TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Jekaterina Vassiljeva 164947

ANALYSIS AND SELECTION OF BEST CASE TOOLS BASED ON REVERSE ENGINEERING AND PATTERNS DETECTION FOR JAVA PROJECTS

Bachelor's thesis

Supervisor: Tarmo Veskioja
PhD
Research Scientist

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jekaterina Vassiljeva 164947

REVERSE ENGINEERINGU JA MUSTRITE TUVASTAMISE PÕHJAL CASE VAHENDITE VÕRDLEV ANALÜÜS JA PARIMA VÄLJAVALIMINE JAVA PROJEKTIDE NÄITEL

bakalaureusetöö

Juhendaja: Tarmo Veskioja
Tehnikateaduste doktor

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Jekaterina Vassiljeva

18.05.2018

Abstract

The goal of this thesis is to analyse reverse engineering abilities of three selected CASE tools for Java-based projects. Selected CASE tools are Enterprise Architect 13.0, Visual paradigm Standard edition 15.0 and The ObjectAid UML Explorer for Eclipse IDE v1.2.2.

The analysis is done for two main reverse engineering capabilities: reverse engineering to class diagrams and reverse engineering to sequence diagrams.

Analysis of reverse engineering to class diagrams is based on twelve GOF design patterns and contains class diagrams for the same set of classes created in each selected CASE tool. CASE tools should be able to reverse engineer classes, find all relationships between them and represent them similarly to default GOF design patterns representation described in "Design Patterns: Elements of Reusable Object-Oriented Software" book [1].

Analysis of reverse engineering to sequence diagram is based on reverse engineering of one class method in each selected CASE tool. CASE tools should be able to reverse engineer the code steps of this class method to a sequence diagram, find all messages sent by this method and represent them correctly.

As a result of this analysis, the best CASE tool (among the compared CASE tools) for the UML model-based documentation of legacy code for the investigated Java-based projects will be identified.

This thesis is written in English and is 84 pages long, including 6 chapters, 59 figures and 5 tables.

Annotatsioon

Reverse engineeringu ja mustrite tuvastamise põhjal CASE vahendite võrdlev analüüs ja parima väljavalimine Java projektide näitel

Käesoleva bakalaureusetöö eesmärgiks on analüüsida kolme valitud CASE vahendi pöördprojekteerimise võimalusi Java projekti näitel. Valitud CASE vahendid on Enterprise Architect 13.0, Visual paradigm Standard edition 15.0 ja The ObjectAid UML Explorer for Eclipse IDE v1.2.2.

Analüüsis on CASE vahendeid võrreldud kahe peamise võimekuse alusel: klassidiagrammi pöördprojekteerimine ja jadadiagrammi pöördprojekteerimine.

Klassidiagrammi pöördprojekteerimise analüüsis keskenduti kaksteistkümnele GOF disaini mustritele, iga CASE vahendiga pöördprojekteerimise tulemus peaks sisaldama klassdiagramme sama klasside hulga kohta. CASE vahendid peavad pöördprojekteerima klassid, leidma kõik seosed nende vahel ja esitama need sarnaselt standardse GOF disaini mustrite esitusega, mis on kirjeldatud raamatus „Design Patterns: Elements of Reusable Object-Oriented Software”.

Jadadiagrammi pöördprojekteerimise analüüs põhineb ühe klassi üksiku meetodi pöördprojekteerimisel kõigis võrreldud CASE vahendites. CASE vahendid peavad pöördprojekteerima meetodi sammud jadadiagrammile, leidma kõik saadetud sõnumid ja kujutama neid korrektselt.

Selle analüüsi tulemusena tuvastatakse (võrdlusse kaasatud CASE vahenditest) parim CASE vahend Java-põhise projekti koodi UML mudeli-põhise dokumentatsiooni tegemiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 84 leheküljel, 6 peatükki, 59 joonist, 5 tabelit.

List of abbreviations and terms

CASE	Computer-aided software engineering
IDE	Integrated development environment
EA	Enterprise Architect [2]
UML	Unified Modeling Language [3]
GOF	Gang of Four. The authors of the "Design Patterns: Elements of Reusable Object-Oriented Software" book [1] came to be known as the "Gang of Four”. [4]
FAS	Feedback Arc Set [5]
Reverse engineering	The process of analysing a subject system to identify the system’s component and their interrelationships and create representation of the system in another form or at a higher level of abstraction. [6]
Design pattern	Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. [1]

Table of contents

List of figures

List of tables

1 Introduction

Nowadays fast and agile development process is a requirement for most enterprises. This requirement often does not allow developers to spend much time on system design and documentation. As a result, knowledge of the system remains in developer's heads. If developer leaves the company or project goes to maintenance to another team, then knowledge gets lost [7].

These forces developers sometimes to slow down and invest time into the documentation of existing codebase. CASE tools with reverse engineering capabilities can help to speed up this process. The author of this thesis did not encounter automatic reverse engineering of code as part of the courses she took during the bachelor studies. Therefore this was an additional incentive to further study reverse engineering of code in this thesis.

This thesis does not contain all possible alternatives for CASE tools and all design patterns. It contains a proposed approach to analyse and evaluate CASE tools reverse engineering capabilities. As a result, it helps to select the best CASE tool for reverse engineering of Java code from selected CASE tools.

There are various CASE tools with different capabilities. For analysis and research, three modern CASE tools were selected. Each of them can be used for documentation of legacy code for Java projects. They are Enterprise Architect 13.0 [2], Visual paradigm Standard edition [8] and The ObjectAid UML Explorer for Eclipse IDE [9].

The goal of this thesis is to identify which of these three CASE tools fits best for documentation of Java-based projects. Two parameters will be used for comparison.

1. CASE tool should be able to represent GOF design patterns on UML class diagrams. This representation should be similar to standard GOF design patterns presentation. For classes used to represent GOF design pattern reverse engineering CASE tool should be able to identify all classes, parameters and methods of this classes and relationships between them.

GOF patterns that will be analysed are twelve out of twenty-three GOF design patterns.

   1. Singleton
   2. Factory
   3. Observer
   4. Strategy
   5. Iterator
   6. Bridge
   7. Adapter
   8. Composite
   9. Template method
   10. Command
   11. Builder
   12. State

2. The CASE tool should be able to create sequence diagrams using reverse engineering capability, find all messages sent in the method and represent all messages correctly.

2 Methodology

The first part of the thesis contains an analysis of class diagrams generated by each CASE tool for each design pattern out of twelve predefined GOF design patterns.

Analysis of each design pattern will follow the same steps. The first step of this thesis describes existing GOF design patterns definitions and its standard UML representation based on the book “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides [1]. The second step describes java project per design pattern used for reverse engineering and describes classes that represent the GOF pattern. The third step contains an analysis of UML representation and UML diagrams generated by each CASE tool.

