MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
«IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE»

A.I. Antoniuk
A.O. Boldak

SOFTWARE ENGINEERING: LABORATORY WORKSHOP

METHODOLOGICAL INSTRUCTIONS
FOR PERFORMING LABORATORY WORK
IN THE DISCIPLINE "SOFTWARE ENGINEERING"

Recommended by the Methodical Council of Igor Sikorsky KPI
as a study guide for students, who are studying
in the specialty 123 "Computer Engineering"

Kyiv
Igor Sikorsky KPI
2022

Reviewer *Kulakov Y.O.*, Dr. Sci. (Engin.), Prof., Professor of the Department of Computer Engineering of Igor Sikorsky KPI

Responsible editor *Pesarenko A.V.*, PhD (Engin.), Associate Professor

The fretboard was provided by the Methodical Council of Igor Sikorskyi KPI
(protocol No. 1 from 02.09.2022)
at the request of the Academic Council of the Faculty of Informatics and Computer Engineering
(protocol No. 9 from 11.05.2022)

Electronic network educational publication

Antoniuk Andrii Ivanovych, PhD (Engin.),
Boldak Andriy Oleksandrovych, PhD (Engin.)

SOFTWARE ENGINEERING: LABORATORY WORKSHOP

METHODOLOGICAL INSTRUCTIONS
FOR PERFORMING LABORATORY WORK IN THE DISCIPLINE "SOFTWARE ENGINEERING"

Methodological instructions for performing laboratory work in the discipline "software engineering" [Electronic source] : Methodological instructions. for stud. Specialty 123 "Computer Engineering" /A.I. Antoniuk, A.O. Boldak; Igor Sikorsky KPI. - Electronic text data (1 file: 648 KB). – Kyiv : Igor Sikorsky KPI, 2022. – 49 p.

CONTENT

PREFACE

The "Software Engineering" discipline is designed to study programming methods and tools, in particular, design patterns. Students who start studying this discipline have already mastered the "Programming" course and have sufficient skills to create simple programs using C++, Python, Object Pascal and Java programming languages.

The practical part of the course consists of nine laboratory works and is designed to acquire skills in using design patterns in software development. All laboratory work is performed in the Eclipse integrated development environment. Papers are sequentially ordered by complexity and cover all topics studied in the course.

The material for each laboratory work contains a goal, theoretical references and recommendations, a general task, options for individual tasks, a list of questions for self-checking, the content of a report on the performance of laboratory work, as well as a list of recommended information sources for the preparation and performance of laboratory work.

The sequence of laboratory work is as follows:

1. Preparation of the software project.
2. UML graphic notation. Project documentation.
3. Composite, Decorator and Proxy software design structural patterns.
4. Flyweight, Adapter, Bridge and Facade software design structural patterns.
5. Iterator, Mediator, Observer behavior patterns.
6. Behavior patterns of Strategy and Chain of Responsibility.
7. Memento, State, Command and Interpreter behavior patterns.
8. Prototype, Singleton and Factory Method creational patterns.
9. Abstract Factory and Builder creational patterns.

LABORATORY WORK №1.
PREPARATION OF THE SOFTWARE PROJECT

Theme

Preparation of the software project

Purpose

To acquire basic skills in using the XML language. Study of the structure of a typical software project, formats of standard project description files. Learning the JAR format. Acquiring skills in using tools for automating the process of building software projects in Java - Apache ANT (Another Neat Tool). Development of a software project based on a typical example.

Brief theoretical information

Extensible Markup Language (abbreviated XML) is a standard proposed by the World Wide Web (W3C) consortium for constructing markup languages for hierarchically structured data for storage and exchange. Is a simplified subset of the SGML markup language. An XML document consists of text characters and is human-readable.

Logical structure. A markup tool in XML is a tag used to define the boundaries of an element. The tag can be: initial (<Element Name>), end (</Element Name>) and empty or closed (<Element Name/>). An XML document has a hierarchical structure consisting of elements, associated attributes and instructions. A non-empty element is defined by a pair of tags (start and end) with the name of that element and the body contained between those tags. The body of an element consists of other elements and/or text. An empty (no body) element can be defined using a single empty tag with the name of this element.

Attributes are a sequence of key-value pairs (attribute name="attribute value") and are either in the initial or in an empty thesis and are associated with the element whose name is specified in the tag. Also, using special tags, document processing instructions (<?Processor parameter?>) and comments (<!-- Comment text -->) are defined.

Correctness. A well-formed document complies with all syntax rules of XML. A document that is not valid cannot be called an XML document. A valid XML document must correspond to:

- The document has only one root element.
- Non-empty elements are marked with start and end tags. Empty elements can be marked with a "closed" tag.
- One element cannot have multiple attributes with the same name. Attribute values are enclosed in either single (') or double (") quotes.
- Tags can be nested, but cannot overlap. Each non-root element must be completely inside another element.
- The actual and declared encoding of the document must match.

Validity. A document is called valid if it is correct and semantically correct, that is, it corresponds to a certain XML dictionary defined by a schema (DTD, XML Schema, or another).

A JAR file is a Java archive that is a ZIP archive with additional metadata in a manifest file (META-INF/MAMFEST.MF). The manifest can contain information about the project name, version, author, starting class, file signature, etc. Any ZIP-compatible archiver can be used to create a JAR.

Apache Ant (an acronym - "Another Neat Tool") is a java utility for automating the process of assembling a software product. Unlike make, the Ant utility is completely platform-independent, requiring only the presence of the Java operating environment (JRE) installed on the operating system. Rejection of the use of operating system commands and the XML format provide the possibility of transferring scripts.

The build process is managed using an XML script, also called a Build file (build.xml). This file contains a root project element consisting of target elements (<target>). Goals are the equivalent of procedures in programming languages and contain calls to task commands. A task is an XML element associated with a java class that performs a certain elementary action. Frequently used tasks: javac, copy, delete, mkdir, jar. Dependencies can be defined between goals (the depends attribute) — each goal is executed only after all goals on which it depends have been fulfilled (if they have already been fulfilled earlier, re-execution is not carried out). Typical examples of goals are clean (removal of intermediate files), compile (compilation of all classes), deploy (deployment of the application on the server). The specific set of goals and their relationship depend on the specifics of the project. Ant allows you to define your own task types by creating Java classes that implement certain interfaces and binding that class to a script element using the <taskdef> element.

Tasks

0. Learn the syntax and structure of the XML language. Be fluent in the concepts of tag, element, and attribute. Be able to edit XML files in a text editor.
1. Familiarize yourself with the purpose and structure of JAR archives. Be able to create JAR archives and run Java classes from the JAR archive using command line tools. Know the purpose of the JAR archive signature.
2. Familiarize yourself with the ANT project assembly automation tool. Study the purpose and structure of the build.xml file, its structural components - goals, tasks, dependencies, etc.
3. Download the archive of a standard project for the Eclipse environment (template.zip) from the laboratory website. Familiarize yourself with the directory structure of a typical project, project description XML files (.project, .classpath, build.xml).
4. Import a typical project into the workspace of the Eclipse environment. Run the TestMain class in the com.labll package. Familiarize yourself with the means of using ANT in the Eclipse environment - editing the build.xml file, performing goals in the ANT view. Execute ANT targets from the Eclipse environment and from the command line.
5. Modify the typical project for its use in subsequent laboratory works. Be sure to replace the name and author of the project in the project description files (.project, build.xml). Modify the build.xml file so that it contains all the required targets, be free to assign each target.
6. Develop and test in eclipse a new target in the build.xml file according to the option.

Task variants

The number of the task variant is calculated as the remainder of the division of the score book number by 9.

0. Create a new-out directory. Copy all project files with jar extension to this directory.
1. Delete from the project all files with the extensions tmp, jar, class, except for those that begin with the letter "a".
2. Create a jar-archive from all project files with java, js, html, htm extensions in the out directory. Copy the archive to the root of the project.
3. Create a jar-archive from all project files with the extension txt in the out directory. Delete such files from the project.
4. Create the build2 directory. Compile the raw project files in the created directory.
5. Delete all files with the extension jar from the out directory and below. Package all files in the src directory starting with the letter “z” into a jar.
6. Create a jar-archive of the project with a name containing the date and time of creation. When creating an archive, skip files with zip and jar extensions.
7. Delete all files with the jar extension from the root of the project. Package the entire project except the one contained in the out directory into a jar project.
8. Compile the test branch of the project. Package the compiled classes in a jar.

Question for self-check

1. Purpose of the XML language. The concept of a dictionary.
2. The structure of the XML document. Types of tags.
3. Correctness of the XML document. Conditions for a well-formed XML document.
4. The structure of the JAR archive, the purpose of the manifest.
5. How to create an executable JAR archive. How to run classes in a JAR archive.
6. Assignment of directories in a typical project.
7. Assignment of .project, .classpath, build.xml files in a typical project.
8. Appointment of ANT. Its difference from make.
9. Structure of the build.xml file. An XML dictionary for ANT.
10. Goals and objectives. Algorithm for creating goals and tasks.
11. Sequence of ANT actions after starting from the command line. Syntax for running ANT from the command line.
12. ANT tools for working with the file system (creating/deleting directories, copying, etc.).
13. ANT tools for compiling raw Java files.
14. ANT tools for creating a JAR archive.
15. Eclipse tools for working with ANT.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the contents of the project directory with relevant comments and a printout of the .project, .classpath, build.xml files with relevant comments, and the results of program testing. A source code file (build.xml) is attached to the protocol.

Recommended references

1. XML - Wikipedia. URL: https://en.wikipedia.org/wiki/XML.
2. XML Tutorial. URL: https://www.w3schools.com/xml.
3. Using JAR Files: The Basics. URL: https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html
4. Apache Ant - Wikipedia. URL: https://en.wikipedia.org/wiki/Apache_Ant
5. Apache ANT – A Tool for Automating Software. URL: https://www.softwaretestinghelp.com/apache-ant-selenium-tutorial-23/

LABORATORY WORK №2. UML GRAPHIC NOTATION, PROJECT DOCUMENTATION

Theme

UML graphic notation, project documentation

Purpose

Familiarization with UML diagram types. Getting basic skills in using the UML language class diagram. Acquisition of skills in the use of UML modeling automation tools using the example of ArgoUML/Umbrello. Project documentation using JavaDoc.

Brief theoretical information

UML (Unified Modeling Language) is a unified modeling language used in the object-oriented programming paradigm. It is an integral part of the unified software development process. UML is intended for visualizing, designing, modeling, and documenting software development. UML is not a programming language, but there are code generation tools based on UML. UML consists of 13 diagrams divided into structural and behavioral.

Structure diagrams:

- Classes - Class diagram
- Component - Component diagram
- Composite structure diagram - Collaboration diagram (UML2.0)
- Deployment - Deployment diagram
- Objects - Object diagram
- Packages - Package diagram

Behavior diagrams:

- Activities - Activity diagram
- Finite automata (states) - State Machine diagram
- Precedents - Use case diagram

Interaction diagrams:

- Cooperation - Collaboration (UML 1.x) / Communications - Communication (UML2.0)
- Interaction overview diagram (UML2.0)
- Sequences - Sequence diagram
- Synchronization - UML Timing Diagram (UML2.0)

A class diagram is a static representation of the model structure. Provides static (declarative) elements, such as: classes, data types, their content and relations. A class diagram can also contain notation for packages and can contain notation for nested packages. Also, a class diagram can contain the designation of some elements of behavior, however, their dynamics are revealed in diagrams of other types.

A class is indicated by a rectangle with three parts: the upper one contains the class name, the middle one contains a list of class attributes, and the lower one contains a list of class operations. Additionally, list elements can be marked with types and scope. The relationship between classes is indicated by various types of lines and arrows. There are the following types of relationships:

- Association - shows that objects of one entity (class) are related to objects of another entity. It is indicated by a solid line. It can be unary - one-sided (indicated by an open arrow indicating the direction of association) and binary. More than two classes can participate in the association, such associations are indicated by lines, one end of which goes to the class block, and the other to the general rhombus. Associations can be named, then roles, affiliations, indicators, multipliers, visibility or other properties are marked at the ends of its line.
- Aggregation - a kind of association, with the relationship between the whole and its parts. As a type of association, an aggregation can be named. Aggregation cannot include several classes at once. Aggregation occurs when one class is a collection or container of others. The existence time of part classes does not depend on the existence time of the whole class. If the container is destroyed, its contents are not. Graphically, aggregation is indicated by an empty rhombus on the whole-class block and a line running from this rhombus to the part-class.

- Composition - a stricter variant of aggregation. During composition, there is a strict dependence of the existence time of the class-whole and classes-parts. If the container is destroyed, all its contents will be destroyed as well. Graphically, it is denoted as aggregation, but with a colored rhombus.
- Generalization (Inheritance) - shows that one of the two related classes (subtype) is a more specific form of the other (supertype), which is called a generalization of the first. In practice, this means that any instance of a subtype is also an instance of a supertype. Graphically, generalization is represented by a line with a closed arrow (empty triangle) in the supertype. Generalization is also known as imitation or "is a" type of connection.
- Implementation - the relationship between two elements of the model, in which one element (the client) implements the behavior specified by the other (the supplier). Graphically, the implementation is presented as well as the generalization, but with a dotted line.
- Dependency is a relationship of use in which a change in the specification of one entails a change in the other, and the opposite is not necessary. It is graphically indicated by a dotted arrow going from the dependent element to the one on which it depends.

Javadoc - generator of documentation in HTML format from comments of raw Java code from Sun Microsystems. Javadoc is a standard for documenting Java classes. Javadoc also provides APIs for creating doclets and taglets that allow the programmer to analyze the structure of a Java application.

Documentation comments are used to document classes, interfaces, fields (variables), constructors, methods, and packages. In each case, the comment must precede the element being documented. Javadoc descriptors begin with the "@" character.

Tasks

1. Familiarize yourself with the purpose and types of diagrams of the UML language. To study the class diagram, to be fluent in elements and relationships between them. Be able to construct class diagrams for Java raw code, as well as generate program code equivalent to a given class diagram.
2. Construct a class diagram containing three interfaces If1, If2, If3 with methods meth1(), meth2(), meth3 and classes that implement them Cl1, Cl2, Cl3, respectively.
3. According to the option (below), implement generalization and aggregation relations on the class diagram.
4. In the prepared project (LW1), create the program package com.lab111.labwork2. In the package, develop interfaces and classes according to the diagram (items 3-4). The implementation of the methods must output the class name and method name to the console).
5. Familiarize yourself with UML modeling automation tools. Be able to use ArgoUML and Umbrello environments at a basic level to develop class diagrams and document software.
6. Using the ArgoUML or Umbrello environment, import the raw codes of the com.lab111.labwork2 package and check the compliance of the constructed class diagram with the developed one (items 3-4). Save the diagram in the project documentation directory.
7. Familiarize yourself with the comment syntax for the JavaDoc documentation automation tool. Modify the raw codes of package com.lab111.labwork2 by adding comments in JavaDoc format.
8. Generate JavaDoc using Eclipse (Project menu) in the project documentation directory.
9. Develop an ANT target for JavaDoc generation. Generate a JavaDoc using the developed ANT target.

Task variants

Generalization (inheritance)

The number of the task variant is calculated as the remainder of dividing the score book number by 9.

