ALGEBRA DESCRIBING SOFTWARE STATIC MODELS

Elena Chebanyuk

Abstract: *Algebra, describing software static models that are represented as the UML-diagrams of classes packages and components is presented in this paper. The major constituents of the algebra, which rests upon set-theory tools, are classes, components, packages, abstract classes and interfaces. The operations defined on these constituents are: inheritance, polymorphism, composition, aggregation and association. The algebra describing software static models allows: precise a description of design patterns structural constituents; design formats of files for saving software static models; provide a mathematical apparatus for formalized description of interconnections between class diagram elements; propose code generation tools while analyzing functional requirements to application and solve other tasks, requiring analysis of interconnections between constituents of class diagram.*

An analysis of the problem domain – "designing dense layings of cutting schemas" is represented in this paper. Two subdomains, namely "designing dense layings for leather goods details" and "designing dense layings for shoe details" are considered. The purpose of this analysis is describing framework of the chosen problem domain in terms of software static model description algebra. Components of the "abstract factory" design pattern have been selected to create interconnections between the classes in the problem domain. A diagram of classes of the problem domain that is designed according to analytical description tis represented.

Keywords: *class diagram; design pattern; set-theory tool; software modeling; abstract factory design pattern; dense laying designing.*

ACM Classification Keywords: D.2.2 [Design Tools and Techniques]; D.2.11 [Software Architectures]

Introduction

Software static models are represented as UML-diagrams are artifacts that are used in any technology of software development. Development of approaches for analytical description constituents of software static models and interconnections between them will increase the effectiveness of different issues that are solved in software life cycle processes. Such issues are: saving information about class diagrams and interconnections between their constituents, identification of design patterns structural components in existing code, elaboration such analytical aspects as model driven architecture (MDA) and model driven development (MDD), description of class functionality and other tasks during software designing and reengineering.

Using analytical description of software structural constituents and interconnections between them allows developing methods for solving tasks that are mentioned above by means of analysis and comparison of analytical expressions.

Related works

Necessity of development approaches that are connected with the analysis of software constituents is a prerequisite appearing of works that are devoted to formalization of class diagram description [Zhu, 2012; Bayley, 2011; Wentao, 2012].

Representation of design patterns as algebraic systems is proposed in paper [Zhu, 2012].

An algebraic system takes into account structural constituents of design pattern, namely classes and interfaces. Behavioral constituents of design patterns are represented by a set of messages. These messages are used for information exchanging between objects during solving tasks of design patterns.

Also the analytical description of design patterns constituents while creating their combination is proposed in paper [Bayley, 2011].

Approach proposing identification of design patterns from existing code by means of interconnections analysis between components of frameworks and processes was proposed in paper [Wentao 2012]. For representation of software static models it was proposed to describe a framework as a set of predicates and variables characterizing a problem domain. Such an approach helps to build behavioral models for research problem domain processes.

Mathematical apparatus for defining practicability for code reuse by means of mapping ontologies is proposed in paper [Chang, 2012]. Authors define indexes for mapping of some class diagram constituents (indexes of matching base and independent classes), documentation, attributes and functional requirements of components. After comparison of these indexes, the percent of matching a problem domain framework and application is defined, the practicability of components reuse from the framework is defined.

In papers [Zhu, 2012; Bayley, 2011] classes of design patterns are described as a set named “Classes”. This representation is too schematic.

Usage of technics for analytical description of class diagrams constituents allows defining roles of classes in a design pattern, to analyze effectiveness of design patterns combination, to compare predicates variables to functionality of some class elements and consider new indexes of ontologies mapping.

Task and challenges

Task: design of the algebra for analytical description of software static models, which allows describing complex diagrams of classes, packages and components as well as operations performed on these components and interconnections between them.

Challenges: the algebra describing software static models allows:

- precise a description of design patterns structural constituents;
- design formats of files for saving software static models;
- provide a mathematical apparatus for formalized description of interconnections between class diagram elements. It can serve as a base to prove theorems, verifying statements and so on;
- propose code generation tools while analyzing functional requirements to application. Doing this it is necessary to match software functional requirements with components of class diagram;
- solve other tasks, requiring analysis of interconnections between constituents of class diagram, for example code reuse task.