The next part of the thesis contains an analysis of sequence diagrams generated by reverse engineering. One sequence diagram will be created for the same class method using each CASE tool. The UML representation of generated sequence diagram by each CASE tool will then be analysed.

Last part will summarise results of previous parts and rank the CASE tools. If the unequivocally best CASE tool cannot be identified, then summarise the relative dominance of CASE tools based on subsets of features.

Project with reverse engineered UML diagrams for all CASE tools will be uploaded to [10].

3 Analysis of reverse engineering process: from java code to UML class diagram

In this chapter, UML representation of twelve GOF design patterns is compared in three CASE tools Enterprise Architect 13.5, Visual Paradigm 15.0 Standard edition and ObjectAid UML Explorer for Eclipse IDE v1.2.2.

A subset of GOF design patterns used for analysis was based on two reasons. The first reason is that all three types (creational, behavioural, and structural) of GOF design patterns should be represented. Second is based subjective opinion of the author about the importance of selected GOF design patterns over others.

The author reverses engineer different java projects and extract elements that represent different GOF design patterns to UML class diagrams and analyse these diagrams. Projects used in the thesis:

1. TestNG testing framework, v6.14.3 [11]
2. “Design patterns implemented in Java” project in GitHub [12] particularly Singleton [13], Adapter [14] and Command [15]
3. Two java applications that author wrote as an exercise for one of the university courses [16]. Code: [17] [18]

To simplify comparison grade from zero to three will be assigned to each CASE tool for each design pattern. Following grading rules will be used:

0 – could not identify any parts of a design pattern

1 – more than one expected part is missing

2 – identify almost everything except one part

3 – identify all parts of a design pattern

3.1 Singleton

3.1.1 Pattern definition

Singleton is a creational pattern that ensures a class only has one instance, and provide a global point of access to it. [1]

Singleton contains a private constructor, private static property that stores class instance and public static method to access this instance.

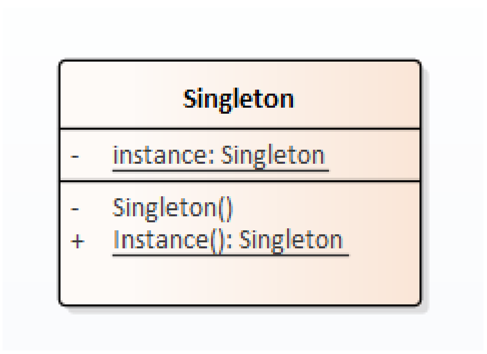


Diagram 1. Singleton pattern

3.1.2 Java project description

For Singleton design pattern analysis, singleton implementations from “Design patterns implemented in Java” project in GitHub is used [12]. This project contains example implementation for many design patterns. It also explains different ways of how the same design pattern can be implemented.

For Singleton pattern, this project contains five implementations of the ivory tower. An ivory tower is a place where wizards can study magic. Only one ivory tower can exist. All wizards should use the same ivory tower.

Three implementations are close to standard singleton described in 3.1.1. They have private static property that stores instance of the class and public static method to retrieve an instance of the class. Difference between them is in the implementation of the constructor of the class and validations that are done during getInstance() call.

One implementation is based on particular java data type called enum. It does not have a private static instance and a public method to access this instance, because enum by itself is constant.

The last implementation has a public static method to access instance, but it does not store instance as the property of singleton class. It is using additional private static HelperHolder class to store instance.

3.1.3 Pattern UML representation analysis

For each singleton implementation, all three CASE tools have identified property that store instances as private and static correctly and marked it with underline and minus sign. Minus sign means that property has private access level. Underline means that it is static.

The way how CASE tools represent instance property is different for each CASE tool. Enterprise architect shows it as part of the class Rectangle. ObjectAid UML Explorer shows it as a unidirectional self-association with 0 or 1 multiplicity and label with the property name. Visual paradigm combines both approaches. It has a property with singleton type in a class rectangle and unidirectional self-association but without multiplicity identifier.

All three CASE tools correctly identified public static methods to access private instance as well and marked them with underline and plus sign. Plus sign means that method has public access level. All CASE tools get 3 points.

Other noticeable difference between CASE tools is a way of “part-of” relationship representation. Visual parading and The ObjectAid UML Explorer use containment association to mark that HelperHolder class is part of InitializingOnDemandHolderIdiom class. Enterprise Architect changes the class name to format OuterClass::NestedClass to show the same relationship.