0. If1 <- If2; If2 <- If3; Cl1 <- Cl3
1. If1 <- If3; If3 <- If2; Cl1 <- Cl2
2. If1 <- If2; If1 <-If3; Cl2 <- Cl3
3. If2 <- If1; If1 <- If3; Cl1 <- Cl3
4. If2 <- If1; If3 <- If1; Cl2 <- Cl1
5. If2 <- If1; If2 <- If3; Cl2 <- Cl3
6. If3 <-If2; If2 <- If1; Cl2 <- Cl1
7. If3 <- If1; If1 <- If2; Cl3 <- Cl1
8. If3 <- If2; If3 <- If1; Cl3 <- Cl2

Aggregation

The number of the task variant is calculated as the remainder of dividing the score book number by 5.

0. If1 <- Cl1; Cl3 <- Cl2; Cl3 <- Cl3
1. If1 <- Cl2; Cl1 <- Cl3; Cl1 <- Cl1
2. If1 <- Cl3; Cl1 <- Cl2; Cl1 <- Cl1
3. If1 <- Cl1; If2 <- Cl1; Cl3 <- Cl2
4. If3 <- Cl2; If2 <- Cl3; Cl3 <- Cl1

Question for self-check

1. Purpose of the UML language.
2. Brief description of UML diagrams.
3. Elements of the class diagram and the relationship between them. Unary and binary relations.
4. Difference between association, aggregation and composition.
5. Relationship of imitation. Notation in the diagram and an example of Java raw code.
6. Implementation relationship. Notation in the diagram and an example of Java raw code.
7. Relationship of association. Notation in the diagram and an example of Java raw code.
8. Aggregation relations. Notation in the diagram and an example of Java raw code.
9. Relationship of the composition. Notation in the diagram and an example of Java raw code.
10. Dependency relationship. Notation in the diagram and an example of Java raw code.
11. Multipliers and roles. Purpose and notation.
12. Stereotypes. Purpose and notation.
13. Types of UML modelers.
14. Purpose of JavaDoc. Syntax of JavaDoc comments.
15. ANT tools for working with JavaDoc.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Unified Modeling Language. URL: https://en.wikipedia.org/wiki/Unified_Modeling_Language
2. Comparison of Unified Modeling Language tools. URL: http://en.wikipedia.org/wiki/List_of_UML_tools
3. Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language Usere Guide. Second edition. - Addison-Wesley, 2005. – 475 p.
4. Class diagram. URL: https://en.wikipedia.org/wiki/Class_diagram
5. Sequence diagram. URL: https://en.wikipedia.org/wiki/Sequence_diagram
6. ArgoUML Documentation Resources. URL: http://www.lysator.liu.se/xenofarm/argouml/work/argouml/www/documentation/index.html
7. Welcome to Umbrello - The UML Modeller. URL: http://uml.sourceforge.net/
8. Umbrello Documentation. URL: https://umbrello.kde.org/documentation.php
9. Javadoc. URL: https://en.wikipedia.org/wiki/Javadoc
10. How to Write Doc Comments for the Javadoc Tool. URL: https://www.oracle.com/cis/technical-resources/articles/java/javadoc-tool.html
11. How to generate a PDF from JavaDoc. URL: https://stackoverflow.com/questions/2322048/how-to-generate-a-pdf-from-javadoc-including-overview-and-package-summaries

LABORATORY WORK №3.
STRUCTURAL DESIGN PATTERNS.
PATTERNS COMPOSITE, DECORATOR, PROXY

Theme

Structural design patterns. Patterns Composite, Decorator, Proxy

Purpose

Getting to know the types of software design patterns. Study of structural patterns. Getting basic skills in using Composite, Decorator and Proxy patterns.

Brief theoretical information

Composite

Problem. How to treat an atomic object and a composition of objects in the same way?

Decision. Define classes for composite (Composite) and atomic (Leaf) objects so that they implement the same interface (Component). The structure of the pattern is shown in Fig.1.

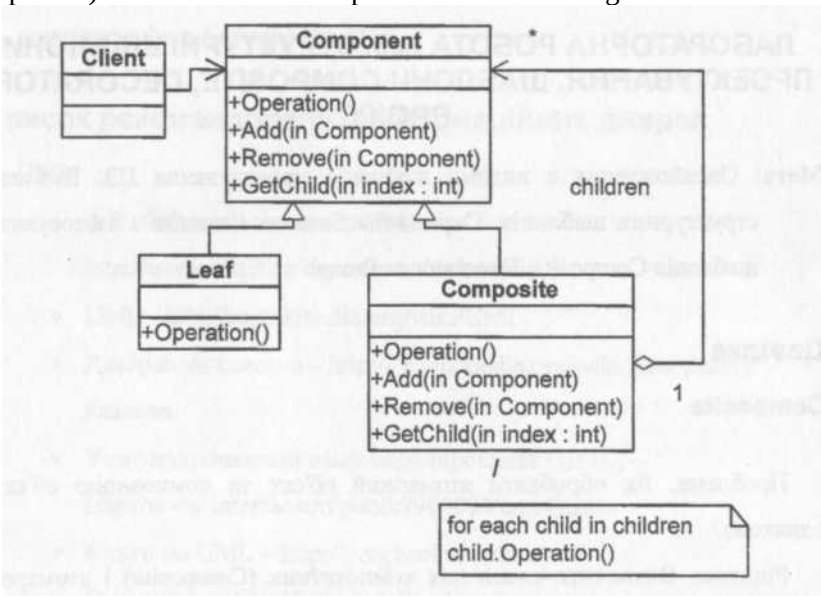

Fig. 1. Structure of the Composite pattern

The Composite pattern consists of the following parts:

- Component - this component is designed for:
  - declaring an interface for composition objects and implementing the corresponding default behavior for an interface common to all classes.
  - declaring an interface for accessing and managing descendants.
  - defining an interface to access the parent component in the recursive structure and implement it, if appropriate (optional).
- Leaf - leaf or petal, this elementary object is intended for:
  - presentation of leafy (that is, elementary or primitive) objects in the composition; a leaf has no offspring.
  - determining the behavior of primitive objects in the composition.
- Composite - it is a composite object designed for:
  - defining the behavior of components that have children.
  - storage of child components.
  - implementation of the child management operation in the Component class interface.
- Client - this component is designed to manage objects in a composition through the Component interface

Decorator

Problem. Put additional duties (transparent to clients) on a separate object, and not on the class as a whole.

Decision. In dynamics, add new duties to the object, without resorting to the generation of subclasses (inheritance). "Component" defines an interface for objects that can be dynamically assigned additional responsibilities, "ConcreteComponent" defines an object that is assigned additional responsibilities, "Decorator" - stores a reference to the "Component" object " and defines an interface corresponding to the "Component" interface. "ConcreteDecorator" imposes additional responsibilities on the component. "Decorator" redirects requests to the "Component" object. The structure of the pattern is shown in Fig.2.

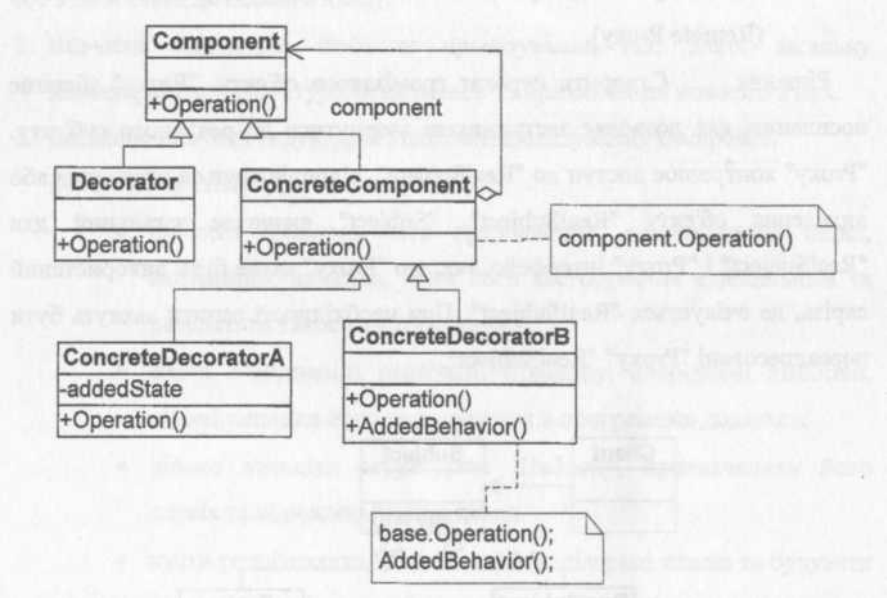

Fig. 2. Structure of the Decorator pattern

The Decorator pattern consists of the following parts:

• Component - this component is designed to define the interface of objects to which new responsibilities can dynamically depend.

• ConcreteComponent - this component is intended to define an object that can be assigned additional responsibilities.

• Decorator - this component is designed to hold a reference to the Component object and define an interface that corresponds to the Component interface.

• ConcreteDecorator - this component is intended to add component responsibilities.

Proxy

Problem. It is necessary to control access to the object for the following purposes:

- create bulky objects "on demand" (Virtual Proxy);
- cache data (SmartLink Proxy);
- dynamically restrict access (Security Proxy):
- provide access to the object in another address space (Remote Proxy).

Decision. Create a surrogate of a bulky object. "Proxy" stores a link that allows the proxy to contact the real entity. A "Proxy" controls access to a "RealSubject" by being responsible for creating or deleting a "RealSubject". "Subject" defines an interface common to "RealSubject" and "Proxy", so that "Proxy" can be used wherever "RealSubject" is expected. If necessary, requests can be redirected to "Proxy" "RealSubject". The structure of the pattern is shown in Fig.3.

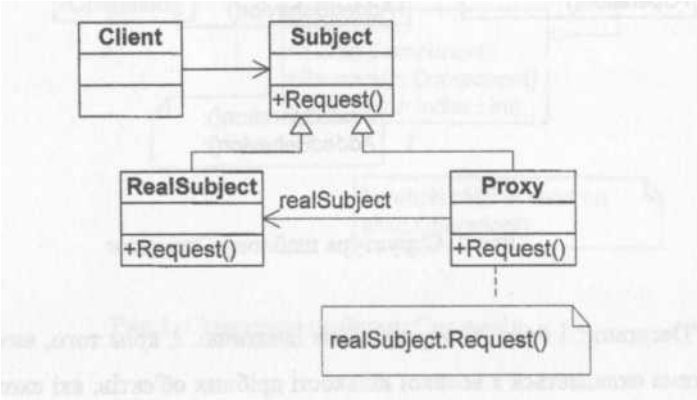

Fig. 3. Structure of the Proxy pattern

The Proxy pattern consists of the following parts:

- Proxy - deputy, this component is intended for:
  - storing a link that allows the proxy to access the real entity; a deputy can refer to a subject if the interfaces of the RealSubject and Subject classes are the same.
  - providing an interface identical to the subject so that the proxy can replace the real subject.
  - controlling access to the real entity and taking responsibility for its creation and deletion.
  - the following duties depend on the type of authorized deputy:
    - it is the responsibility of remote proxies to encode the request and its arguments, and to send the encoded request to a real entity in another address space.
    - the ability of virtual deputies to cache additional information about a real subject in order to postpone its creation.
    - the ability of deputy defenders to check whether the subscriber has the access permissions necessary to fulfill the request.
- Subject - this component is intended to define a common interface for RealSubject and Proxy, in order to allow a proxy to be used in place of RealSubject anywhere.
- RealSubject - this component is intended to define the real-world entity that the proxy represents.

Applying several "Decorators" to one "Component" allows you to combine duties arbitrarily, for example, one property can be added twice.

Greater flexibility than static inheritance: you can add and remove responsibilities at runtime, while using inheritance you would have to create a new class for each additional responsibility. This pattern allows you to avoid classes overloaded with methods at the upper levels of the hierarchy - new duties can be added as needed.

"Decorator" and its "Component" are not identical, and, in addition, it turns out that the system consists of a large number of small objects that are similar to each other and differ only in the way they are interconnected, not in the class and not in the values of their internal variables - such a system is difficult to study and debug.

Tasks

1. Familiarization with the purpose and types of software design patterns. Study the classification of software design patterns. Know the names of patterns that belong to a certain class.
2. To study structural patterns of software design. Know the general characteristics of structural patterns and the purpose of each of them.
3. Study in detail the Composite, Decorator and Proxy structural design patterns. For each of them:
   - study the pattern, its purpose, alternative names, motivation, cases when its use is appropriate and the results of such use;
   - know the peculiarities of the implementation of the pattern, related patterns, known cases of its application in software applications;
   - fluently master the structure of the pattern, the assignment of its classes and the relationships between them;
   - be able to recognize a pattern in the UML class diagram and build the raw codes of Java classes that implement the pattern.

4. Create the com.lab111.labwork3 software package in the prepared project (LR1). In the package, develop interfaces and classes that implement tasks (according to the option) using one or more patterns (p.3). In the classes being developed, fully implement the methods related to the functioning of the pattern. The methods that implement business logic should be closed with stubs that output information about the called method and its arguments to the console. Example of business method implementation:

   void draw(int x, int y){ System.out.println("Draw method with parameters x="+x+" y="+y); }

5. Complete documentation of the developed classes (also methods and fields) using automated means, while the documentation should sufficiently highlight the role of a certain class in the general structure of the pattern and the specifics of the specific implementation.

Task variants

The number of the task variant is calculated as the remainder of dividing the score book number by 12.

0. Define the specifications of the classes that provide graphic primitives and their compositions in the vector graphics editor. Each primitive has placement attributes - position (x and y coordinates) and size (width and height). Implement a business method for displaying such placement attributes for primitives (specified in the constructor) and compositions (dynamically calculated).
1. Define the specifications of the classes that provide a parsing tree of a complex expression with quotation marks according to the syntactic rules:

   <expression>::=<simple expression> | <complex expression>
   <simple expression>::=<constant> | <variable>
   <constant>::=(<number>)
   <variable>::=(<name>)
   <complex expression>::-(<expression><operation sign><expression>)
   <operation sign>::=+|-|*|/

   Implement a business method for displaying the content of an element as an expression.
2. Define class specifications for representing the game space with a multi-level hierarchical structure. Implement a business method for calculating the area occupied by an element in conventional units.
3. Define class specifications for presenting block diagrams of algorithms with block organization according to the semantic diagram (Fig.4). Implement a business method for identifying an element and its relationships with other flowchart elements.