Algebra describing software static models

For representing of the algebra for description of software static models describe main constituents of algebra and operations that are performed on them.

1. Constituents of the algebra

1.1 Class

Define a class C as a subset of cartesian product of the following sets: properties – A, fields – X and methods – B

$$\begin{cases} C \subseteq A \times X \times B \\ A \times X \times B = \{<\alpha_1,\chi_1,\beta_1>,<\alpha_1,\chi_1,\beta_2>,<\alpha_1,\chi_2,\beta_1>\ldots<\alpha_n,\chi_m,\beta_k>\} \end{cases} \quad (1)$$

The power of such a set is denoted as follows $|A \times X \times B| = n \cdot m \cdot k$, where n – is a number of class C properties, m – is a number of its fields, k – is a number of methods. Respectively number of tuples in (1) defined by number of properties, fields and methods which interact with each other. Every triple can contain one empty set. All properties and method of a class C with modifier private are denoted as follows $C^{private}$, public - C^{public} and protected - $C^{protected}$ respectively. Class C is denoted as follows:

$$C = C^{public} \cup C^{private} \cup C^{protected} \quad (2)$$

At least one set from the expression (2) can't be empty.

All elements of a set X (fields of class C) are related to $C^{private}$, that is

$$C^{private} = C^{private} \cup X \quad (3)$$

If when the description starts the name of class is known class is denoted as follows $C(name)$.

1.2 Abstract class

Denote an abstract class as C^a. All properties and methods of a class C^a with modifier abstract are denoted as follows $C^{abstract}$.

An abstract class has all properties of an ordinary class (1), only set $C^{abstract}$ shouldn't be empty, that is $C^{abstract} \neq \varnothing$. When tuples of an abstract class (1) are formed, two elements from three may be described as empty sets.

1.3 Interface

Interface is denoted as I and described as a set of methods B^I, $I = \{\beta^I\}$ with empty realization. Every method β_j^I corresponds to the condition $\beta_j^I = \varnothing,\ j = 1,\ldots,n^I$ where n^I – number of methods in interface I.

$$\begin{cases} I = \{B^I\}\ B^I = \{\beta_1^I, \beta_2^I, \ldots, \beta_{n^I}^I\} \\ B^I = \{\beta^I \mid \beta_j^I = \varnothing\}\ j = 1,\ldots,n^I \end{cases} \quad (4)$$

The note: an upper index in denotation (for instance B^I) shows, that component belongs to particular whole (in our example B^I - set of methods in interface I). A lower index in denotation defines a number of component in a set of entities.

1.4 Component

We consider component according to widely used definition of components is the following, due to Szyperski: “A software component is a unit of composition with contractually specified interfaces and explicit context

dependencies only. A software component can be deployed independently and is subject to composition by third parties[5]".

Denote a set of component classes, as Δ and a class from this set as C_j^{comp} $j = 1,...,n^{comp}$, where n^{comp} - number of classes in component. That is:

$$\Delta = \{C_1^{comp} \cup C_2^{comp} \cup ... \cup C_{n^{comp}}^{comp}\} \quad (5)$$

1.5 Software module

Consider software module as a set of classes that are gathered in one library.

2. Operation that are performed on algebra constituents

For the description of operations that are performed on classes the term functionality is proposed. Functionality of a class is denoted as a set of methods that can be called and properties can be set. Functionality of a class C is denoted as follows $F(C)$.

The purpose of operations that are performed over some class C is spreading of its functionality. Functionality of some class after performing some operation is denoted as $F(C)^{opeartion}$ where operation – type of operation that is performed over class.

That is, spreading of a functionality, after performing some operations is denoted as follows:

$$F(C)^{opeartion} = F(C) \cup F(C)^*, \quad (6)$$

where $F(C)^*$ - functionality that was added after executing of some operation.

Algebra for description of static software modules defines the following operation: inheritance, polymorphism, association, aggregation and composition.