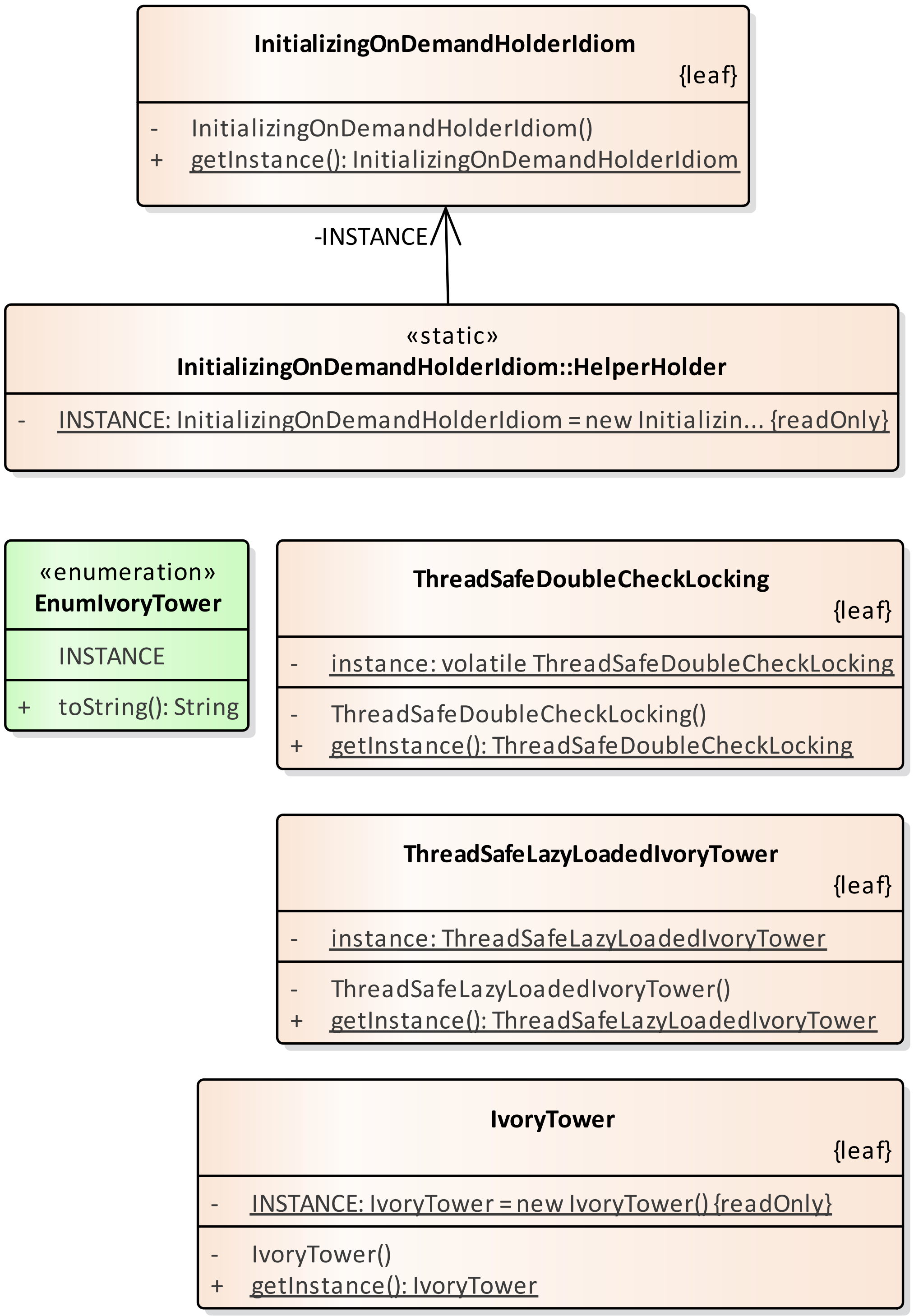

InitializingOnDemandHolderIdiom
{leaf}
- InitializingOnDemandHolderIdiom()
+ getInstance(): InitializingOnDemandHolderIdiom
-INSTANCE
«static»
InitializingOnDemandHolderIdiom::HelperHolder
- INSTANCE: InitializingOnDemandHolderIdiom = new Initializin... {readOnly}
«enumeration»
EnumIvoryTower
INSTANCE
+ toString(): String
ThreadSafeDoubleCheckLocking
{leaf}
- instance: volatile ThreadSafeDoubleCheckLocking
- ThreadSafeDoubleCheckLocking()
+ getInstance(): ThreadSafeDoubleCheckLocking
ThreadSafeLazyLoadedIvoryTower
{leaf}
- instance: ThreadSafeLazyLoadedIvoryTower
- ThreadSafeLazyLoadedIvoryTower()
+ getInstance(): ThreadSafeLazyLoadedIvoryTower
IvoryTower
{leaf}
- INSTANCE: IvoryTower = new IvoryTower() {readOnly}
- IvoryTower()
+ getInstance(): IvoryTower

Diagram 2. Singleton. Enterprise Architect

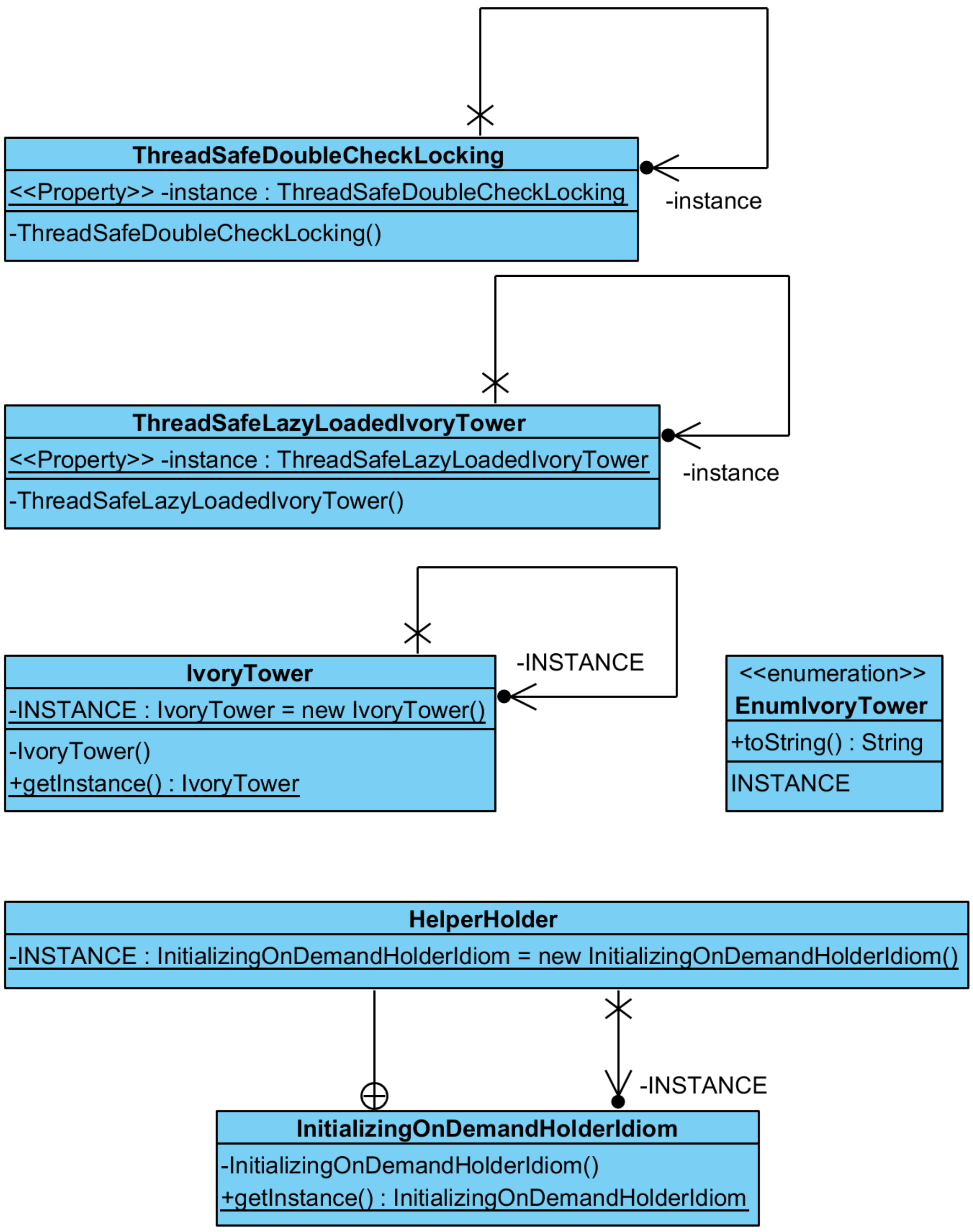

ThreadSafeDoubleCheckLocking
<<Property>> -instance : ThreadSafeDoubleCheckLocking
-ThreadSafeDoubleCheckLocking()
-instance
ThreadSafeLazyLoadedIvoryTower
<<Property>> -instance : ThreadSafeLazyLoadedIvoryTower
-ThreadSafeLazyLoadedIvoryTower()
-instance
IvoryTower
-INSTANCE : IvoryTower = new IvoryTower()
-IvoryTower()
+getInstance() : IvoryTower
-INSTANCE
<<enumeration>>
EnumIvoryTower
+toString() : String
INSTANCE
HelperHolder
-INSTANCE : InitializingOnDemandHolderIdiom = new InitializingOnDemandHolderIdiom()
-INSTANCE
InitializingOnDemandHolderIdiom
-InitializingOnDemandHolderIdiom()
+getInstance() : InitializingOnDemandHolderIdiom