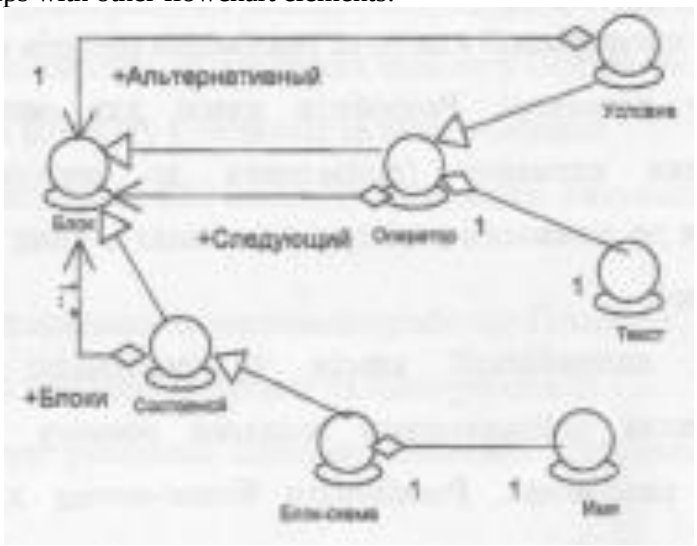

Fig. 4. Semantic diagram of algorithm block diagrams

4. Define the class specifications for presenting the file system in the form of a tree of objects (a file is a leaf object; a directory is a node object). Each object has a size attribute (for a file it is set in the constructor, for directories it is calculated). Implement a get size business method for the directory class.
5. Define class specifications and implementation of methods for rendering the selected graphic element in the vector graphics editor. Provide the ability to dynamically change the display of the element.
6. Define the specifications of classes of additional graphic images for graphic elements in the vector graphics editor. Give examples of the use of the developed wrapper classes.
7. Define class specifications for rendering graphic manipulators of geometric properties (position, size) in the vector graphics editor.
8. Define class specifications and implementation of methods for elements in a text editor. Develop classes for dynamically changing the display of an element (uppercase, lowercase, adding a newline character at the end, etc.).
9. Define class specifications and method implementations for manipulating large images with transparent caching. Implement a business method to determine the color of a point by its coordinates.
10. Define class specifications and implementation of methods for manipulating images with the possibility of their "late loading". Implement a business method to determine the color of a point by its coordinates.
11. Define the specifications of classes and the implementation of methods for manipulating images with the possibility of controlling access to the object — access is open only to points whose coordinates (x, y) lie within $x1<x<x2$ and $y1<y<y2$ (values of x1, x2, y1, y2 are specified in the constructor). Implement a business method to determine the color of a point by its coordinates.

Question for self-check

1. Software design patterns. Appointment. A brief history of creation.
2. Classification of software design patterns.
3. Designation of software design structural patterns.
4. Brief description of each structural pattern.
5. Names, purpose and motivation of the Composite pattern.
6. The structure of the Composite pattern and its members.
7. Features of the implementation of the Composite pattern. The result of using the pattern.
8. Names, purpose and motivation of the Decorator pattern.
9. The structure of the Decorator pattern and its members.
10. Features of the implementation of the Decorator pattern. The result of using the pattern.
11. Names, purpose and motivation of the Proxy pattern.
12. Structure of the Proxy pattern and its participants.
13. Features of the implementation of the Proxy pattern. The result of using the pattern.
14. Types of Proxy pattern.
15. Patterns used in conjunction with Composite, Decorator and Proxy.
16. The difference between Decorator and Proxy in the constructor specification.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Design Patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Indianapolis: - Addison-Wesley, 1994. - 417 p. ISBN: 0201633612.
2. Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1, 2nd Edition. - Wiley Publishing, 2002. - 480 p. ISBN: 0471258393
3. Design Patterns. URL: https://en.wikipedia.org/wiki/Design_Patterns
4. Design Pattern – Overview. URL: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
5. Structural Design Patterns. URL: https://refactoring.guru/design-patterns/structural-patterns
6. Structural Design Pattern. URL: https://www.scaler.com/topics/design-patterns/structural-design-pattern/
7. What is Structural Design Pattern? URL: https://www.scaler.com/topics/design-patterns/structural-design-pattern/
8. Structural design patterns. URL: https://www.javatpoint.com/structural-design-patterns
9. Structural pattern – Wikipedia. URL: https://en.wikipedia.org/wiki/Structural_pattern
10. Structural patterns. URL: https://sourcemaking.com/design_patterns/structural_patterns/

LABORATORY WORK №4.
STRUCTURAL DESIGN PATTERNS.
PATTERNS FLYWEIGHT, ADAPTER, BRIDGE, FACADE

Theme

Structural design patterns. Patterns Flyweight, Adapter, Bridge, Facade.

Purpose

Study of structural patterns. Obtaining basic skills in the use of Flyweight, Adapter, Bridge, Facade patterns.

Brief theoretical information

Flyweight

Problem. It is necessary to provide support for a large number of small objects.

Decision. Create an object that can be used in multiple contexts at the same time, and, in each context, it appears as an independent object (indistinguishable from a non-distributable instance). "Flyweight" declares an interface through which widgets can obtain external state or otherwise affect it, "ConcreteFlyweight" implements the "Flyweight" class interface and adds internal state as needed. The internal state is stored in the "ConcreteFlyweight" object, while the external state is stored or computed in the "Client" ("Client" passes it to "Flyweight" when operations are called).

An object of class "ConcreteFlyweight" is allocated. Any state in it must be internal, i.e., independent of the context, "FlyweightFactory" - creates objects - "Flyweight" (or provides an instance that already exists) and manages them. "UnsharedConcreteFlyweight" - not all subclasses of "Flyweight" are necessarily shared. "Client" - stores a reference to one or more "Flyweights", calculates and stores the external state of the "Flyweight". The structure of the pattern is shown in Fig.5.

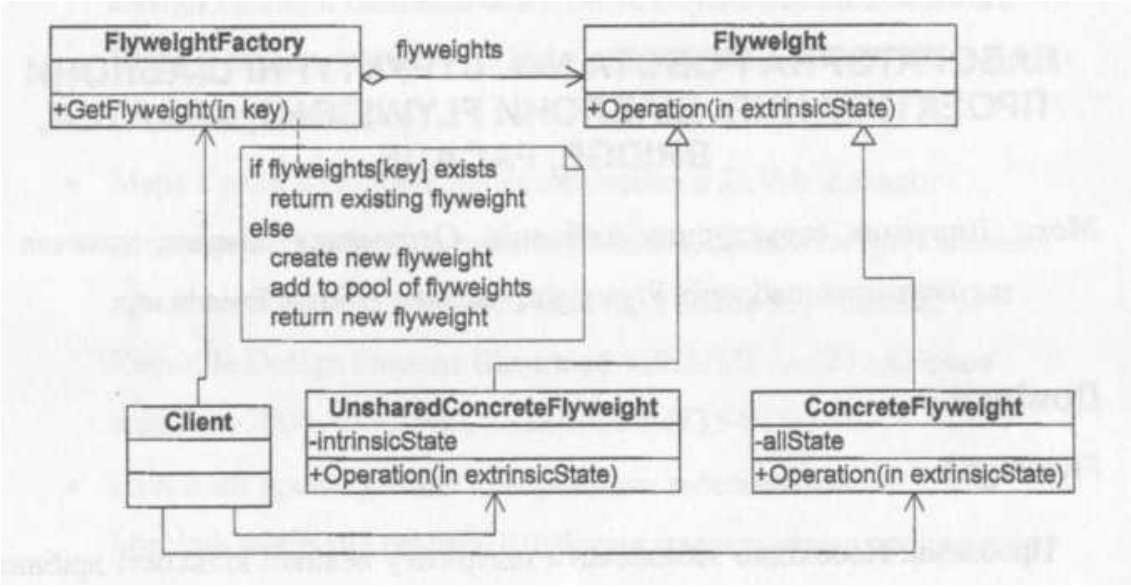

Fig. 5. Structure of the Flyweight pattern

The Flyweight pattern consists of the following parts:

- Flyweight - lightweight (adapter), this component is intended to declare an interface through which adapters can receive external state and modify it in some way.
- ConcreteFlyweight - this component is designed to implement the Flyweight class interface and add internal state as needed. An object of the ConcreteFlyweight class must be distributed (that is, shared). Any state it stores must be internal, meaning it must be context-independent.

- UnsharedConcreteFlyweight - this component is intended to anchor certain Flyweight subclasses, meaning not all subclasses need to be divided. The Flyweight interface allows separation but does not enforce it. Often, UnsharedConcreteFlyweight objects have ConcreteFlyweight objects as descendants at some level of the fixture structure.
- FlyweightFactory - this component is designed to create and manage Flyweight objects, and to ensure proper separation of Flyweights. If the client requests a flyweight, the FlyweightFactory object provides an existing instance or creates a new instance if one does not exist.
- Client - this component is designed to hold a reference to the lightweight(s) and to compute or store the external state of the lightweight(s).

Flyweights model entities that are too large to be represented by objects. It makes sense to use this pattern if the following conditions are met at the same time:

- the application uses a large number of objects, due to which the storage costs are quite high,
- most of the state of objects can be taken outside,
- many groups of objects can be replaced by a relatively small number of objects because the state of the objects is externalized.

Due to the reduction of the total number of instances and state output, memory is saved.

Adapter

Problem. It is necessary to ensure the interaction of incompatible interfaces or how to create a single stable interface for several components with different interfaces.

Decision. Convert the original interface of the component to another type using an intermediate object - an adapter, that is, add a special object with a common interface within the framework of this program and redirect communication from external objects to this object - an adapter. The structure of the pattern is shown in Fig.6.

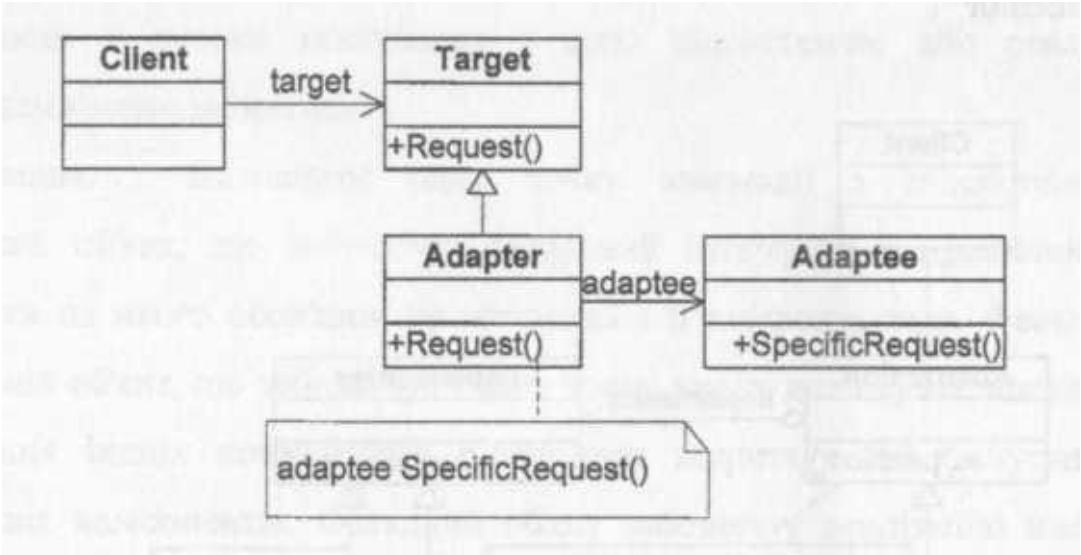

Fig. 6. Structure of the Adapter pattern

The Adapter pattern consists of the following parts:

- Target - this component is intended to define the domain-specific interface used by the client.
- Client - this component is designed to interact with objects corresponding to the target interface.
- Adaptee - this component is designed to identify an existing interface that needs to be adapted.
- Adapter - this component is designed to adapt the Adaptee interface (adapted) to the target interface.

Bridge

Problem. It is necessary to separate the abstraction from the implementation so that both can be changed independently. When using imitation, the implementation is tightly bound to the abstraction, which makes independent modification difficult.

Decision. Place abstraction and implementation in separate class hierarchies. "Abstraction" defines the "Abstraction" interface and stores a reference to the "Implementor" object, "RefinedAbstraction" extends the interface specified in "Abstraction". "Implementor" defines the interface for the implementation classes, it does not have to exactly match the interface of the "Abstraction" class - both interfaces can be completely different. Usually, the interface of the "Implementor" class provides only primitive operations, and the "Abstraction" class defines higher-level operations based on these primitives. "ConcreteImplementor" contains a concrete implementation of the "Implementor" class. The "Abstraction" object redirects "Client" requests to its "Implementor" object. The structure of the pattern is shown in Fig.7.

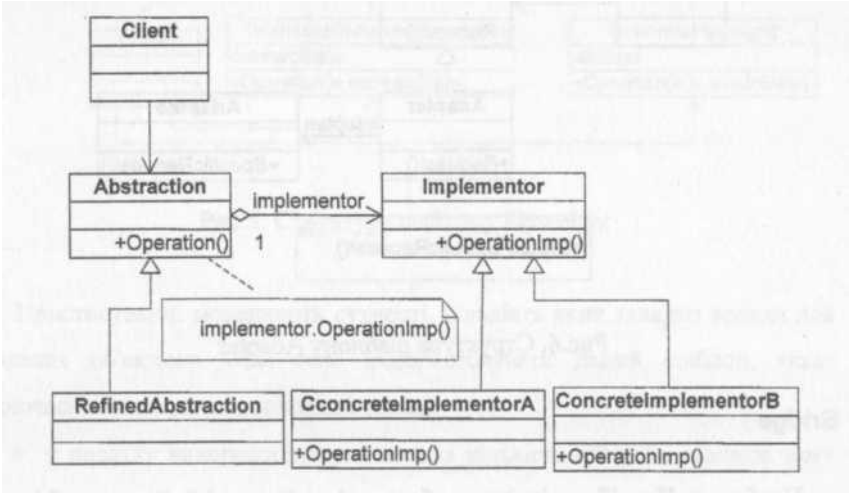

Fig. 7. Structure of the Bridge pattern

The Bridge pattern consists of the following parts:

- Abstraction - this component is designed to define an abstraction interface and store a reference to an object of Implementor type.
- RefinedAbstraction - this component is designed to extend an interface that was defined by Abstraction.
- Implementor - this component is designed to define the interface of the implementation classes. This interface need not exactly match the Abstraction interface; in fact, the two interfaces can be quite different. Typically, the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ConcreteImplementor - this component is designed to implement the Implementor interface and define its specific implementation.

A pattern can be applied if, for example, the implementation needs to be changed during the program's operation.

Decoupling the implementation from the interface allows runtime configuration of the Implementor and Abstraction. In addition, it should be noted that the separation of the "Abstraction" and "Implementor" classes eliminates implementation dependencies that are established at the compilation stage: to change the "Implementor" class, it is not necessary to recompile the "Abstraction" class at all.

Facade

Problem. How to provide a unified interface with a set of disparate implementations or interfaces, for example, with a subsystem, if high coupling to that subsystem is undesirable or the implementation of the subsystem may change?

Decision. Define one point of interaction with the subsystem - a facade object that provides a common interface with the subsystem and assign it the responsibility of interaction with its components. A facade is an external object that provides a single point of entry for subsystem services. The implementation of other

subsystem components is closed and not available to external components. The facade object provides the implementation of the "Resistant to changes" pattern from the point of view of protection against changes in the implementation of the subsystem.

Use the Facade pattern when

• you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

• there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.

• you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

The structure of the pattern is shown in Fig.8.

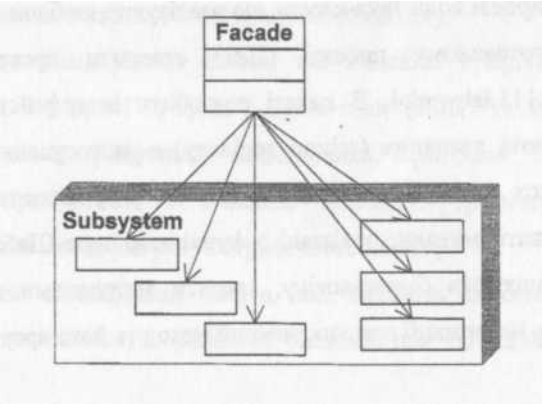

Fig. 8. Structure of the Facade pattern

The Facade pattern consists of the following parts:

- Facade – this component is designed to store and hide information about which system classes are responsible for the request; this component is also designed to delegate customer requests to the appropriate system objects.
- Subsystem classes – classes of subsystems, these components are intended for:
  - implementation of functionality of subsystems.
  - performance of the work assigned by the "Facade" object.
  - isolation from the facade, meaning they do not store any references to the facade.