2.1. Inheritance

Denote C_1 as a successor of a class C_0. Generalize - in hierarchy of inheritance C_j - is a successor of class a C_0 number j. (For instance C_2 - is a successor of a class C_1 and so on). This is also true for interfaces.

If several classes inherit class C_0 set of such classes is denoted as $E = \{C_{11}, C_{12}, ..., C_{1k}\}$, where k – is the number of classes in the set ϕ.

2.1.1 Inheritance of classes

The functionality of a class C_1 after performing inheritance operation, is denoted as follows:

$$F(C_1)^{inh} = F(C_1) \cup (F(C_0) \setminus F(C_0^{private})), \quad (7)$$

where $F(C_1)^{inh}$ - functionality of a class C_1 after performing inheritance operation.

Expression (6) shows that functionality of a successor, namely $F(C_1)$ is spread on base class functionality $F(C_0)$, excluding a functionality of base class private methods , namely $F(C_0^{private})$.

If C_j is a successor number j of a class C_0, then its functionality is spread to functionality of all hierarchy classes $F(C_i)$ where i=0,…,j-1, excluding their private methods functionality $F(C_i^{private})$. It is denoted as follows:

$$F(C_j)^{inh} = F(C_j) \cup (\bigcup_{i=1}^{i=j-1} (F(C_i) \setminus F(C_i^{private}))) \quad (8)$$

2.1.2. Inheritance of interfaces

When a class C inherits an interface I it is necessary to override all methods of this interface. $B^I = \{\beta_i^I \mid \beta_i^I \neq \varnothing\}, i = 1,\dots,n^I$. The functionality of overridden methods is defined as $F(B^I)$. That is,

$$\begin{cases} F(C^{public})^{inh} = F(C^{public}) \cup F(B^I) \\ B^I = \{\beta_i^I \mid \beta_i^I \neq \varnothing\} i = 1,\dots,n^i \end{cases} \quad (9)$$

If a class C inherits p interfaces (multiply interface inheritance), its functionality $F(C^{public})^{inh}$ is spread over functionality of interfaces that are inherited by a class that is:

$$\begin{cases} F(C^{public})^{inh} = F(C^{public}) \cup \bigcup_{j=1}^{p} F(B^{I_j}) \\ B_j^{I_j} = \{\beta_i^{I_j} \mid \beta_i^{I_j} \neq \varnothing\} i = 1,\dots,n^{I_j} \end{cases}, \text{ j=1, …, p} \quad (10)$$

2.2. Polymorphism

Consider three classes in hierarchy C_0, C_1 and C_2. A set of methods, that belong to class C_0 with the modifier virtual is denoted as follows - $C_0^{virtual}$. Sets of methods, that belong to classes C_1 (C_2) with the modifier override is denoted as $C_1^{override}$ ($C_2^{override}$). Consider the following methods $\beta^{C_0} \in \mathrm{B}^{C_0}$, $\beta^{C_1} \in \mathrm{B}^{C_1}$ and $\beta^{C_2} \in \mathrm{B}^{C_2}$, signatures of methods $\beta^{C_0} \in \mathrm{B}^{C_0}$ and $\beta^{C_1} \in \mathrm{B}^{C_1}$ ($\beta^{C_2} \in \mathrm{B}^{C_2}$) differ only modifiers virtual and override respectively, and signatures of methods $\beta^{C_1} \in \mathrm{B}^{C_1}$ and $\beta^{C_2} \in \mathrm{B}^{C_2}$ fully match.

Functionality of the method $F(\beta^{C_2})$ is denoted as follows $F(\beta^{C_2}) = F(\beta^{C_i})$, where

$$i = \begin{cases} i = 2, \beta^{C_2} \neq \varnothing \\ i = 1, \beta^{C_2} = \varnothing \text{ and } \beta^{C_1} \neq \varnothing \end{cases} \quad (11)$$

Generalize: functionality of a method $\beta^{Cj} \in \mathrm{B}^{C_j} \text{ and } \beta^{Cj} \in C_j^{virtual}$ is denoted as follows: $F(\beta^{Cj}) = F(\beta^{C_i})$, where $i = \max(\beta^{C_i}) \neq \varnothing, i = 1,\dots,j$.