Diagram 3. Singleton. Visual paradigm

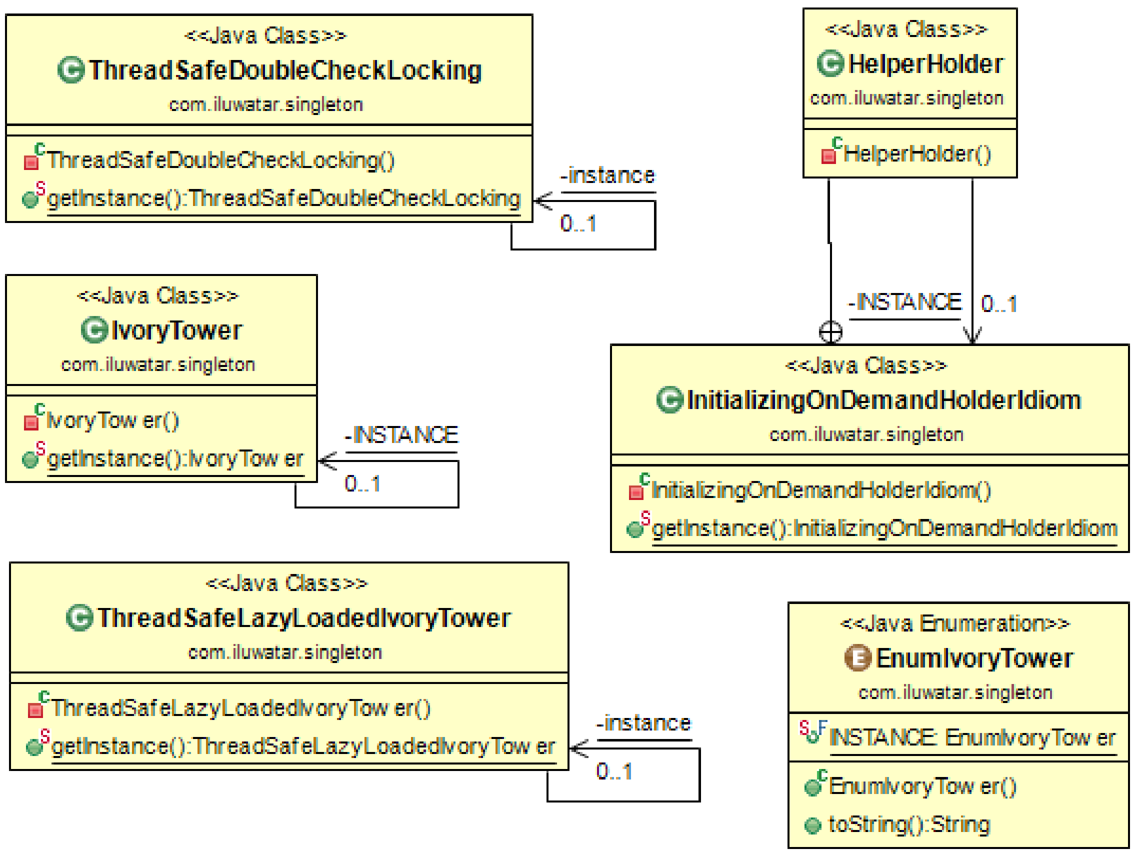


Diagram 4. Singleton. The ObjectAid UML Explorer for Eclipse

3.2 Factory method

3.2.1 Pattern definition

The factory method is a creational pattern. It is used Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. [1]

The pattern contains factory abstract class or interface, the concrete implementation of the factory class, product abstract class or interface and concrete implementation of the product class.

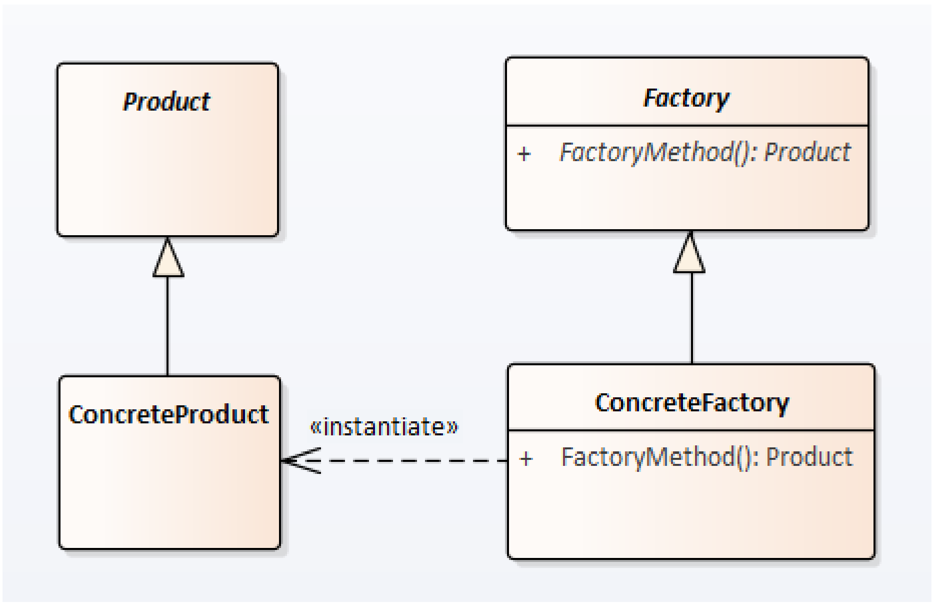


Diagram 5. Factory method pattern

3.2.2 Java project description

For factory method analysis I use TestNG test framework project [11].

TestNG project one of the implementations of factory method design pattern is DefaultListenerFactory class.

DefaultListenerFactory is concrete class. It implements an interface called ITestNGListenerFactory. Factory method name is createListener. This method can create an instance of any class that extends the interface of ITestNGListener.

TestNG have many interfaces that extend ITestNGListener and realisation for some of them. Not all classes that extend ITestNGListener are shown in the diagrams.

