abstract factory pattern class structure*

Use of this pattern made us possible to interchange concrete classes without changing the code that uses them. To be more specific, one of the sample places it is used in our application is to separate the details of implementation of different kinds of CAD engines we are going to use to generate technical documentation of stairs. It will also allow us to add another CAD engines in the future without any substantial change inside the code.

There is one, main, project that contains Abstract Factory – `DrawingEngineFactory` and Abstract Product – `IDrawingEngine`.

`IDrawingEngine` constitutes interface for a type of graphical CAD engine object. There are only declarations of subroutines referenced to all CAD engines, which draw particular graphic objects such as line, arc, etc. and also `stairs` object like blade, bush, newel etc. `DrawingEngineFactory` declares an interface for creating different kinds of CAD graphical engines.

Furthermore, there are also other projects with classes, which implement the `IDrawingEngine` interface. They are in fact implement subroutines declared there. These classes represent Concrete Products such as `AutoCADDrawingEngine`, `CatiaDrawingEngine` and many others we can decide to join with separate projects.

Our main form, `frmApplication` is the Client and uses interfaces declared in the classes named `DrawingEngineFactory` and `IDrawingEngine`.

We have chosen the design slightly different that classical abstract factory pattern. One detail that is missing from standard factory pattern is misuse of ConcreteFactory classes. Instead of this, the client can use specific CAD engine through an abstract `DrawingEngineFactory`. This class consists of read-only property of `IDrawingEngine` type – `DrawingEngine`, which returns appropriate concrete drawing engine instance. It is possible because we store the information the designated configuration file. The information stored is: Assembly file and Type of object. Assembly file is the full path to the library file (`.dll`) of the specific CAD engine. Type of object is simply the name of the class, from which specific object is going to be created. It depends on user personal settings, which drawing engine will be in use.

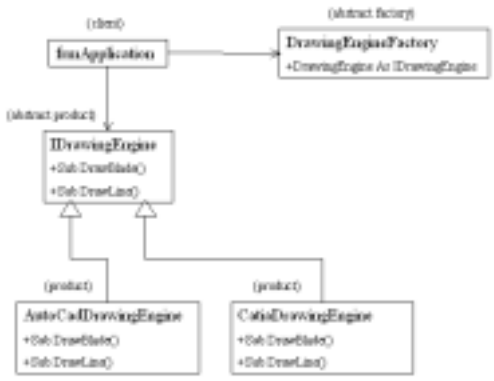

Fig. 8. *General abstract factory pattern class structure*

If we want to execute drawing subroutine from the Client code directly, we have to create the new `DrawingEngineFactory` object with specific `CADDrawingEngine` stored in its property and then call appropriate subroutine of the drawing engine, for example

```
Dim factory As New _
  Engines.DrawingEngineFactory _
  factory.DrawingEngine.DrawBlade(blade)
```

The biggest advantage of using this software pattern is that we do not need to change any of the Client code when choosing, switching to or developing different drawing engine. The only required action is to update settings in the configuration file.

## 2.4. Singleton pattern

Singleton pattern is a software creational pattern that ensures a class only has one instance, and provides a global point of access to it [15]. Singleton pattern is useful when exactly one object is needed to coordinate actions across the system. This pattern is implemented by creating a class with a method that creates a new instance of the object if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made either private or protected.

The classes and/or objects participating in this pattern are (fig.4):

- Singleton (Load Balancer): defines an Instance operation that lets clients access its unique instance. Instance is a class operation, responsible for creating and maintaining its own unique instance.

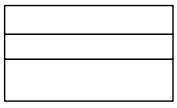

Fig. 9. *Singleton pattern class structure*

In our application we use singleton pattern to provide ability of creating only one instance of `StairCaseApplication` class. This class allows us to create object that stores all data about staircase projects during application runtime. The singleton pattern ensures that only one instance of application class can be created, which means that only one executable of the LaScala program can be loaded into memory.

Design patterns have helped us in the way we was thinking about, designing and implementing object-oriented application. We have found them applicable in many stages of this project development. They gave us new level of abstraction for system design. They provided us with a common vocabulary to communicate, explore and discuss various design alternatives

## 3. .NET Technology

Microsoft's .NET is the new programming model for building desktop, mobile, and Web-based applications [9,10,11]. This is very important feature. With .NET we are not limited to build one-tier, closed application. This is in fact complete programming environment giving us the flexibility – we can build desktop application for engineers designing staircases, with possible infinite number of interfaces for example database storage of staircase design cases or staircase design alternatives, web-services for design presentation to the management or Mostostal's customer. It was possible because we have embedded the system with .NET architecture. We would like here to refer to the .NET architecture, which is three things: a library of unified core classes that provide the core for applications, presentation classes for developing web and desktop (Windows) applications, the Common Language Runtime (CLR) [10,13], an environment in which .NET programs are executed.