2.3. Association

Consider two independent classes C_0^{I} and C_0^{II}, that are not interconnected by relationship of inheritance. Define a set of C_0^{I} methods in class, that are called from class C_0^{II} as Ω. If classes C_0^{I} and C_0^{II} are interconnected by

relationship of association, then the functionality of C_0^{I} spreads on methods from the set Ω. It is denoted as follows:

$$F(C_0^{I})^{acc} = F(C_0^{I}) \cup \Omega, \Omega \neq \varnothing \quad (12)$$

where $F(C_0^{I})^{acc}$ - is a functionality of class C_0^{I} when it is interconnected by relationship of association with class C_0^{II}.

2.4. Aggregation

Functionality of a class C_0^{I} when it is interconnected by relationship of aggregation with class C_0^{II} is denoted as follows:

$$F(C_0^{I})^{aggr} = F(C_0^{I}) \cup (F(C_0^{II}) \setminus F(C_0^{II private})) \quad (13)$$

where $F(C_0^{I})^{aggr}$ - is a functionality of class C_0^{I} when it is interconnected by relationship of aggregation with the class C_0^{II}. When object of type C_0^{II} is created in a method of class C_0^{I}, it is possible to call all methods of this object C_0^{II}, excluding private.

2.5. Composition

Considering, that aggregation and composition relationships are differ by its content, not by structure when classes C_0^{I} and C_0^{II} are interconnected by relationship of aggregation, the functionality of class C_0^{I} is spreads similar to (14) and is denoted as follows:

$$F(C_0^{I})^{comp} = F(C_0^{I}) \cup (F(C_0^{II}) \setminus F(C_0^{II\ private})) \quad (14)$$

where $F(C_0^{I})^{comp}$ - is a functionality of class C_0^{I} when it is interconnected by relationship of composition with the class C_0^{II}.

Example of describing class diagram for the problem domain “designing dense layings for cutting schemas”

Describe interconnections between entities of a chosen problem domain in terms of software static model description algebra.

Consider two subdomains of chosen domain, namely “design dense laying for shoes details” and “design dense laying for leather goods details”. Laying is basic for designing of layout and sections that are used for cutting schemas designing.

Classes of the problem domain – “designing of cutting schemas for roller materials” and interconnections between them are represented in paper [Чебанюк, 2012], mathematical models for estimation of cutting schemas effectiveness and algorithms of their designing were represented in papers [Чупринка, 2012; Чупринка, 2011; Чебанюк, 2008].

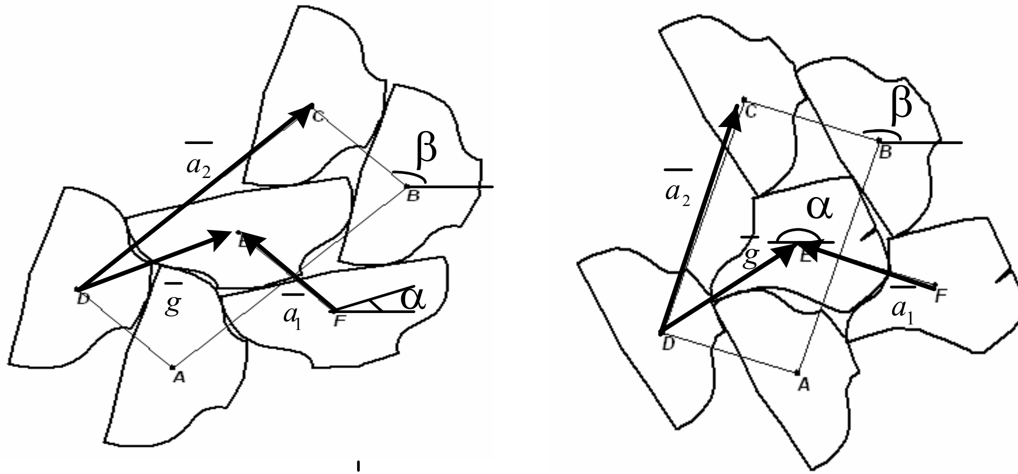
In order to define constituents on class diagram it is necessary to make an analysis of problem domain classes “designing dense laying of cutting schemas” Classes: $C(\det ail)$ – represents detail in laying. Representation of detail was proposed in [Чебанюк, 2008]. $C(grid)$ – represents grid of laying. $C(a\lg orithm)$ – represents