3.2.3 Pattern UML representation analysis

All three CASE tools found inheritance dependency between ITestNGListenerFactory interface and DefaultListenerFactory class. For this interface and class, CASE tools also found dependency relationships with ITestNGListener.

None of the CASE tools displays dependency association between factory interface or realisation and concrete implementation of ITestNGListener and gets 2 points for this pattern representation.

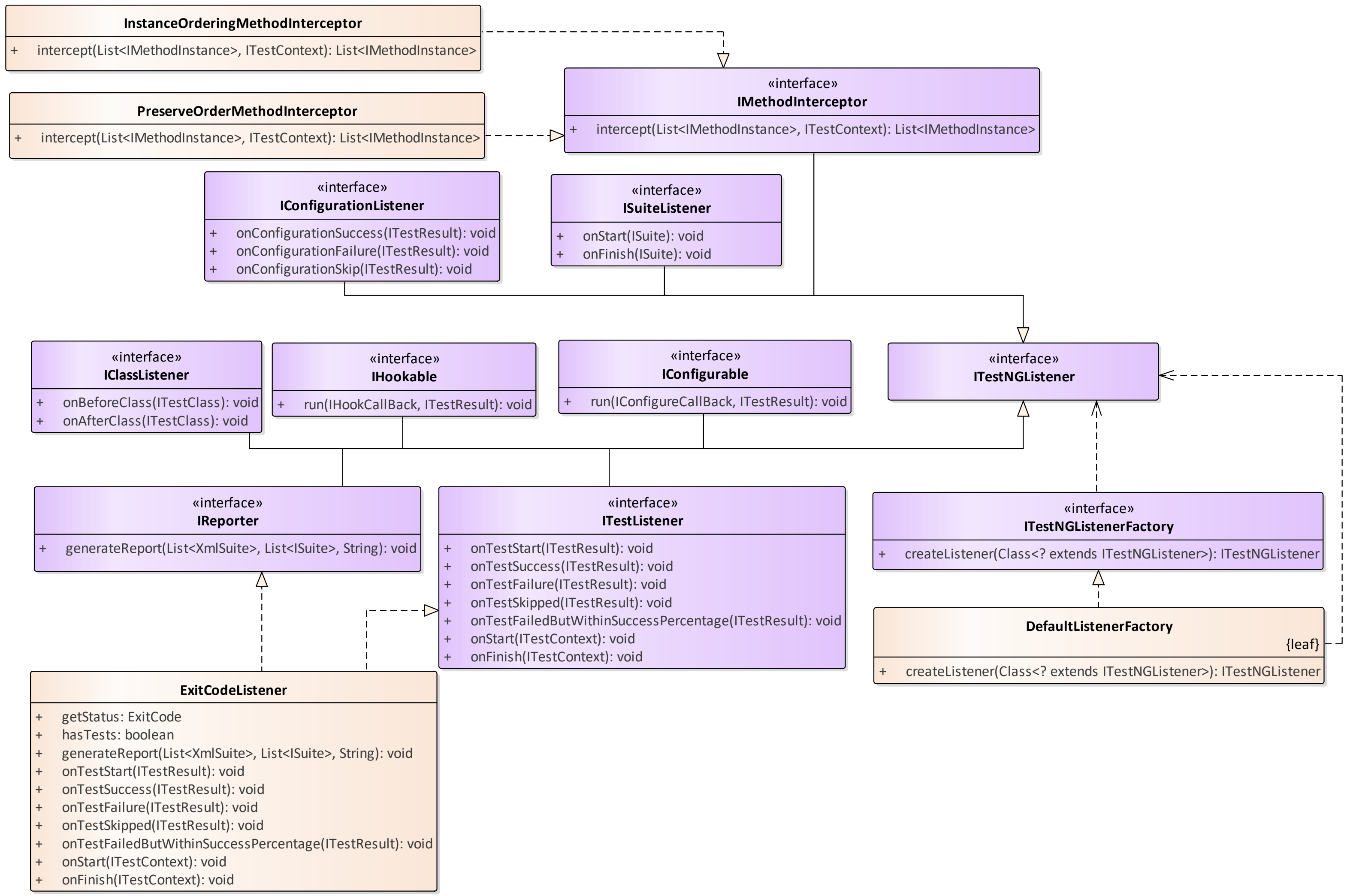

InstanceOrderingMethodInterceptor
+ intercept(List<IMethodInstance>, ITestContext): List<IMethodInstance>
PreserveOrderMethodInterceptor
+ intercept(List<IMethodInstance>, ITestContext): List<IMethodInstance>
«interface» IMethodInterceptor
+ intercept(List<IMethodInstance>, ITestContext): List<IMethodInstance>
«interface» IConfigurationListener
+ onConfigurationSuccess(ITestResult): void
+ onConfigurationFailure(ITestResult): void
+ onConfigurationSkip(ITestResult): void
«interface» ISuiteListener
+ onStart(ISuite): void
+ onFinish(ISuite): void
«interface» IClassListener
+ onBeforeClass(ITestClass): void
+ onAfterClass(ITestClass): void
«interface» IHookable
+ run(IHookCallBack, ITestResult): void
«interface» IConfigurable
+ run(IConfigureCallBack, ITestResult): void
«interface» ITestNGListener
«interface» IReporter
+ generateReport(List<XmlSuite>, List<ISuite>, String): void
«interface» ITestListener
+ onTestStart(ITestResult): void
+ onTestSuccess(ITestResult): void
+ onTestFailure(ITestResult): void
+ onTestSkipped(ITestResult): void
+ onTestFailedButWithinSuccessPercentage(ITestResult): void
+ onStart(ITestContext): void
+ onFinish(ITestContext): void
«interface» ITestNGListenerFactory
+ createListener(Class<? extends ITestNGListener>): ITestNGListener
DefaultListenerFactory
{leaf}
+ createListener(Class<? extends ITestNGListener>): ITestNGListener
ExitCodeListener
+ getStatus: ExitCode
+ hasTests: boolean
+ generateReport(List<XmlSuite>, List<ISuite>, String): void
+ onTestStart(ITestResult): void
+ onTestSuccess(ITestResult): void
+ onTestFailure(ITestResult): void
+ onTestSkipped(ITestResult): void
+ onTestFailedButWithinSuccessPercentage(ITestResult): void
+ onStart(ITestContext): void
+ onFinish(ITestContext): void