Tasks

1. To establish the purpose of software design patterns, their classification. Know the name and brief description of each of the patterns belonging to a certain class.
2. Repeat the structural patterns of software design. Know the general characteristics of structural patterns and the purpose of each of them.
3. Study in detail the structural design patterns of Flyweight, Adapter, Bridge, Facade. For each of them:
   - study the pattern, its purpose, alternative names, motivation, cases when its use is appropriate, and the results of such use;
   - know the peculiarities of the implementation of the pattern, related patterns, known cases of its application in software applications;
   - fluently master the structure of the pattern, the assignment of its classes and the relationships between them;
   - be able to recognize a pattern in the UML class diagram and build source codes of Java classes that implement the pattern.
4. In the prepared project (LR1), create the program package com.lab111.labwork4. In the package,

develop interfaces and classes that implement tasks (according to the option) using one or more patterns (p.3). In the classes being developed, fully implement the methods related to the operation of the pattern. Methods that implement business logic should be closed with stubs that output information about the called method and its arguments to the console.

5. Complete documentation of the developed classes (including methods and fields) using automated means, while the documentation should sufficiently highlight the role of a certain class in the general structure of the pattern and the specifics of the specific implementation.

Task variants

The number of the task variant is calculated as the remainder of the division of the score book number by 11.

0. Define class specifications and method implementations for Latin alphabet glyph objects and string objects. Glyph objects have coordinate attributes and are used as components when building composite string objects. Ensure efficient use of memory when working with a large number of glyph objects. Implement a string output method.
1. Define the specifications of the classes that provide graphic objects in the raster graphics editor - primitives (point) and their compositions (rectangular image). Ensure efficient use of memory when working with a large number of graphic objects. Implement the method of drawing a graphic object.
2. Define the specifications of the classes that provide graphic objects in the vector graphics editor - primitives (line) and their compositions (rectangle, triangle). Ensure efficient use of memory when working with a large number of graphic objects. Implement the method of drawing a graphic object.
3. Define the specifications of classes that provide icon objects for the image of file system elements when building a graphic user interface (GUI) - primitives (files) and their compositions (directories). Ensure efficient use of memory when working with a large number of graphic objects. Implement the method of drawing a graphic object.
4. Define the specifications of the classes that provide graphic objects in the vector graphics editor - primitives (point) and their compositions (circle). Primitives have the coordinate attributes in the Cartesian system, and composition objects - in the polar system. Accordingly, the point interface contains the setX(int) and setY(int) methods, and the circle drawing method can only operate with the setRo(double), setPhi(double) primitive methods (which are absent in the point class). Provide the ability to use point functionality when drawing a circle without changing the point interface and circle drawing method.
5. Define the specifications of the classes that provide graphic objects in the vector graphics editor - primitives (point) and their compositions (line). Primitives have integer coordinate attributes (in pixels), and composition objects have rational attributes (in centimeters). Accordingly, the point interface contains the setX(int) and setY(int) methods, and the line drawing method can only operate with the setX(double), setY(double) primitive methods (which are absent in the point class). Provide the ability to use point functionality when drawing a line without changing the point interface and line drawing method.
6. Define the specifications of the classes that provide graphic objects in the vector graphics editor - primitives (line) and their compositions (rectangle). Primitives have the coordinate attributes in a system with the center of coordinates in the upper left corner of the screen with an inverse abscissa axis, and composition objects with the center of coordinates in the middle of the screen and the standard direction of the abscissa axis. Provide the ability to use the line functionality when drawing a rectangle without changing the point and line drawing methods.
7. Define the specifications of the classes that provide elements of the graphical user interface (GUI) — button, window, etc. Ensure separation of abstraction and implementation so that interface elements can have implementations for different libraries (eg Qt and GTK) transparent to the user. Implement the element drawing method.
8. Define the specifications of the classes that provide graphic objects in the vector graphics editor (rectangle) through different API1 and API2 interfaces. Provide transparent for the user the possibility of replacing the implementation of graphic objects. Implement the element drawing method.
9. Define the specifications of the classes that provide objects for manipulating the elements of the file system - files and directories. The file interface contains methods open(String path, boolean

createIfNotExist), close() and delete(String path) to open, close and delete a file (with createIfNotExist=true the file will be created if it does not exist or truncated to zero length if it exists). The directory interface contains the create(String path) and rmdir(String path) methods for creating and deleting a directory. Specify a subsystem of 3 files and 2 directories. Provide the ability to create and delete such a subsystem through the create(), destroy() methods and change the structure of the subsystem without affecting its user.

10. Define the specifications of the classes that provide graphic objects in the vector graphics editor — line (Line) and raster image (Image). The line interface contains the setOpacity(double op) method, which adjusts its opacity level between fully opaque (op == 1.0) and invisible (op == 0.0). The bitmap interface contains the setTransparency(double tr) method, which adjusts its transparency level between fully transparent (tr == 1.0) and fully opaque (tr == 0.0). Specify a subsystem from the image and the lines that frame it. Provide the ability to turn on/off the display of the subsystem through the show(boolean vis) method, and change the type of framing (horizontal, vertical, full, etc.) without affecting the user of this subsystem.

Question for self-check

1. Classification of software design patterns.
2. Designation of software design structural patterns.
3. Brief description of each structural pattern.
4. Names, purpose and motivation of the Flyweight pattern.
5. Structure of the Flyweight pattern and its members.
6. Features of the implementation of the Flyweight pattern. The result of using the patterns.
7. Names, purpose and motivation of the Adapter pattern.
8. The structure of the Adapter pattern and its participants.
9. Features of the implementation of the Adapter pattern. The result of using the pattern.
10. Names, purpose and motivation of the Bridge pattern.
11. Structure of the Bridge pattern and its members.
12. Features of the implementation of the Bridge pattern. The result of using the pattern.
13. Names, purpose and motivation of the Facade pattern.
14. Structure of the Facade pattern and its members.
15. Features of Facade pattern implementation. The result of using the pattern.
16. The difference between Adapter, Decorator and Proxy in the constructor specification.
17. Types of adapters. Two-way and dynamic (pluggable) adapters.
18. Patterns used in conjunction with Flyweight, Adapter, Bridge, Facade.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Design Patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Indianapolis: - Addison-Wesley, 1994. - 417 p. ISBN: 0201633612.
2. Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1, 2nd Edition. - Wiley Publishing, 2002. - 480 p. ISBN: 0471258393
3. Design Patterns. URL: https://en.wikipedia.org/wiki/Design_Patterns
4. Design Pattern – Overview. URL: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
5. Structural Design Patterns. URL: https://refactoring.guru/design-patterns/structural-patterns
6. Structural Design Pattern. URL: https://www.scaler.com/topics/design-patterns/structural-design-pattern/
7. What is Structural Design Pattern? URL: https://www.scaler.com/topics/design-patterns/structural-design-pattern/
8. Structural design patterns. URL: https://www.javatpoint.com/structural-design-patterns
9. Structural pattern – Wikipedia. URL: https://en.wikipedia.org/wiki/Structural_pattern
10. Structural patterns. URL: https://sourcemaking.com/design_patterns/structural_patterns/

LABORATORY WORK №5.
BEHAVIORAL DESIGN PATTERNS.
PATTERNS ITERATOR, MEDIATOR, OBSERVER

Theme

Behavioral design patterns. Patterns Iterator, Mediator, Observer.

Purpose

Learning behavioral design patterns. Getting basic skills in using Iterator, Mediator and Observer patterns.

Brief theoretical information

Iterator

Problem. A composite object, for example, a list, must provide access to its elements (objects) without revealing their internal structure, and it is necessary to sort through the list differently depending on the task.

Decision. The "Iterator" class is created, which defines the interface for accessing and sorting elements, "Concretelterator" implements the "Iterator" class interface and monitors the current position when traversing "Aggregate". "Aggregate" defines the interface for creating an iterator object. "ConcreteAggregate" implements the iterator creation interface and returns an instance of the "Concretelterator" class, "Concretelterator" keeps track of the current object in the aggregate and can calculate the next object when iterating.

The pattern supports various ways of sorting the aggregate, several sortings can be active at the same time. The structure of the pattern is shown in Fig.9.

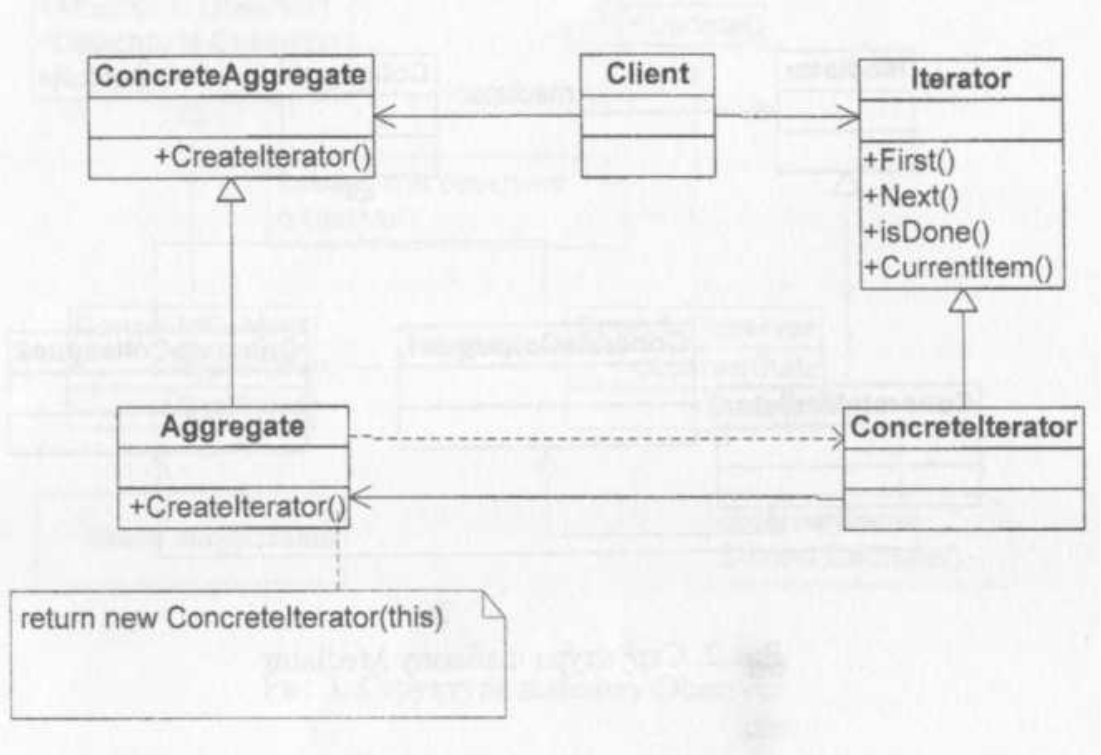

Fig. 9. The structure of the Iterator pattern

The Iterator pattern consists of the following parts:

- Iterator - this component is designed to define the element access and traversal interface.
- ConcreteIterator - this component is designed to implement the Iterator interface and track the current position when traversing the aggregate.

- Aggregate - this component is designed to define the Iterator object creation interface.
- ConcreteAggregate - this component is designed to implement the Iterator creation interface to return an instance of the corresponding ConcreteIterator.

Mediator

Problem. Ensure the interaction of a large number of objects, while ensuring loose coupling and eliminating the need for objects to explicitly refer to each other.

Decision. Create an object that encapsulates the way a large number of objects interact. The structure of the pattern is shown in Fig.9.

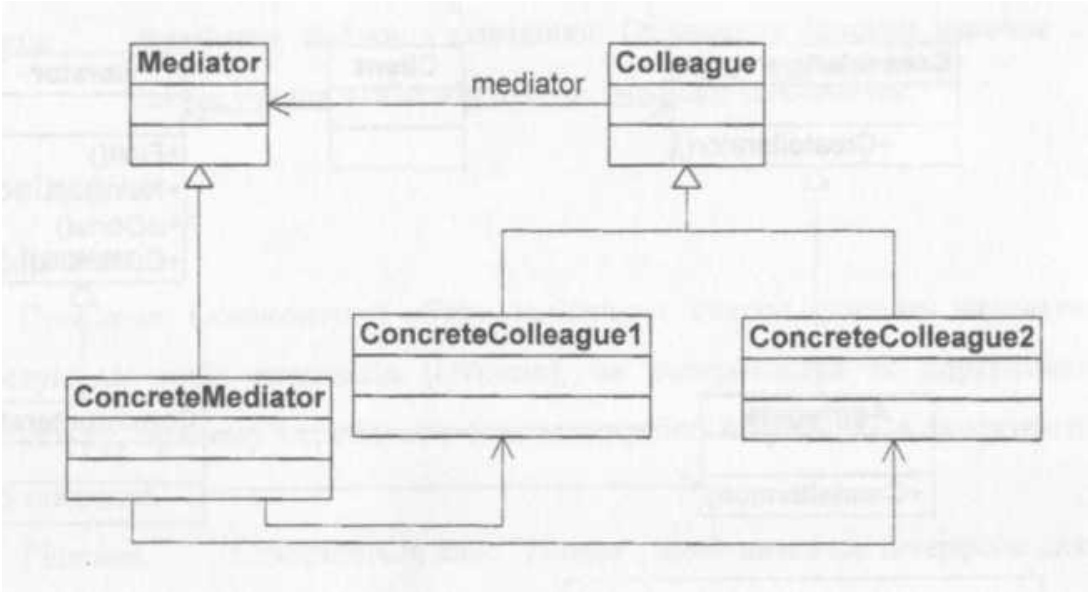

Fig. 10. The structure of the Mediator pattern

The Mediator pattern consists of the following parts:

- Mediator - this component is intended to define a communication interface with Colleague objects.
- ConcreteMediator - this component is designed to implement communication behavior by coordinating Colleague objects and storing information about their Colleagues.
- Colleague classes - this component is designed to define the Colleague class that corresponds to the mediator object and ensure that each Colleague communicates with its mediator, preventing communication with other objects.

"Mediator" defines the interface for exchanging information with "Colleague" objects, "ConcreteMediator" coordinates the actions of "Colleague" objects. Each "Colleague" class knows about its "Mediator" object, all "Colleagues" exchange information only with the mediator, in its absence they would have to exchange information directly. "Col league" send requests to and receive requests from the intermediary. "Mediator" implements cooperative behavior by forwarding each request to one or more "Colleagues". The pattern eliminates the connection between "Colleague", centralizing the management.

Observer

Problem. One object ("Observer") must know about the change of states or some events of another object. At the same time, it is necessary to maintain a low level of communication with the object - "Observer".

Decision. Define the "Observer" interface. "ConcreteObserver" objects - subscribers implement this interface and dynamically register to receive information about some event in "Subject". Then, when the specified event is implemented in "ConcreteSubject", all subscriber objects are notified. The structure of the pattern is shown in Fig.11.