algorithm building of grid considering peculiarities and technological limitations of chosen problem subdomain. $C(Laying)$ – represents a laying.

Define the interconnection between problem domain entities. In order to do this it is necessary to consider the peculiarities of algorithms allowing to build different types of dense laying.

Statement of problem designing of dense laying for two types of details, having arbitrary configuration of edges: among double grid layings $W = W(\overline{a_1}, \overline{a_2}, \overline{g})$ of polygons $S_1(\alpha)$ and $S_2(\beta)$, having density of laying $\delta_S(W)$ find laying $W^* = W(\overline{a_1}^*, \overline{a_2}^*, \overline{g}^*)$, corresponds to the following condition (pict. 1): the density $\delta_S(W^*)$ of double grid laying for polygons $S_1(\alpha)$ and $S_2(\beta)$, was built on this grid corresponds the following statement:

$$\delta_S(W^*) = \max_W \delta_S(W) = \max_W \frac{|S_1| + |S_2|}{S_{abcd}} = \max_W \frac{|S_1| + |S_2|}{\det W} = \max_W \frac{|S_1| + |S_2|}{|[\overline{a_1} \times \overline{a_2}]|} \quad (15)$$



Pict. 1. Examples of double grids

At our case $\overline{a_1} = \overline{AD} = \{a_{1x}, a_{1y}\}$, $\overline{a_2} = \overline{AB} = \{a_{2x}, a_{2y}\}$, $\overline{g} = \overline{AE} = \{g_x, g_y\}$, S_1 and S_2 - closed polygons, that are represented by details, $|S_1|$ and $|S_2|$ – squares of these details (pict. 1), $S_1(\alpha)$ and $S_2(\beta)$ - polygons $S_1(S_2)$ that are rotated on angle $\alpha(\beta)$, Considering the fact, that $|S_1|$ and $|S_2|$ are the constant values, the task can be formulated differently: among all double grids $W = W(\overline{a_1}, \overline{a_2}, \overline{g})$, are available for laying of figures $S_1(\alpha)$ and $S_2(\beta)$, find such $W^* = W(\overline{a_1}^*, \overline{a_2}^*, \overline{g}^*)$, which determinant has the minimum value.

An alignment of details in laying while cutting schemas of shoes details designing is done using mathematical apparatus of locus of vector function of dense positioning [Чебанюк, 2008]. The condition, proving the existence of a laying that vectors AD and FE are equal.

While designing cutting schema of leather goods details an alignment of details in laying is due to taking in accounts linear effects [9]. The best laying is characterized by maximum value of linear effects by height and weight.

Class Laying, namely $C(Laying)$, has the next properties – details that are used for building of a laying, namely $C(\det ail1)$, $C(\det ail2)$; grid for building of a laying, namely $C(grid)$, a percent of material usage, namely P, square of laying, namely S.

Methods of class laying – an estimation of a percent of usage of material, namely estimate (); creation of a laying, namely - create(); saving of laying parameters, namely - save(); visualization of laying parameters, namely - visualize().

Analytical representation of a class C(Laying):

$$\begin{cases} \text{C(Laying)} = \text{AxB} \\ \text{A} = \{\alpha_0,...,\alpha_4 \mid \alpha_0 = \text{C(detail1)}, \alpha_1 = \text{C(detail2)}, \alpha_2 = \text{S}, \alpha_3 = \text{C(grid)}, \alpha_4 = \text{P}\} \\ \text{B} = \{\beta_0,...,\beta_3 \mid \beta_0 = \text{create}, \beta_1 = \text{estimate}, \beta_2 = \text{save}, \beta_3 = \text{visualize}\} \end{cases} \quad (16)$$

Considering, that algorithms of designing a layings are different, when laying of different types are built , we propose the design pattern “abstract factory” for realization software for building dense laying.