Diagram 6. Factory method. Enterprise Architect

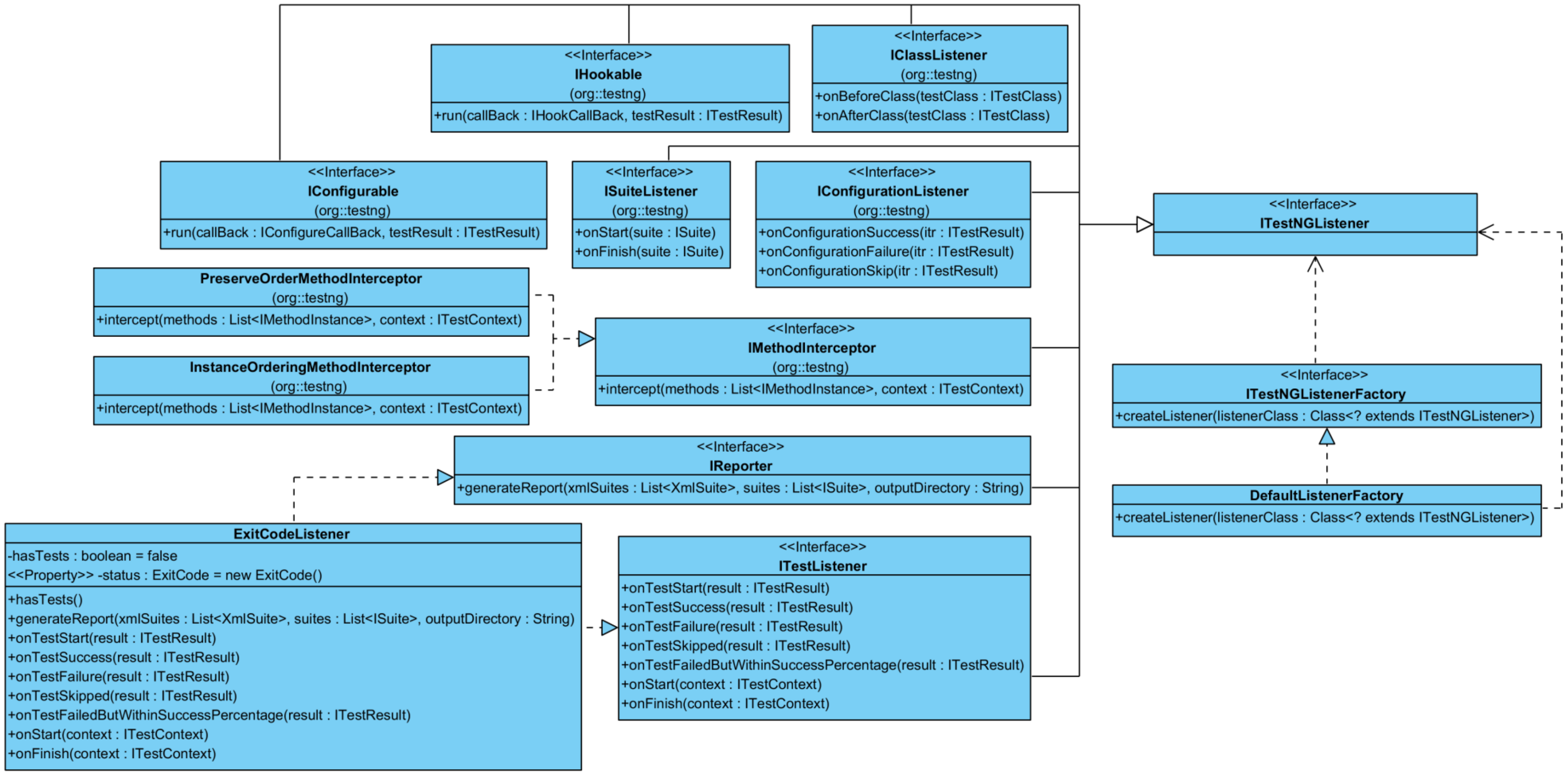
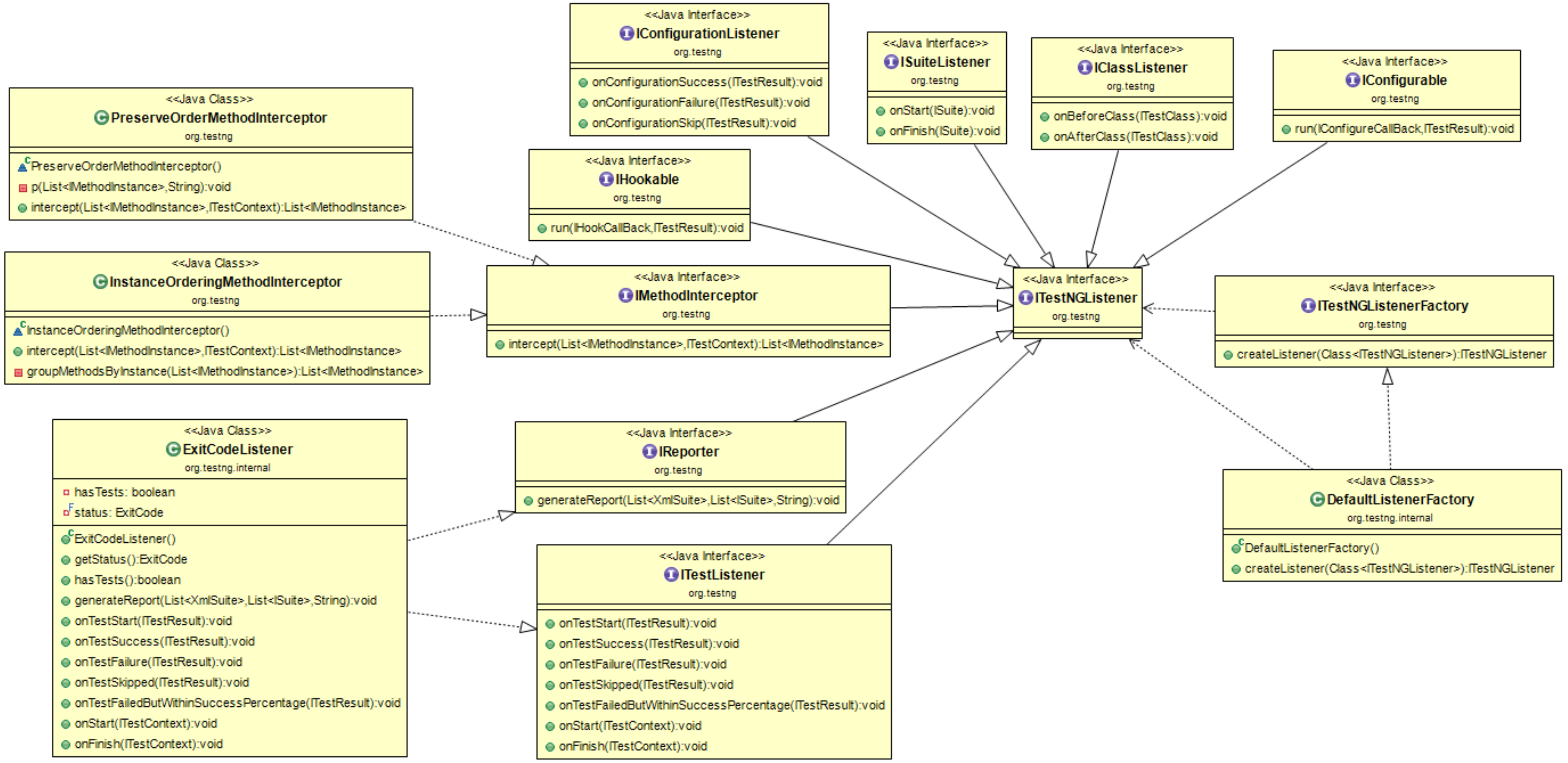