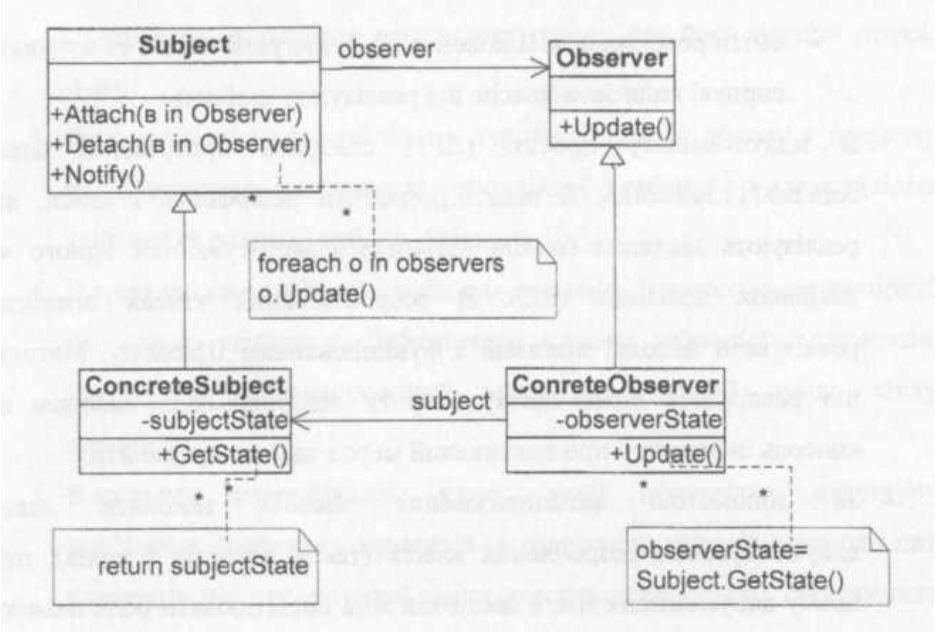

Fig. 11. The structure of the Observer pattern

The Observer pattern consists of the following parts:

- Subject - this component is designed to store information about its observers (the number of observers per subject is not limited) and to provide an interface for attaching and detaching Observer objects.
- Observer - this component is intended to define an update interface for objects that should be notified of changes to the subject.
- ConcreteSubject - this component is designed to store state that is of interest to ConcreteObserver objects and send a message to its observers when the state changes.
- ConcreteObserver - this component is designed for:
    - supporting a reference to the ConcreteSubject object;
    - storage of data that corresponds to the subject's data;
    - implementation of the update interface, which is defined in the Observer class, in order to maintain its state according to the subject.

Tasks

1. Learn behavioral patterns for software design. Know the general characteristics of behavior patterns and the purpose of each of them.
2. Study in detail the behavior patterns for software design - Iterator, Mediator and Observer. For each of them:
    - study the Template, its purpose, alternative names, motivation, cases when its use is appropriate and the results of such use;
    - know the peculiarities of the implementation of the Template, related patterns, known cases of its application in software applications;
    - fluently master the structure of the Template, the assignment of its classes and the relationships between them;
    - be able to recognize a pattern in the UML class diagram and build the raw codes of Java classes that implement the pattern.
3. In the prepared project (LR1), create the com.lab111.labwork5 software package. In the package, develop interfaces and classes that implement tasks (according to the option) using one or more patterns (item 2). In the developed classes, fully implement the methods related to the functioning of the pattern. Methods that implement business logic should be closed with stubs that output information about the called method and its arguments to the console.
4. Complete documentation of the developed classes (including methods and fields) using automated means, while the documentation should sufficiently highlight the role of a certain class in the general structure of the pattern and the specifics of the specific implementation.

Task variants

The number of the task variant is calculated as the remainder of the division of the score book number by 9.

0. Define class specifications that encapsulate a linear list of objects and implement the possibility of sequential traversal in the forward and reverse directions, bypassing the empty elements of this structure and not revealing its essence to the user.
1. Define class specifications that encapsulate a linear list of integers and implement normal sequential traversal and sequential traversal in an ordered structure.
2. Define class specifications that encapsulate a linear list of character strings and implement the possibility of a regular sequential traversal and traversal with additional aggregate filtering (for example, filtering by the length of the string, by its first letter, etc.).
3. Define class specifications for sequential traversal in the forward and reverse directions of the relational table with the possibility of performing a selection (filtering) operation.
4. Define class specifications for the element of the playing field (cell) and the space itself. Ensure weak connection of elements. Implement a centralized mechanism for compatible changes in the state of elements.
5. Define the specification of a class that encapsulates the structure of related graphic elements and the implementation of methods of interaction of these elements during a compatible change of properties (colors). Ensure weak connection of elements.
6. Define class specifications for presenting a relational table and foreign key restrictions with the possibility of checking it when field values are changed. Ensure weak connection of elements.
7. Define class specifications for presenting elements of the graphical user interface — GUI (windows, buttons, text areas). Implement a mechanism for reacting to events in any of the elements.
8. Define class specifications for presenting a relational table. Implement a trigger mechanism — performing additional actions when an element is changed.

Question for self-check

1. Classification of software design patterns.
2. Assignment of behavior patterns for software design.
3. A brief description of each pattern of behavior.
4. Names, purpose and motivation of the Iterator pattern.
5. Structure of the Iterator pattern and its participants.
6. Features of the implementation of the Iterator pattern. The result of using the pattern.
7. Names, purpose and motivation of the Mediator pattern.
8. Structure of the Mediator pattern and its participants.
9. Features of the implementation of the Mediator pattern. The result of using the pattern.
10. Names, purpose and motivation of the Observer pattern.
11. Structure of the Observer pattern and its members.
12. Features of the implementation of the Observer pattern. The result of using the pattern.
13. Patterns used in conjunction with Iterator, Mediator and Observer.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Design Patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Indianapolis: - Addison-Wesley, 1994. - 417 p. ISBN: 0201633612.
2. Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1, 2nd Edition. - Wiley Publishing, 2002. - 480 p. ISBN: 0471258393
3. Design Patterns. URL: https://en.wikipedia.org/wiki/Design_Patterns
4. Design Pattern – Overview. URL: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
5. Behavioral pattern - Wikipedia. URL: https://en.wikipedia.org/wiki/Behavioral_pattern#:~:text=In%20software%20engineering%2C%20behavioral%20design,flexibility%20in%20carrying%20out%20communication
6. Behavioral Design Patterns. URL: https://refactoring.guru/design-patterns/behavioral-patterns
7. Behavioral patterns. URL: https://sourcemaking.com/design_patterns/behavioral_patterns
8. Behavioral Design Patterns. URL: https://www.javatpoint.com/behavioral-design-patterns
9. Behavioral Design Patterns. URL: https://programmingline.com/software-design-patterns/behavioral-design-patterns
10. Behavioral Patterns. URL: https://howtodoinjava.com/design-patterns/behavioral/

LABORATORY WORK №6.
BEHAVIORAL DESIGN PATTERNS.
PATTERNS STRATEGY,
CHAIN OF RESPONSIBILITY, VISITOR

Theme

Behavioral design patterns. Patterns Strategy, Chain of Responsibility, Visitor.

Purpose

Learning behavioral design patterns. Getting basic skills in using Strategy, Chain of Responsibility, Visitor patterns.

Brief theoretical information

Strategy

Problem. How to design changeable but reliable algorithms or strategies.

Decision. Define for each algorithm or strategy a separate class with the standard interface "Strategy".

Multiple “ConcreteStrategy” classes are created, each of which contains the same polymorphic “AlgorithmInterface” method. The strategy object is linked to the context object (the object to which the algorithm is applied). The structure of the pattern is shown in Fig.12.

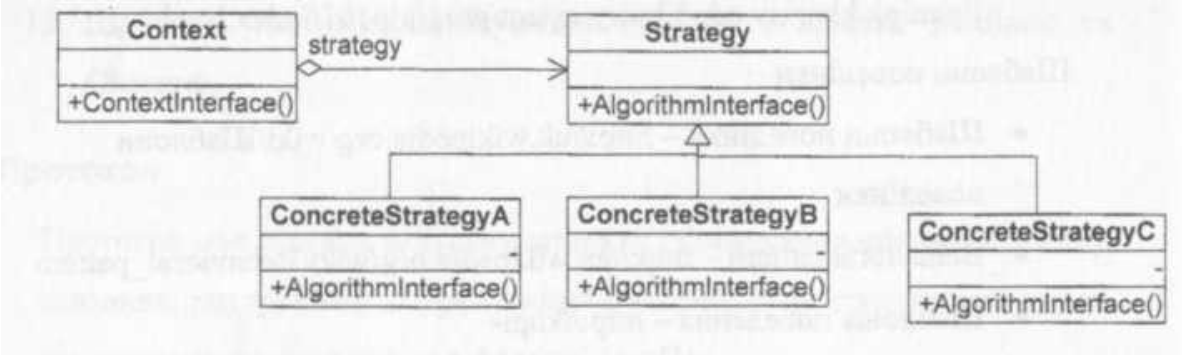

Fig. 12. The structure of the Strategy pattern

The Strategy pattern consists of the following parts:

- Strategy - this component is intended to declare an interface common to all supported algorithms. The Context class uses this interface to call the algorithm defined in ConcreteStrategy.
- ConcreteStrategy - this component is designed to implement the algorithm that was declared in the Strategy interface.
- Context - this component is designed for:
    - settings on the ConcreteStrategy object;
    - storing a link to the Strategy object;
    - defining the interface that allows the strategy to access its data.

Chain of Responsability

Problem. The request must be processed by several objects.

Decision. Bind objects - receivers of the request in a chain and pass the request along this chain until it is processed. "Handler" defines an interface for processing requests, and possibly implements communication with a successor. "ConcreteHandler" handles the request it is responsible for. Having access to its successor, "ConcreteHandler" forwards the request to it if it cannot handle the request itself. The structure of the pattern is shown in Fig.13.

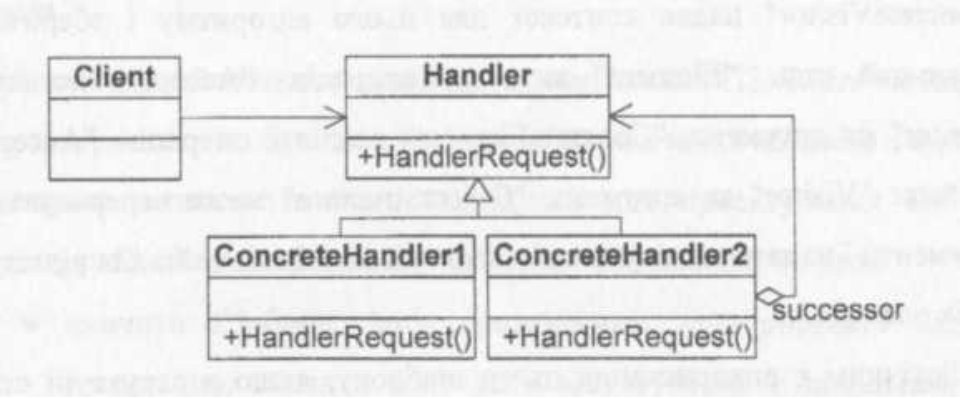

Fig. 13. The structure of the Chain of Responsability pattern

The Chain of Responsability pattern consists of the following parts:

- Handler - this component is intended to define the request processing interface and implement the successor link (optionally).
- ConcreteHandler - this component is designed for:
  - processing requests for which he is responsible;
  - gaining access to your successor;
  - forwarding the request to its successor if the ConcreteHandler cannot handle the request.
- Client - this component is designed to initiate a request to a ConcreteHandler object on the chain.

It is logical to use this pattern if there is more than one object capable of processing the request and the handler is not known in advance (and should be found automatically) or if the entire set of objects capable of processing the request should be specified automatically.

The pattern weakens the connection (the object is not obliged to "know" who exactly will process its request). But there is no guarantee that the request will be processed because it has no explicit recipient.

Visitor

Problem. An operation is performed on each object of some structure. Define a new operation without changing object classes.

Decision. A client using this pattern must create an object of the ConcreteVisitor class and then visit each element of the structure. "Visitor" declares the operation "VisitConcreteElement" for each class "ConcreteElement" (the name and signature of this operation identify the class whose element is visited by "Visitor" - that is, the visitor can access the element directly). "ConcreteVisitor" implements all operations, declarations in the "Visitor" class. Each operation implements a fragment of the algorithm defined for the class of the corresponding object in the structure. The "ConcreteVisitor" class provides the context for this algorithm and stores its local state. "Element" defines an "Accept" operation that takes "Visitor" as an argument, "ConcreteElement" implements an "Accept" operation that takes "Visitor" as an argument. An "ObjectStructure" can enumerate its arguments and provide a high-level interface for the visitor to visit its elements.

It is logical to use this pattern if the structure contains objects of many classes with different interfaces, and it is necessary to perform operations on them that depend on specific classes, or if the classes that define the structure of objects rarely change, but new operations on this structure are often added.

The pattern simplifies the addition of new operations, combines related operations in the "Visitor" class. At the same time, adding new "ConcreteElement" classes is complicated, since a new abstract operation must be declared in the "Visitor" class. The structure of the pattern is shown in Fig.14.

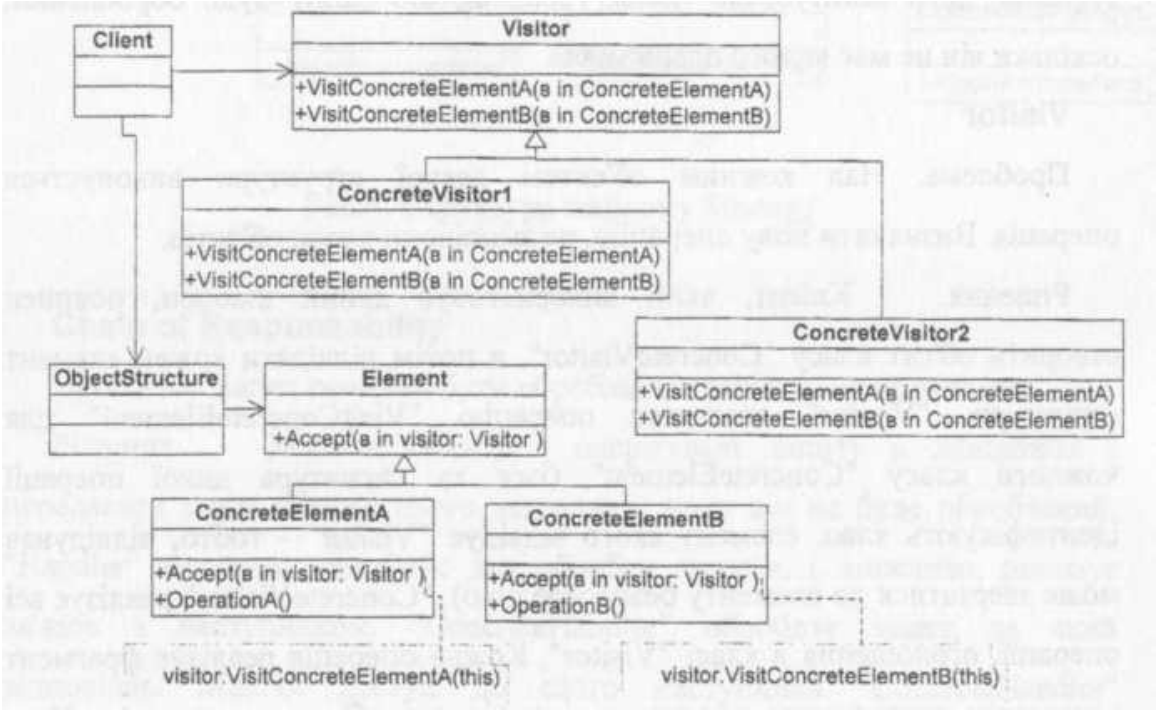

Fig. 14. The structure of the Visitor pattern

The Visitor pattern consists of the following parts:

- Visitor (NodeVisitor) - this component is designed to declare the Visit operation for each ConcreteElement class in the object structure. The operation name and signature identify the class that sends the visit request to the visitor. This allows the visitor to access the element directly through its specific interface.
- ConcreteVisitor (TypeCheckingVisitor) - this component is designed to implement all operations that were declared by the visitor (Visitor). Each operation implements a fragment of the algorithm defined for the corresponding object class in the structure. ConcreteVisitor provides the algorithm context and stores its local state. This condition often accumulates results while traversing the structure.
- Element (Node) - this component is intended to define an Accept operation that accepts a visitor as an argument.
- ConcreteElement (AssignmentNode,VariableRefNode) - this component is designed to implement an Accept operation that accepts a visitor as an argument.
- ObjectStructure (Program) - this component is designed for:
    - enumeration of its elements.
    - providing a high-level interface that allows the visitor to visit their items.
    - the ability to create a composite object (composite or collection, such as a list or set).