From a class $C(Laying)$ we inherit two classes $C(Leather_laying)$ and $C(Shoe_laying)$ for designing of laying for shoe and leather goods details respectively.

Then class $C(Laying)$ and its method Create() make an abstract, namely $C^a(Laying)$

Describe classes $C(Leather_laying)$ and $C(Shoe_laying)$. This classes are designed for creating of single grid when single laying is created [9]. Laying, were built on single grids, allow to design cutting schemas with details of one type. Functionality of a class $C(Leather_laying)$ is denoted as follows:

$$\begin{cases} F(C(Leather_laying))^{inh} = F(C(Leather_laying) \cup C^a(Laying)) \\ C(Leather_laying) = B \\ B = \{Create\} \end{cases} \quad (17)$$

Class $C(Shoe_laying)$ is described similarly to statement (17).

Consider, that in order to design laying for two types of details, one need usage additional operations in comparison with process of single grid design. That is why classes $C(Leather_layng_two)$ and $C(Shoe_laying_two)$ are successors of classes $C(Leather_laying)$ and $C(Shoe_laying)$ - respectively.

Functionality of a class $C(Leather_layng_two)$ is denoted as follows:

$$\begin{cases} F(C(Leather_laying_two))^{inh} = F(C(Leather_laying_two)) \cup F(C(Leather_laying)) \\ C(Leather_laying_two) = B \\ B = \{Create\} \end{cases} \quad (18)$$

Functionality of a class $C(Shoe_laying_two)$ is described similarly to statement (18).

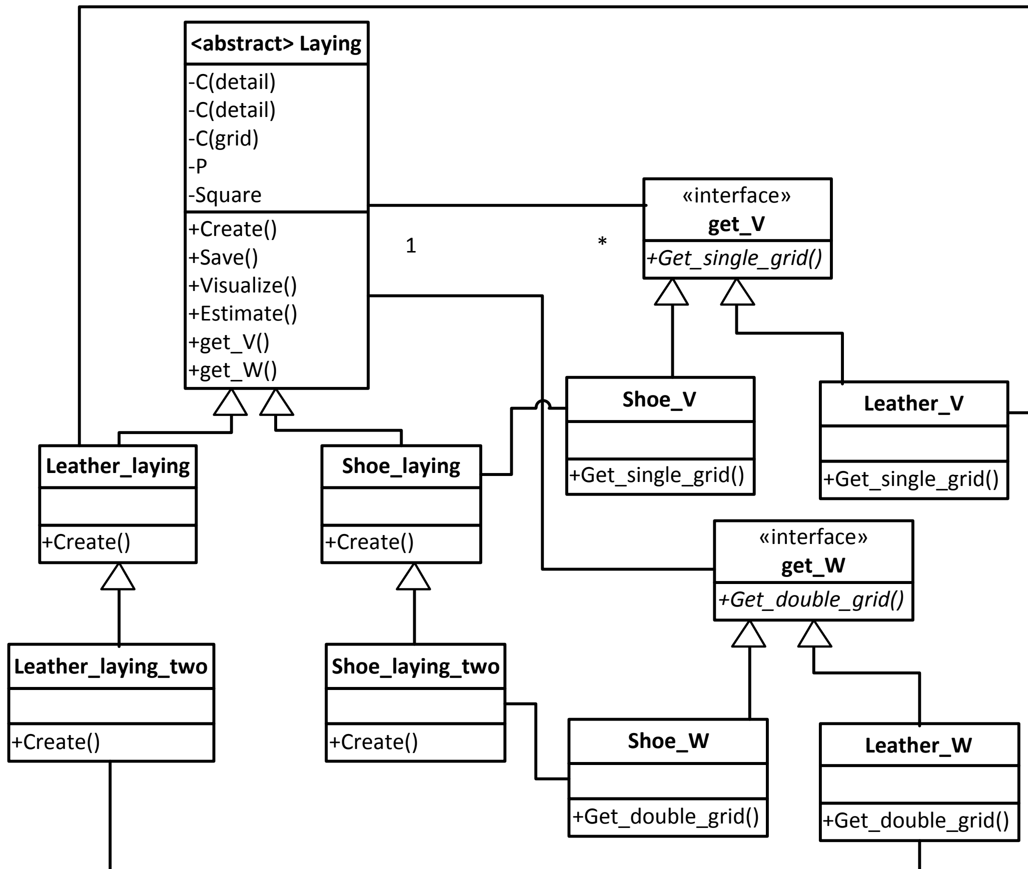
In order to design pattern “abstract factory” define interfaces, are used for designing of layings. Doing this analyze the processes of problem domain [Чупринка, 2012; Чупринка, 2011; Чебанюк, 2008] and interconnections between classes.