<<Interface>>
IHookable
(org::testng)
+run(callBack : IHookCallBack, testResult : ITestResult)
<<Interface>>
IClassListener
(org::testng)
+onBeforeClass(testClass : ITestClass)
+onAfterClass(testClass : ITestClass)
<<Interface>>
IConfigurable
(org::testng)
+run(callBack : IConfigureCallBack, testResult : ITestResult)
<<Interface>>
ISuiteListener
(org::testng)
+onStart(suite : ISuite)
+onFinish(suite : ISuite)
<<Interface>>
IConfigurationListener
(org::testng)
+onConfigurationSuccess(itr : ITestResult)
+onConfigurationFailure(itr : ITestResult)
+onConfigurationSkip(itr : ITestResult)
<<Interface>>
ITestNGListener
PreserveOrderMethodInterceptor
(org::testng)
+intercept(methods : List<IMethodInstance>, context : ITestContext)
InstanceOrderingMethodInterceptor
(org::testng)
+intercept(methods : List<IMethodInstance>, context : ITestContext)
<<Interface>>
IMethodInterceptor
(org::testng)
+intercept(methods : List<IMethodInstance>, context : ITestContext)
<<Interface>>
ITestNGListenerFactory
+createListener(listenerClass : Class<? extends ITestNGListener>)
<<Interface>>
IReporter
+generateReport(xmlSuites : List<XmlSuite>, suites : List<ISuite>, outputDirectory : String)
DefaultListenerFactory
+createListener(listenerClass : Class<? extends ITestNGListener>)
ExitCodeListener
-hasTests : boolean = false
<<Property>> -status : ExitCode = new ExitCode()
+hasTests()
+generateReport(xmlSuites : List<XmlSuite>, suites : List<ISuite>, outputDirectory : String)
+onTestStart(result : ITestResult)
+onTestSuccess(result : ITestResult)
+onTestFailure(result : ITestResult)
+onTestSkipped(result : ITestResult)
+onTestFailedButWithinSuccessPercentage(result : ITestResult)
+onStart(context : ITestContext)
+onFinish(context : ITestContext)
<<Interface>>
ITestListener
+onTestStart(result : ITestResult)
+onTestSuccess(result : ITestResult)
+onTestFailure(result : ITestResult)
+onTestSkipped(result : ITestResult)
+onTestFailedButWithinSuccessPercentage(result : ITestResult)
+onStart(context : ITestContext)
+onFinish(context : ITestContext)

Diagram 7. Factory method, Visual Paradigm



<<Java Interface>>
IConfigurationListener
org.testng
onConfigurationSuccess(ITestResult):void
onConfigurationFailure(ITestResult):void
onConfigurationSkip(ITestResult):void
<<Java Interface>>
ISuiteListener
org.testng
onStart(ISuite):void
onFinish(ISuite):void
<<Java Interface>>
IClassListener
org.testng
onBeforeClass(ITestClass):void
onAfterClass(ITestClass):void
<<Java Interface>>
IConfigurable
org.testng
run(IConfigureCallBack,ITestResult):void
<<Java Class>>
PreserveOrderMethodInterceptor
org.testng
PreserveOrderMethodInterceptor()
p(List<IMethodInstance>,String):void
intercept(List<IMethodInstance>,ITestContext):List<IMethodInstance>
<<Java Interface>>
IHookable
org.testng
run(IHookCallBack,ITestResult):void
<<Java Class>>
InstanceOrderingMethodInterceptor
org.testng
InstanceOrderingMethodInterceptor()
intercept(List<IMethodInstance>,ITestContext):List<IMethodInstance>
groupMethodsByInstance(List<IMethodInstance>):List<IMethodInstance>
<<Java Interface>>
IMethodInterceptor
org.testng
intercept(List<IMethodInstance>,ITestContext):List<IMethodInstance>
<<Java Interface>>
ITestNGListener
org.testng
<<Java Interface>>
ITestNGListenerFactory
org.testng
createListener(Class<ITestNGListener>):ITestNGListener
<<Java Class>>
ExitCodeListener
org.testng.internal
hasTests: boolean
status: ExitCode
ExitCodeListener()
getStatus():ExitCode
hasTests():boolean
generateReport(List<XmlSuite>,List<ISuite>,String):void
onTestStart(ITestResult):void
onTestSuccess(ITestResult):void
onTestFailure(ITestResult):void
onTestSkipped(ITestResult):void
onTestFailedButWithinSuccessPercentage(ITestResult):void
onStart(ITestContext):void
onFinish(ITestContext):void
<<Java Interface>>
IReporter
org.testng
generateReport(List<XmlSuite>,List<ISuite>,String):void
<<Java Interface>>
ITestListener
org.testng
onTestStart(ITestResult):void
onTestSuccess(ITestResult):void
onTestFailure(ITestResult):void
onTestSkipped(ITestResult):void
onTestFailedButWithinSuccessPercentage(ITestResult):void
onStart(ITestContext):void
onFinish(ITestContext):void
<<Java Class>>
DefaultListenerFactory
org.testng.internal
DefaultListenerFactory()
createListener(Class<ITestNGListener>):ITestNGListener

Diagram 8. Factory method. The ObjectAid UML Explorer for Eclipse

3.3 Observer

3.3.1 Pattern definition

The observer is a behavioural pattern used to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [1]

The pattern contains the subject interface, concrete subject implementation, observer interface and concrete observer implementation. In this context, the subject is an abstract class, that stores list of observers and notifies all of them when its state changes. Concrete subject stores state and notify observers when its state changes. The observer is an interface for sending updates. The concrete observer is storing observer state and maintains its consistency with the subject.