Tasks

1. Repeat patterns of behavior for software design. Know the general characteristics of behavior patterns and the purpose of each of them.
2. Study in detail the behavior patterns for software design — Strategy, Chain of Responsibility and Visitor. For each of them:
    - study the pattern, its purpose, alternative names, motivation, cases when its use is appropriate and the results of such use;
    - know the peculiarities of the implementation of the pattern, related patterns, known cases of its application in software applications;
    - fluently master the structure of the pattern, the assignment of its classes and the relationships between them;
    - be able to recognize a pattern in the UML class diagram and build the source codes of Java classes that implement the pattern.

3. In the prepared project (LR1), create the program package com.lab111.labwork6. In the package, develop interfaces and classes that implement tasks (according to the variant) using one or more patterns (item 2). In the developed classes, fully implement the methods related to the functioning of the pattern. Methods that implement business logic should be closed with stubs that output information about the called method and its arguments to the console.
4. Complete documentation of the developed classes (including methods and fields) using automated means, while the documentation should sufficiently highlight the role of a certain class in the general structure of the pattern and the specifics of the specific implementation.

Task variants

The number of the task variant is calculated as the remainder of the division of the score book number by 10.

0. Define the specifications of a class that contains an array of integers and a method of sorting it. Provide the ability to dynamically change the algorithm and sorting direction through external parameterization.
1. Define the specifications of the class that contains the table and the method of displaying it in the form of a diagram. Provide the ability to dynamically change the type of diagram through external parameterization.
2. Determine the specifications of the class that contains the mathematical function and the method of displaying it in the form of a graph. Provide the ability to dynamically change the coordinate system of the graph (Cartesian, polar, etc.) by means of external parameterization.
3. Define the specifications of classes that implement containers for integers and text strings with the possibility of sorting them. Provide the ability to dynamically change the sorting algorithm through external parameterization. The implementation of the sorting algorithm must be independent of the type of data being sorted.
4. Define the specifications of the classes that implement elements of the graphical user interface — panels (composite) and buttons (component). Implement a decentralized mechanism for handling the mouse cursor movement event. The number of interface components that respond to this event can change dynamically.
5. Define the specifications of the classes implementing the processing of HTTP requests of various types (for example, GET and POST). Realize the possibility of dynamically changing the number of processors. Ensure decentralization and weak connection of processors.
6. Define class specifications for the element of the playing field (cell) and the space itself. Ensure weak connection of elements. Implement a decentralized mechanism for compatible changes in the state of elements.
7. Define the specifications of the classes that implement elements of the graphical user interface — panels (composite) and buttons (component). Implement the mechanism of additional operations on the structure of the graphic interface without changing its elements. As an illustration of such a mechanism, develop an operation for counting the number of elements of the same type.
8. Determine the specifications of the classes that implement the elements of the computer structure (processor, memory, video card, etc.). Implement a mechanism for additional operations on the computer structure without changing its elements. As an illustration of such a mechanism, develop an operation for determining the power consumed by a computer.
9. Define the specifications of the classes that implement the elements of the network structure (cable, server, workstation). Implement a mechanism for additional operations on the network structure without changing its elements. As an illustration of such a mechanism, develop an operation for determining the estimate of such a structure.

Question for self-check

1. Classification of software design patterns.
2. Assignment of behavior patterns for software design.
3. A brief description of each pattern of behavior.
4. Names, purpose and motivation of the Strategy pattern.
5. Structure of the Strategy pattern and its participants.
6. Features of the implementation of the Strategy pattern. The result of using the pattern.

7. Names, purpose and motivation of the Chain of Responsibility pattern.
8. Structure of the Chain of Responsibility pattern and its members.
9. Features of the implementation of the Chain of Responsibility pattern. The result of using the pattern.
10. Names, purpose and motivation of the Visitor pattern.
11. Structure of the Visitor pattern and its participants.
12. Features of the implementation of the Visitor pattern. The result of using the pattern.
13. Patterns used in conjunction with Strategy, Chain of Responsibility and Visitor.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Design Patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Indianapolis: - Addison-Wesley, 1994. - 417 p. ISBN: 0201633612.
2. Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1, 2nd Edition. - Wiley Publishing, 2002. - 480 p. ISBN: 0471258393
3. Design Patterns. URL: https://en.wikipedia.org/wiki/Design_Patterns
4. Design Pattern – Overview. URL: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
5. Behavioral pattern - Wikipedia. URL: https://en.wikipedia.org/wiki/Behavioral_pattern#:~:text=In%20software%20engineering%2C%20behavioral%20design,flexibility%20in%20carrying%20out%20communication
6. Behavioral Design Patterns. URL: https://refactoring.guru/design-patterns/behavioral-patterns
7. Behavioral patterns. URL: https://sourcemaking.com/design_patterns/behavioral_patterns
8. Behavioral Design Patterns. URL: https://www.javatpoint.com/behavioral-design-patterns
9. Behavioral Design Patterns. URL: https://programmingline.com/software-design-patterns/behavioral-design-patterns
10. Behavioral Patterns. URL: https://howtodoinjava.com/design-patterns/behavioral/

LABORATORY WORK № 7.
BEHAVIORAL DESIGN PATTERNS.
PATTERNS MEMENTO, STATE, COMMAND, INTERPRETER

Theme

Behavioral design patterns. Patterns Memento, State, Command, Interpreter.

Purpose

Learning behavioral design patterns. Getting basic skills in using Memento, State, Command, Interpreter patterns.

Brief theoretical information

Memento

Problem. It is necessary to fix the state of the object to implement, for example, the rollback mechanism.

Decision. Capture and transfer (without breaking the encapsulation) its internal state outside the object so that it can be restored later in the object. "Memento" stores the internal state of the "Originator" object and denies access to itself to all other objects except the "Originator", which has access to all data to restore to the previous state. "Caretaker" can only transfer "Memento" to other objects. The "Originator" creates a "Memento" containing a snapshot of the current internal state and uses the "Memento" to restore the internal state. The "Caretaker" is responsible for maintaining "Memento", while not performing any operations on "Memento" or examining its internal contents. "Caretaker" requests "Memento" from "Originator", keeps it for a while, then returns it to "Originator". The structure of the pattern is shown in Fig.15.

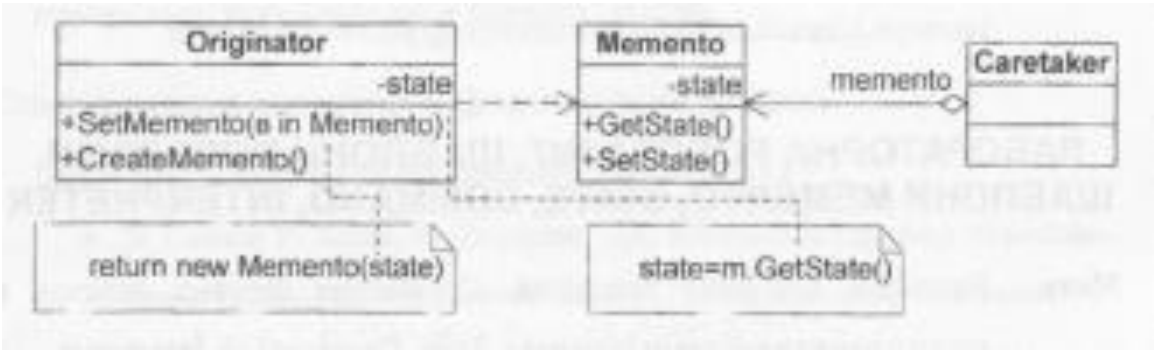

Fig. 15. The structure of the Memento pattern

The Memento pattern consists of the following parts:

- Memento (SolverState) - note, this component is intended for:
  - storage of the internal state of the Originator (creator) object. The amount of stored information can be different and is determined by the needs of its creator.
  - protection against access by all other objects except the creator. Memento actually has two interfaces: narrow and wide. The caretaker has a narrow interface to Memento, to enable the transfer of the memento to other objects. At the same time, the Originator (creator) has a wide interface to provide access to all the data necessary to restore its previous state. Only the creator who produced the memento is allowed access to the internal state of the Memento.
- Originator (ConstraintSolver) - author, creator, owner, this component is intended for:
  - creating a memento with a snapshot of the current internal state;
  - using a memento when restoring the internal state.
- Caretaker (undo mechanism) - messenger, this component is designed to implement the mechanism of cancellation and preservation of the memento; never operates with the contents of the memento and does not study its contents.

State

Problem. Vary the object's behavior depending on its internal state.

Decision. The Context class delegates state-dependent requests to the current ConcreteState object (stores an instance of the ConcreteState subclass that defines the current state) and defines an interface of interest to clients. "ConcreteState" implements the behavior associated with some state of the "Context" object. "State" defines an interface for encapsulating the behavior associated with a specific "Context" instance. The pattern localizes state-dependent behavior and divides it into parts corresponding to states, transitions between states become explicit. The structure of the pattern is shown in Fig.16.

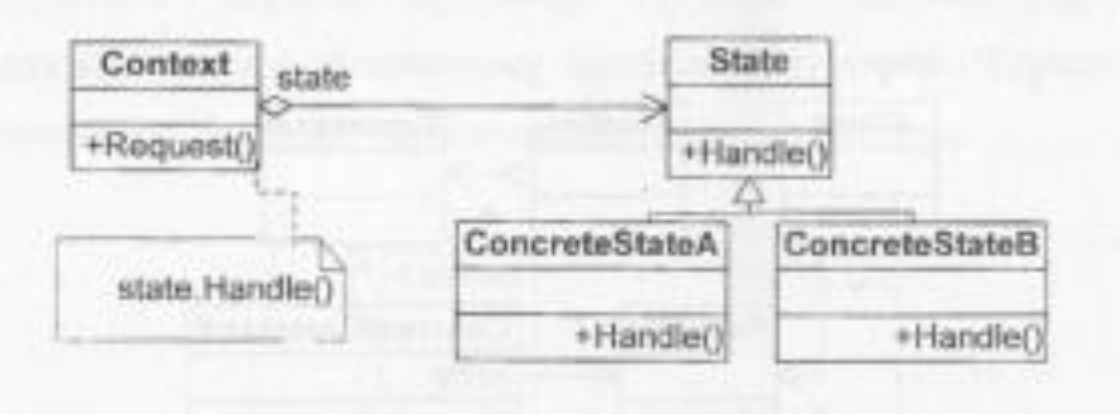

Fig. 16. The structure of the State pattern

The State pattern consists of the following parts:

- Context - this component is designed to define the interface of interest to clients and store an instance of the ConcreteState subclass that defines the current state.
- State - this component is intended to define an interface for encapsulating the behavior associated with a particular state of the context.
- ConcreteState subclasses - this component is intended for each subclass to implement a behavior that corresponds to a particular state of the Context.

Command

Problem. It is necessary to send a request to the object without knowing what operation is requested and who will be the recipient.

Decision. Encapsulate the request as an object. The "Client" creates a "ConcreteCommand" object that calls receiver operations to execute the request, the "Invoker" sends the request by executing the "Command" Execute() operation. "Command" declares the interface to perform the operation, "ConcreteCommand" defines the relationship between the "Receiver" object and the Action() operation, and in addition implements the Execute() operation by calling the corresponding operations of the "Receiver" object. "Client" creates an instance of the "ConcreteCommand" class and sets its receiver, "Invoker" calls the command to perform the request, "Receiver" (any class) has information about how to perform the operations necessary to perform the request. The structure of the pattern is shown in Fig.17.

The Command pattern consists of the following parts:

- Command - this component is intended to declare the operation execution interface.
- ConcreteCommand (PasteCommand, OpenCommand) - this component is designed to define the connection between the Receiver object (receiver) and the action, and also to implement the execution by calling the appropriate operations on the receiver.
- Client (Application) - this component is designed to create a ConcreteCommand object and set its receiver.
- Invoker (MenuItem) - call (initiator) , this component is designed to call a command to perform a request.
- Receiver (Document, Application) - recipient, this component is intended to perform an operation related to the execution of a request. Any class can be a receiver.

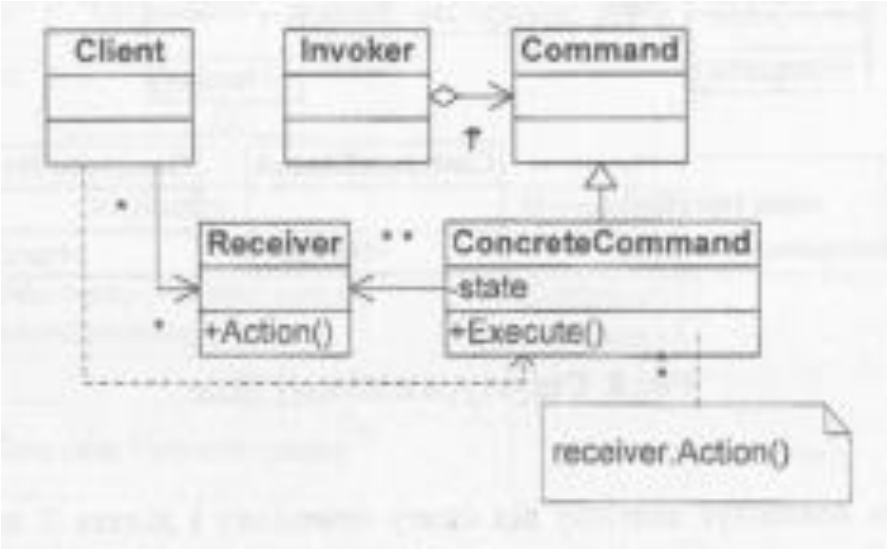

Fig. 17. The structure of the Command pattern

The Command pattern breaks the connection between the object that initiates the operation and the object that has information about how to execute it, and also creates a "Command" object that can be extended and manipulated as an object.