Define interfaces: interface for obtaining of a single grid - $I(Get_V)$, interface for obtaining of a double grid - $I(Get_W)$. It is necessary to define different interfaces due to fact that different parameters are calculated when layings of different types are designed.

Design a class diagram for solving task of chosen problem domain using design pattern "abstract factory". "Abstract factory" needs when different types of layings and vectors are created. In order to do this describe interconnections between classes, using terms of the algebra for description of software static models.

$$\begin{cases} \begin{cases} C^a(Laying) = A \times B, A = \{\alpha_0,...,\alpha_5\} \\ C(Laying)^{abstract} = \{\beta_0 \mid \beta_0 = Create\} \\ F(C^a(Laying))^{acc} = F(C^a(Laying)) \cup I(Get_V) \cup I(Get_W) \end{cases} \\ \begin{cases} F(C(Leather_laying))^{inh} = F(C(Leather_laying)) \cup F(C^a(Laying)) \\ F(C(Leather_laying))^{acc} = F(C(Leather_laying)) \cup \Omega, \\ \Omega = \{\beta_0 \mid \beta_0 = Get_\sin gle_grid\}\ \beta_0 \in Leather_V \end{cases} \\ \begin{cases} F(C(Leather_laying_two))^{inh} = F(C(Leather_laying_two)) \cup F(C(Leather_laying)) \\ F(C(Leather_laying_two))^{acc} = F(C(Leather_laying_two)) \cup \Omega, \\ \Omega = \{\beta_0 \mid \beta_0 = Get_double_grid\}\ \beta_0 \in Leather_W \end{cases} \\ \begin{cases} F(C(Shoe_laying))^{inh} = F(C(Shoe_laying)) \cup F(C^a(Laying)) \\ F(C(Shoe_laying))^{acc} = F(C(Shoe_laying)) \cup \Omega, \\ \Omega = \{\beta_0 \mid \beta_0 = Get_\sin gle_grid\}\ \beta_0 \in Show_V \end{cases} \\ \begin{cases} F(C(Shoe_laying_two))^{inh} = F(C(Shoe_laying_two)) \cup F(C(Shoe_laying)) \\ F(C(Shoe_laying_two))^{acc} = F(C(Shoe_laying_two)) \cup \Omega, \\ \Omega = \{\beta_0 \mid \beta_0 = Get_double_grid\}\ \beta_0 \in Show_W \end{cases} \\ \begin{cases} I(Get_V) = \{\beta_0 \mid \beta_0 = Get_\sin gle_grid\} \\ F(C(Shoe_V))^{inh} = F(C(Shoe_V)) \cup I(Get_V) \\ F(C(Leather_V))^{inh} = F(C(Leather_V)) \cup I(Get_V) \end{cases} \\ \begin{cases} I(Get_W) = \{\beta_0 \mid \beta_0 = Get_double_grid\} \\ F(C(Shoe_W))^{inh} = F(C(Shoe_W)) \cup I(Get_W) \\ F(C(Leather_W))^{inh} = F(C(Leather_W)) \cup I(Get_W) \end{cases} \end{cases} \quad (19)$$

The statement (19) represents only those classes and interconnections between them of chosen problem domain, which are essential for design pattern “abstract factory”. Class diagram of chosen problem domain is represented on pict. 2.