In .NET, code is compiled twice - first into Intermediate Language - by the compiler on development machine. Then again - at runtime, when the code is executed by the CLR. This has very positive outcomes: regardless of the language in which the source code is written, whether it's C#, Visual Basic .NET or C++, it's compiled into the same intermediate language and this intermediate language is distributed to the end user. Thanks to this we could write one, consistent application, where main design logic is written in Visual Basic, but other code, like 3D Graphics is written in C#.

## 4. SUMMARY

We would like to emphasize that that object-oriented technology is more than just away of programming or organizing application code. It applies certain techniques to the entire software development lifecycle. This is the way thinking of and modeling real life tasks.

During the development of this project we have discovered use of the notion of design patterns very

useful, especially when dealing with the complexity of the CAD system with its formal model, user interface and calculations running in background (application logic). Design patterns represent the most frequently used communication structures of programs. This makes them reusable or better to say applicable in many different fields. We have it proven very useful when developing customized CAD application. By separating different aspects of the objected oriented model of the application – according to each design paradigm guidelines- each of these aspects can be developed, modified and maintained independently.

When using design patterns we were actually possible to simplify complex problems by making the encountered problems more general and by treating them at higher level of abstraction. These patterns also could be tested in parallel on different structures, we could develop complete software components, utilize and test the functionality of one software pattern, and afterwards – transfer its concepts by deploying it into our LaScala project. These well-tested components generated by using design patterns guides let us lower the possibility of making errors and saved time in development and testing.

## 5. FUTURE WORK

At the time of writing this document most parts of the new, redesigned system works fine and we put our focus on the components displaying 3D graphics (DirectX) and generation AutoCAD documentation (DXF).

After we complete all tasks related to this project we plan to focus on creating a generic CAD framework where we will offer consistent architecture with complete library of classes available for rapid development of specialized CAD tools. The architecture will be derived from LaScala project. This framework will be like *meta- software pattern* consisting of special configuration of patterns we have been using in our current project and described in this article.

**References**

[1] Trowbridge D., Mancini D., Quick D., G. Hohpe, Newkirk J., Lavigne D.. *Enterprise Solution Patterns Using Microsoft .NET*. Microsoft Corporation, 2003.

[2] Jarosz A.: *Budowa generatora modeli geometrycznych schodów przemysłowych dla Mostostal Siedlce – Stężenia*. IPBM, Warsaw University of Technology, master thesis, 2005.

[3] Chyliński Ł., Pruszyński J.: *Budowa gene-ratora modeli geometrycznych schodów przemysłowych dla Mostostal Siedlce – Barierki i Pomosty*. IPBM, master thesis, Warsaw University of Technology, 2005.

[4] Gil M.: *Proces projektowania w budowie maszyn z zastosowaniem narzędzi kompu-terowych do integracji jego elementów*. IPBM, Warsaw University of Technology, PhD thesis, WPW, 2001.

[5] Ferguson J., Patterson B., Beres J., P. Boutquin, Gupta M.. *C# Bible*. Wiley Publishing, Inc., 2002.

[6] Szustakiewicz K. Szustakiewicz K..: *Budowa generatora modeli geometrycznych schodów przemysłowych dla Mostostal Siedlce – Schody Spiralne*. IPBM, Warsaw University of Technology, master thesis, 2005.

[7] Hamilton J.. *Object-Oriented Programming with Visual Basic .NET*. O'Reilly, October 2002.

[8] Brown E.. *Windows Forms Programming with C#*. Manning Publications Co., 2002.

[9] Wakefield C., Sonder H.-E., Wei Meng Lee. *VB.NET Developer's Gui*de. Syngress Publishing, Inc., 2001.

[10] Grundgeiger D.. *Programming Visual Basic .NET*. O'Reilly, January 2002.

[11] Riel, Arthur J. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[12] Spencer K., Eberhard T., Alexander J.. OOP: Building *Reusable Components with Microsoft Visual Basic .NET*. Microsoft Press, November 9, 2002.

[13] Reynolds-Haertle R. A.. *OOP with Microsoft Visual Basic .NET and Microsoft Visual C# Step by Step*. Microsoft Press, 2002.

[14] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[15] Gamma, Helm, Johnson, and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.