Interpreter

Problem. There is a changeable task that occurs frequently.

Decision. Create an interpreter that solves this problem.

The task of finding lines by pattern can be solved by creating an interpreter that determines the grammar of the language. "Client" builds a sentence in the form of an abstract syntactic tree, the nodes of which contain objects of classes "TerminalExpression" and "NonterminalExpression" (recursive), then "Client" initializes the context and calls the Interpret(Context) operation. On each node of the "TerminalExpression" type, the Interpret(Context) operation is implemented for each subexpression. For the "NonterminalExpression" class, the Interpret(Context) operation defines the recursion base. "AbstractExpression" defines an abstract Interpret(Context) operation common to all nodes in the abstract syntax tree. "Context" contains information global to the interpreter.

The structure of the pattern is shown in Fig.18.

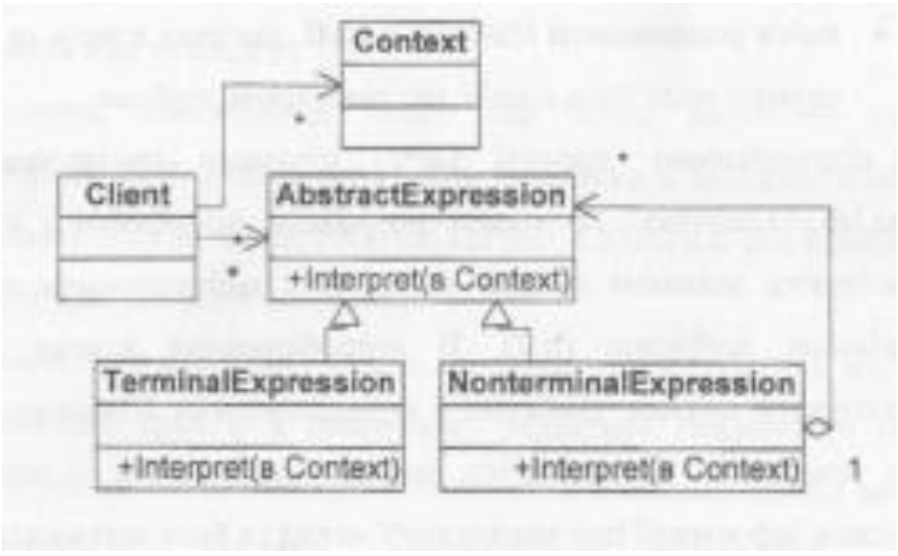

Fig. 18. The structure of the Interpreter pattern

The Interpreter pattern consists of the following parts:

- AbstractExpression (RegularExpression) - this component is intended to declare an abstract interpretation operation common to all nodes in the abstract syntax tree.
- TerminalExpression (LiteralExpression) - this component is designed to implement the Interpret operation associated with the final expression symbols in the grammar and to create the instance required for each final expression symbol in the sentence.
- NonterminalExpression (AlternationExpression, RepetitionExpression, SequenceExpressions) - this component is designed to:
  - display in the form of a corresponding class of each rule R :: = R1 R2 ... Rn in the grammar;
  - storage of variable instances of the AbstractExpression type for each of the expressions R1, ... , Rn;
  - implementation of the Interpret operation for non-finite expressions in the grammar; the interpretation can be recursive for the variables that represent the expressions R1, … , Rn..
- Context - this component is designed to store information that is common to the Interpreter.
- Client - this component is designed to construct (or obtain) an abstract syntax tree representing a particular sentence in the language defined by the grammar. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes; also this component is intended to call the Interpret operation.

Thanks to the pattern, the grammar becomes easy to extend and change, the implementations of the classes describing the nodes of the abstract syntactic tree are similar (easy to code). You can easily change the way expressions are evaluated. But it is difficult to follow a grammar with a large number of rules.

Tasks

1. Repeat patterns of behavior for software design. Know the general characteristics of behavior patterns and the purpose of each of them.
2. Study in detail the behavior patterns for software design - Memento, State, Command and Interpreter. For each of them:
   - study the pattern, its purpose, alternative names, motivation, cases when its use is appropriate and the results of such use;
   - know the peculiarities of the implementation of the pattern, related patterns, known cases of its application in software applications;
   - fluently master the structure of the pattern, the assignment of its classes and the relationships between them;
   - be able to recognize a pattern in the UML class diagram and build the source codes of Java classes that implement the pattern.
3. In the prepared project (LW1), create the program package com.lab111.labwork7. In the package, develop interfaces and classes that implement tasks (according to the option) using one or more patterns (item 2). In the developed classes, fully implement the methods related to the functioning of the Template. Methods that implement business logic should be closed with stubs that output information about the called method and its arguments to the console.
4. Complete documentation of the developed classes (including methods and fields) using automated means, while the documentation should sufficiently highlight the role of a certain class in the general structure of the pattern and the specifics of the specific implementation.

Task variants

The number of the task variant is calculated as the remainder of the division of the score book number by 8.

0. Define the specifications of classes that provide graphic elements (circle, triangle, etc.) with different attributes (color, position, size, etc.) in the vector editor. Implement a mechanism for saving/setting the state of the element.
1. Determine the specifications of the class that provides the character in the game space with the necessary attributes (position of the character, composition of artifacts, level of "health", etc.). Implement a mechanism for saving/setting the state of the character.

2. Define the specifications of the classes that provide operations on the table in the database. Implement a mechanism for organizing transactions when performing operations on the table.
3. Define the specifications of the class providing the network connection of the TCP protocol. Implement a change of behavior depending on the state of the connection (LISTENING, ESTABLISHED, CLOSED) without using cumbersome conditional operators.
4. Define the specifications of the classes providing the drawing tools and workspace in the graphic editor. Implement a mechanism for changing the reaction to clicking the mouse button depending on the selected tool. Avoid using cumbersome conditional constructions.
5. Define the specifications of the classes that provide a queue of HTTP requests for processing. Implement the possibility of excluding requests from the queue without processing, and changing the position of the request due to a change in the priority value.
6. Define the specifications of the classes providing reactions to pressing menu items and toolbar buttons. Provide the possibility of dynamic changes in the reaction, as well as the formation of macro-reactions (a sequence of predetermined reactions).
7. Define class specifications for parsing algebraic expressions with operations +, -, *, /.

Question for self-check

1. Classification of software design patterns.
2. Assignment of behavior patterns for software design.
3. A brief description of each pattern of behavior.
4. Names, purpose and motivation of the Memento pattern.
5. Structure of the Memento pattern and its members.
6. Peculiarities of implementing the Memento pattern. The result of using the pattern.
7. Names, purpose and motivation of the State pattern.
8. The structure of the State pattern and its members.
9. Features of the implementation of the State pattern. The result of using the pattern.
10. Names, purpose and motivation of the Command pattern.
11. Structure of the Command pattern and its members.
12. Features of the implementation of the Command pattern. The result of using the pattern.
13. Names, purpose and motivation of the Interpreter pattern.
14. Structure of the Interpreter pattern and its members.
15. Features of the implementation of the Interpreter pattern. The result of using the pattern.
16. Patterns used in conjunction with Memento, State, Command and Interpreter.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Design Patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Indianapolis: - Addison-Wesley, 1994. - 417 p. ISBN: 0201633612.
2. Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1, 2nd Edition. - Wiley Publishing, 2002. - 480 p. ISBN: 0471258393
3. Design Patterns. URL: https://en.wikipedia.org/wiki/Design_Patterns
4. Design Pattern – Overview. URL: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
5. Behavioral pattern - Wikipedia. URL: https://en.wikipedia.org/wiki/Behavioral_pattern#:~:text=In%20software%20engineering%2C%20behavioral%20design,flexibility%20in%20carrying%20out%20communication
6. Behavioral Design Patterns. URL: https://refactoring.guru/design-patterns/behavioral-patterns
7. Behavioral patterns. URL: https://sourcemaking.com/design_patterns/behavioral_patterns
8. Behavioral Design Patterns. URL: https://www.javatpoint.com/behavioral-design-patterns
9. Behavioral Design Patterns. URL: https://programmingline.com/software-design-patterns/behavioral-design-patterns
10. Behavioral Patterns. URL: https://howtodoinjava.com/design-patterns/behavioral/

LABORATORY WORK №8.
CREATIONAL DESIGN PATTERNS.
PATTERNS PROTOTYPE, SINGLETON, FACTORY METHOD

Theme

Creational design patterns. Patterns Prototype, Singleton, Factory Method.

Purpose

Learning creational design patterns. Getting basic skills in using Prototype, Singleton, Factory Method patterns.

Brief theoretical information

Prototype

Problem. The system should not depend on how objects are created, arranged and presented in it.

Decision. Create new objects using cloning. "Prototype" declares an interface for cloning itself. "Client" creates a new object by contacting "Prototype" with a request to clone "Prototype". The structure of the pattern is shown in Fig.19.

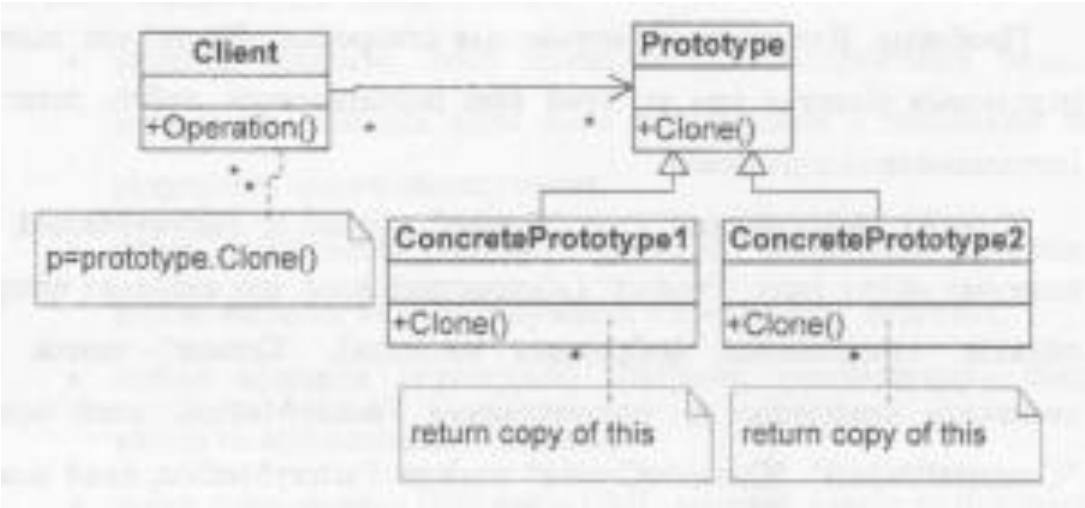

Fig. 19. The structure of the Prototype pattern

The Prototype pattern consists of the following parts:

- Prototype - this component is intended to declare an interface for cloning.
- ConcretePrototype - this component is designed to implement the operation of cloning itself.
- Client - this component is designed to create a new object by asking the prototype to clone itself.

Singleton

Problem. Only one instance of a custom class is needed, and different objects must access that instance through a single access point.

Decision. Create a class and define a static class method that returns this single object. The structure of the pattern is shown in Fig.20.

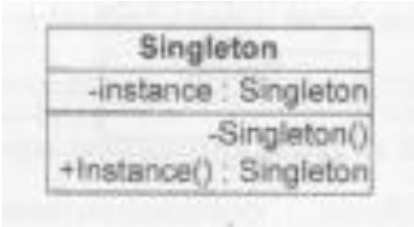

Fig. 20. The structure of the Singleton pattern

The Singleton pattern consists of the following parts:

- Singleton - this component is designed to define the Instance operation, which allows clients to access a single unique instance (Instance is a class operation), and to maintain responsibility for creating its own unique instance.

It is more rational to create a static instance of a special class, rather than to declare the necessary methods static, because when using instance methods, you can apply the mechanism of inheritance and create subclasses. Static methods in programming languages are not polymorphic and do not allow overlapping in derived classes. The solution based on the creation of an instance is more flexible, because later you may need not a single instance of the object, but several.

Factory Method

Problem. Define an interface for creating an object, but let subclasses decide which class to instantiate, that is, delegate instantiation to subclasses.

Decision. The abstract class "Creator" declares FactoryMethod, which returns an object of type "Product" (an abstract class that defines the interface of objects created by the factory method). "Creator" can also define a default implementation of FactoryMethod, which returns "ConcreteProduct". "ConcreteCreator" overrides FactoryMethod, which returns a "ConcreteProduct" object. "Creator" "relies" on its subclasses to implement FactoryMethod, which returns an object "ConcreteProduct". The structure of the pattern is shown in Fig.21.

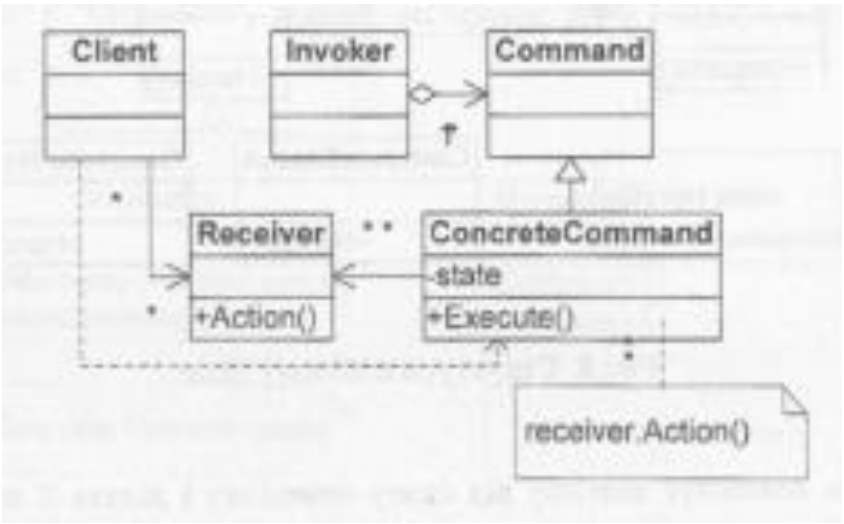

Fig. 21. The structure of the Factory Method pattern

The Factory Method pattern consists of the following parts:

- Product - this component is intended to define the interface of objects that are created by the factory method.
- ConcreteProduct - this component is designed to implement the Product interface.
- Creator - creator, this component is for:
    - declaring a factory method that returns an object of type Product; at the same time, the Creator may be given the opportunity to define a default implementation for the factory method that returns the ConcreteProduct object;
    - calling the factory method to create the Product object.

The pattern saves the designer from having to build application-dependent classes into the code. At the same time, an additional level of subclasses arises.

Tasks

1. To study reproductive patterns. To know the general characteristics of generating patterns and the purpose of each of them.
2. Study in detail the generative patterns - Prototype, Singleton and Factory Method. For each of them:
    - study the pattern, its purpose, alternative names, motivation, cases when its use is appropriate and the results of such use;
    - know the peculiarities of the implementation of the pattern, related patterns, known cases of its application in software applications;
    - fluently master the structure of the pattern, the assignment of its classes and the relationships between them;
    - to be able to recognize a pattern in the UML class diagram and build source Java-class codes that implement the pattern.
3. Create the program package com.lab111.labwork8 in the prepared project (LW1). In the package, develop interfaces and classes that implement tasks (according to the option) using one or more patterns (item 2). In the developed classes, fully implement the methods related to the functioning of the pattern. Methods that implement business logic should be closed with stubs that output information about the called method and its arguments to the console.
4. Complete documentation of the developed classes (including methods and fields) using automated means, while the documentation should sufficiently highlight the role of a certain class in the general structure of the pattern and the specifics of the specific implementation.