Pict. 2 Class diagram of the problem domain “designing dense laying for cutting schemas”

Conclusion

The algebra describing software static models is represented in this paper.

In comparing with description of classes, that was proposed in papers [Zhu, 2012; Bayley, 2011; Wentao, 2012], statements (1)-(14) allow to obtain detailed description constituents of class diagram and interconnections between them, propose analytical description of design patterns and their composition.

Detailed representation of a class diagram allows usage the algebra describing static software models while analyzing practicability of code reuse or designing new software modules or components.

For example, in paper [Chang, 2012] it is possible to make more precise analysis of software modules by means of increasing number of comparing indexes.

Also using the statements (1) - (14) allows to propose format of files, that can be used for saving information about software static models and methods of their visualization.

Usage of the algebra for description software static models with technics of functional requirements analysis allows designing class diagrams by means of analysis problem domain (19) or application processes.

Further exploration

Using the algebra describing software static models allows to represent frameworks analytically by means of grouping class diagrams constituents according to design patterns and propose a method of matching problem domain processes to constituents of a class diagram while designing frameworks.

For this it is necessary to propose a concept of mapping processes characteristics and design patterns. This concept also allows defining necessary components from a framework can be reused while analyzing functional requirements application.

Bibliography

[Bayley, 2011] I. Bayley, H. Zhu. A Formal Language for the Expression of Pattern Compositions. Int'l J. on Advances in Software 4(3&4), Pages 354 – 366, 2011.

[Chang, 2012] S. Chang, F. Colace, M. Santo, E. Zegarra, Y. Qie. An Approach for Software Component Reusing based on Ontological Mapping The 24th International Conference on Software Engineering & Knowledge Engineering Pages 180-187, 2012

[Kung-Kui, 2006] L. Kung-Kiu, M. Ornaghi, Zh. Wang. "A software component model and its preliminary formalisation." Formal Methods for Components and Objects. Springer Berlin/Heidelberg, 2006.

[Wentao, 2012] Wentao Ma, Xiaoyu Zhou, Xiaofang Qi, Ju Qian, Lei Xu, Rui Yang Identification of design patterns using dependence analysis The 24th International Conference on Software Engineering & Knowledge Engineering, Pages 289-292, 2012.

[Zhu, 2012] H. Zhu, I.Bayley. An algebra of design patterns. ACM Transactions on Software Engineering and Methdology in press, 2012.

[Чебанюк, 2008] О.В. Чебанюк, В.І. Чупринка Методика автоматичної побудови розкрійних схем для двох видів плоских геометричних об'єктів. Проблеми програмування. - 2008. – №2-3. – С. 724-730.

[Чебанюк, 2012] О.В.Чебанюк, В.І.Чупринка Метод доменного аналізу для ефективного моделювання процесів при проведенні експериментів з використанням програмного забезпечення. Проблеми програмування. - 2012. – №2-3. – С. 168- 173.

[Чупринка, 2011] В.І. Чупринка, О.В. Чебанюк Автоматизоване проектування раціональних схем розкрою рулонних матеріалів на деталі виробів шкіргалантереї. Вісник Східноукраїнського національного університету ім. Даля. – 2011. – №7(2). – С. 46-50.

[Чупринка, 2012] В.И. Чупринка, Е.В. Чебанюк Математическая модель оценки эффективности раскладок при построении раскройных схем рулонных материалов. Техническое регулирование: базовая основа качества товаров и услуг. :зб. науч. трудов – Шахти ЮРГУЄС , 2012. – С.193-198

Authors' Information

Elena Chebanyuk *– lecturer in National aviation university, associate professor of software engineering department, Ukraine; e-mail: chebanyuk.elena@gmail.com*

Major Fields of Scientific Research: Domain analysis, Domain engineering, Code reuse.