Task variants

The number of the task variant is calculated as the remainder of the division of the score book number by 11.

0. Define the specifications of the classes to represent the composite structure of the game space. Implement deep and surface cloning of such a structure.
1. Define class specifications and method implementation for the mechanism of cloning graphic elements in the vector graphics editor. Provide the possibility of both deep and surface cloning.
2. Define the specifications of the classes that provide graphic elements (primitives and their compositions) in the vector graphics editor. Implement a mechanism for cloning elements with a depth parameter.
3. Define class specifications for presenting the file system in the form of a tree of objects (a file is a leaf object, a directory is a node object). Implement a mechanism for cloning such objects with a depth parameter.
4. Define class specifications for representing the game manager, which consists of a game space and a list of game chips. Ensure the possibility of creating only one copy of the administrator.
5. Define the specifications of the classes for presenting the relational table, the database schema and the validator of requests to the table. Ensure the possibility of creating only one copy of the database schema.
6. Define class specifications for representing the composite structure of an algebraic expression, a variable map, and an expression calculator. Ensure the existence of only one variable map.

7. Define class specifications that encapsulate a linear list of objects and a forward and backward iterator for this structure.
8. Define class specifications that encapsulate a linear list of integers and a forward traversal iterator in an ordered structure.
9. Define class specifications for implementing the aggregate and its iterator, which implements the possibility of changing the search algorithm for the next element during program execution.
10. Define class specifications for the implementation of the composite and its iterators — for traversing the structure using depth-first (DFS) and breadth-first (BFS) search methods.

Question for self-check

1. Classification of software design patterns.
2. Designation of generative patterns.
3. A brief description of each generating pattern.
4. Names, purpose and motivation of the Prototype pattern.
5. The structure of the Prototype pattern and its participants.
6. Features of the implementation of the Prototype pattern. The result of using the pattern.
7. Singleton Pattern Names, Purpose, and Motivation.
8. The structure of the Singleton pattern and its members.
9. Features of Singleton pattern implementation. The result of using the pattern.
10. Names, purpose and motivation of the Factory Method pattern.
11. Structure of the Factory Method pattern and its members.
12. Features of the implementation of the Factory Method pattern. The result of using the pattern.
13. Patterns used in conjunction with Prototype, Singleton and Factory Method.
14. Deep and surface cloning.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Design Patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Indianapolis: - Addison-Wesley, 1994. - 417 p. ISBN: 0201633612.
2. Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1, 2nd Edition. - Wiley Publishing, 2002. - 480 p. ISBN: 0471258393
3. Design Patterns. URL: https://en.wikipedia.org/wiki/Design_Patterns
4. Design Pattern – Overview. URL: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
5. Creational pattern - Wikipedia. URL: https://en.wikipedia.org/wiki/Creational_pattern
6. Creational Design Pattern. URL: https://social.technet.microsoft.com/wiki/contents/articles/13211.creational-design-pattern.aspx
7. Creational patterns. URL: https://sourcemaking.com/design_patterns/creational_patterns
8. Introduction to Creational Design Patterns. URL: https://www.baeldung.com/creational-design-patterns
9. Creational Design Patterns. URL: https://www.gofpatterns.com/creational/index.php
10. Creational Patterns. URL: https://howtodoinjava.com/design-patterns/creational/

LABORATORY WORK №9.
CREATIONAL DESIGN PATTERNS.
PATTERNS ABSTRACT FACTORY, BUILDER

Theme
Creational design patterns. Patterns Abstract Factory, Builder.

Purpose
Learning creational design patterns. Getting basic skills in using Abstract Factory, Builder patterns.

Brief theoretical information

Abstract Factory

Problem. Create a family of interconnected or interdependent objects (without specifying their specific classes).

Decision. Create an abstract class that declares an interface for creating concrete classes.

A pattern isolates specific classes. Because "AbstractFactory" encapsulates the responsibility for creating classes and the process of creating them, it insulates the client from the implementation details of the classes. Simplified the replacement of ``AbstractFactory" because it is used in the application only once when instantiated.

It should be noted that the "AbstractFactory" interface fixes the set of objects that can be created. This makes it somewhat difficult to extend "AbstractFactory" to make new objects.

The structure of the pattern is shown in Fig.22.

The Abstract Factory pattern consists of the following parts:

- AbstractFactory - this component is intended to declare an interface for operations that create abstract product objects.
- ConcreteFactory - this component is designed to implement the operation of creating specific product objects.
- AbstractProduct - this component is intended to declare an interface for the product object type.
- ConcreteProduct - this component is intended to define a product object that must be created by the corresponding specific factory; is also intended to implement the AbstractProduct interface.
- Client - this component is intended to use only the interfaces that have been declared by the AbstractFactory and AbstractProduct classes.

Builder

Problem. To separate the construction of a complex object from its representation, so that the same construction can result in different representations. The algorithm for creating a complex object should not depend on what parts the object consists of and how they connect to each other.

Decision. "Client" creates the "Director" manager object and configures it with the "Builder" object. The "Director" tells the "Builder" that another part of the "Product" needs to be built. The "Builder" processes the requests of the "Director" and adds new parts to the "Product", then the "Client" takes the "Product" from the "Builder".

The "Builder" object provides the "Director" object with an abstract interface for constructing the "Product" behind which it can hide the presentation and internal structure of the product, and furthermore the process of building the "Product". To change the internal representation of "Product", it is sufficient to define a new view of "Builder". This pattern isolates the code that implements the creation of the object and its presentation. The structure of the pattern is shown in Fig.23.

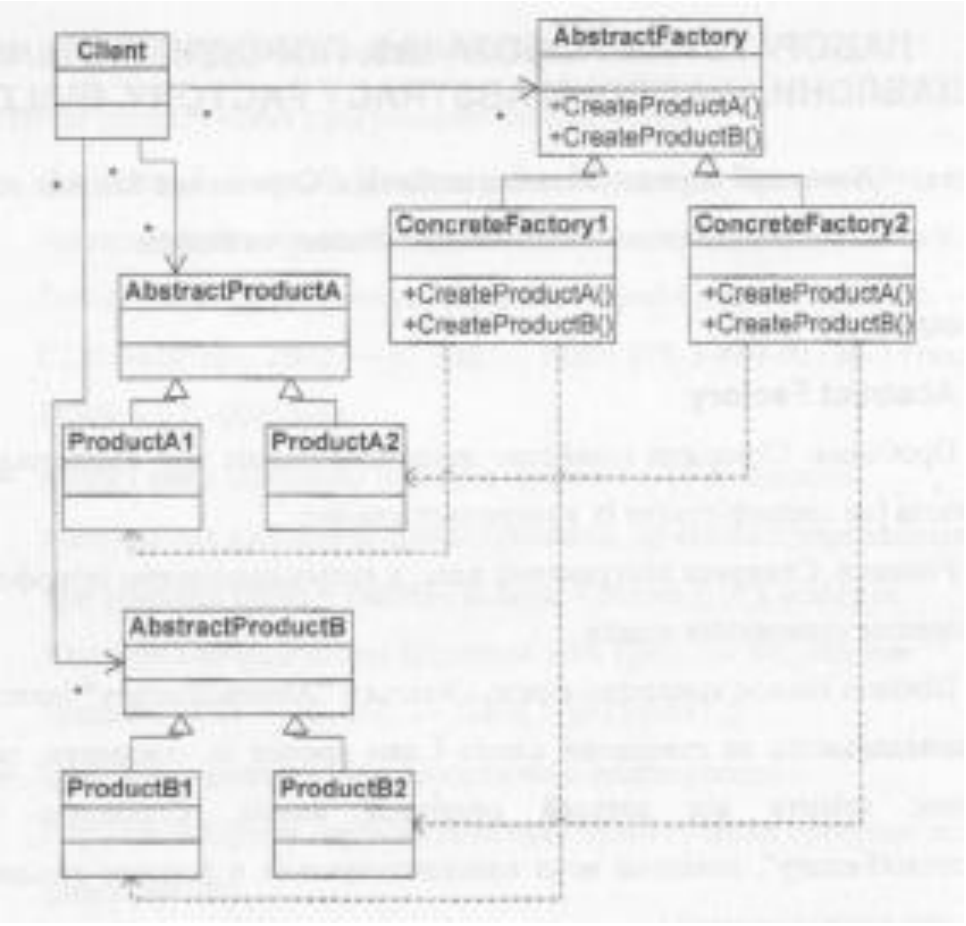

Fig. 22. The structure of the Abstract Factory pattern

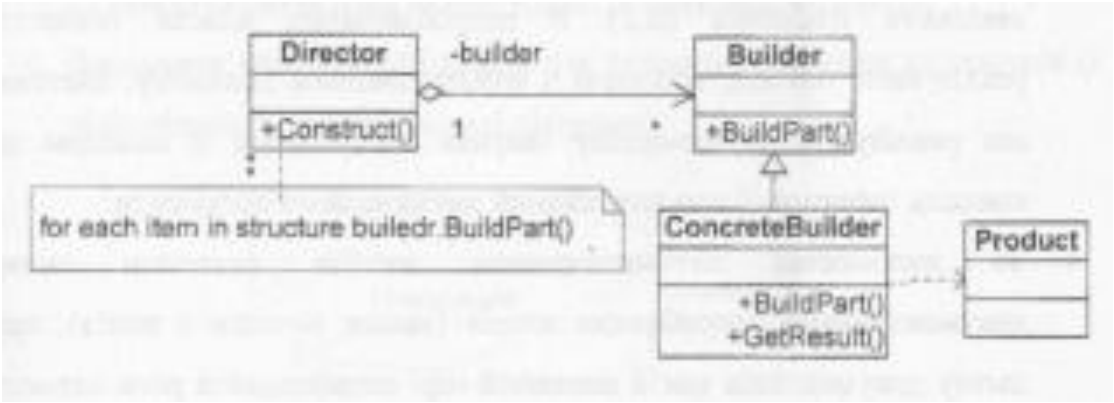

Fig. 23. The structure of the Builder pattern

The Builder pattern consists of the following parts:

- Builder - this component is intended to define an abstract interface for creating parts of a Product object.
- ConcreteBuilder - this component is designed to:
  - designing and assembling a bunch of product parts by implementing the Builder interface.
  - defining and tracking the view it creates.
  - providing an interface to access the product.
- Director - this component is designed to build an object using the Builder interface.
- Product - this component is designed to:
  - presentation of a complex object under construction; at the same time, ConcreteBuilder creates an internal representation of the product and defines the process of its assembly.
  - connecting the classes that define the component parts, including the interfaces for assembling the parts into the final result.

Tasks

1. Repeat the generative patterns. To know the general characteristics of generating patterns and the purpose of each of them.
2. Study the generative patterns in detail - Abstract Factory and Builder. For each of them:
   - study the pattern, its purpose, alternative names, motivation, cases when its use is appropriate and the results of such use;
   - know the peculiarities of the implementation of the Template, related patterns, known cases of its application in software applications;
   - fluently master the structure of the Template, the assignment of its classes and the relationships between them;
   - be able to recognize a pattern in the UML class diagram and build the raw codes of Java classes that implement the pattern.
3. In the prepared project (LW1), create the com.lab111.labwork9 software package. In the package, develop interfaces and classes that implement tasks (according to the option) using one or more patterns (item 2). In the developed classes, fully implement the methods related to the functioning of the pattern. Methods that implement business logic should be closed with stubs that output information about the called method and its arguments to the console.
4. Complete documentation of the developed classes (including methods and fields) using automated means, while the documentation should sufficiently highlight the role of a certain class in the general structure of the pattern and the specifics of the specific implementation.

Task variants

The number of the task variant is calculated as the remainder of the division of the score book number by 11.

0. Define class specifications for presenting a family of GUI widgets implemented on different APIs (WinAPI, GTK). Provide the possibility of client-transparent extension of the implementation for other APIs (Qt, OSX).
1. 1. Define class specifications for presenting a family of tools for working with object data through various APIs (DB, File). Provide the possibility of a client-transparent extension of the implementation for other APIs (WebService).
2. Define class specifications for presenting a family of universal interactive development environment tools (Validator, Compiler, Debugger) with their implementation for different languages (Java, C++). To provide the possibility of client-transparent extension of implementation for languages (ObjectPascal).
3. Define the class specifications for the rectangular game space and the loader of its configuration from an external file.
4. Define class specifications for presenting block diagrams of algorithms (according to the semantic diagram Fig. 24) and its loader from an external file.

Fig. 24. Semantic diagram of the algorithm block diagrams

5. Determine the class specifications for the parse tree builder of a complex expression (according to the Backus-Naur Form) based on its symbolic representation.

   <expression>::=<simple expression> | <complex expression>
   <simple expression>::=<constant> | <variable>
   <constant>::=(<number>)
   <variable>::=(<name>)
   <complex expression>::=(<expression><operation sign><expression>)
   <operation sign>::=+|-|*|/

6. Define class specifications for record representation, relational table and its loader from an external file.
7. Define the specifications of the classes for presenting the relational table and the builder of the direct product of the tables.
8. Define the specifications of the classes for presenting the relational table and the table projection builder.
9. Define class specifications for relational table representation, database schema, and appropriate loader. Ensure the possibility of creating only one copy of the database schema.
10. Define the specifications of the classes for presenting the elements of the vector graphic editor (primitive and composite). Implement the possibility of building a composite image based on the downloaded specification file.

Question for self-check

1. Classification of software design patterns.
2. Designation of generative patterns.
3. A brief description of each generating pattern.
4. Names, purpose and motivation of the Abstract Factory pattern.
5. The structure of the Abstract Factory pattern and its members.
6. Features of the implementation of the Abstract Factory pattern. The result of using the pattern.
7. Names, purpose and motivation of the Builder pattern.
8. The structure of the Builder pattern and its members.
9. Features of the implementation of the Builder pattern. The result of using the pattern.
10. Patterns used in conjunction with Abstract Factory and Builder.

Protocol

The protocol must contain the title page (with the number of the score book), tasks, a printout of the class diagram, developed program code, generated documentation in JavaDoc format and the results of program testing. Source code files (*.java) must be added to the protocol.

Recommended references

1. Design Patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Indianapolis: - Addison-Wesley, 1994. - 417 p. ISBN: 0201633612.
2. Mark Grand. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1, 2nd Edition. - Wiley Publishing, 2002. - 480 p. ISBN: 0471258393
3. Design Patterns. URL: https://en.wikipedia.org/wiki/Design_Patterns
4. Design Pattern – Overview. URL: https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
5. Creational pattern - Wikipedia. URL: https://en.wikipedia.org/wiki/Creational_pattern
6. Creational Design Pattern. URL: https://social.technet.microsoft.com/wiki/contents/articles/13211.creational-design-pattern.aspx
7. Creational patterns. URL: https://sourcemaking.com/design_patterns/creational_patterns
8. Introduction to Creational Design Patterns. URL: https://www.baeldung.com/creational-design-patterns
9. Creational Design Patterns. URL: https://www.gofpatterns.com/creational/index.php
10. Creational Patterns. URL: https://howtodoinjava.com/design-patterns/creational/