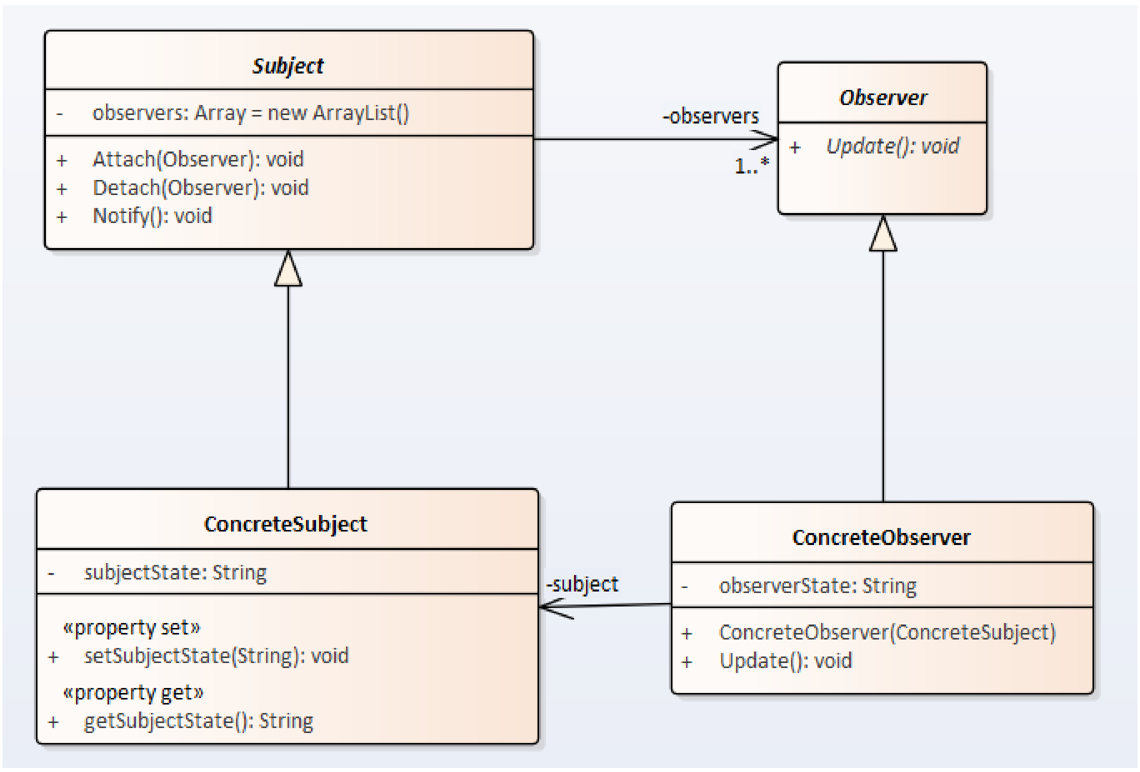


Diagram 9. Observer pattern

3.3.2 Java project description

For observer design pattern analysis, I am using small java application that the author wrote as an exercise for Software Architecture and Design course (IDU1550).

This application implements logic for the gateway, that opens only when a person pays for entrance and closes when one person passes the gate. If a person tries to pass without paying, an alarm is triggered. Gateway implementation is using observer GOF pattern to notify about payments and alarms. It is using java.util.Observable and java.util.Observer classes from Java core library to implement observer pattern.

3.3.3 Pattern UML representation analysis

In this project, Java core library implementation of subject and observer are used. Source code for java.util.Observable and java.util.Observer is not imported to the CASE tools projects. The minimal expectation in this case that CASE tools will be able to see and indicate on diagrams that classes implement or extend Observable and Observer.

Visual Paradigm and The ObjectAid UML Explorer could not find dependencies between StatusObserver and GatewayWithObserver classes. These CASE tools also do not indicate that StatusObserver is implementing Observer and GatewayWithObserver extends Observable. None parts of design pattern were found, and both CASE tools get 0 points.

Enterprise Architect could not find dependencies between StatusObserver and GatewayWithObserver classes as well, but on the right corner of the class box, it indicates that StatusObserver implements Observer and GatewayWithObserver generalises Observable. Most important parts of the pattern were found, and Enterprise Architect gets 2 points.

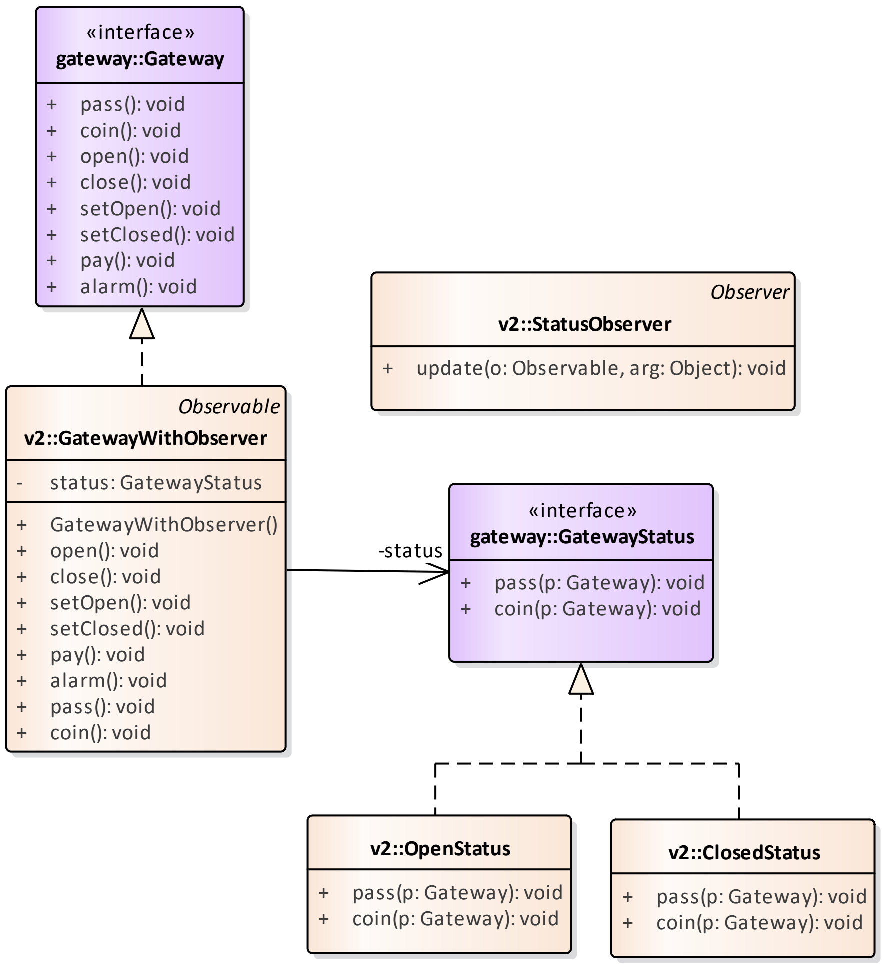


Diagram 10. Observer. Enterprise Architect

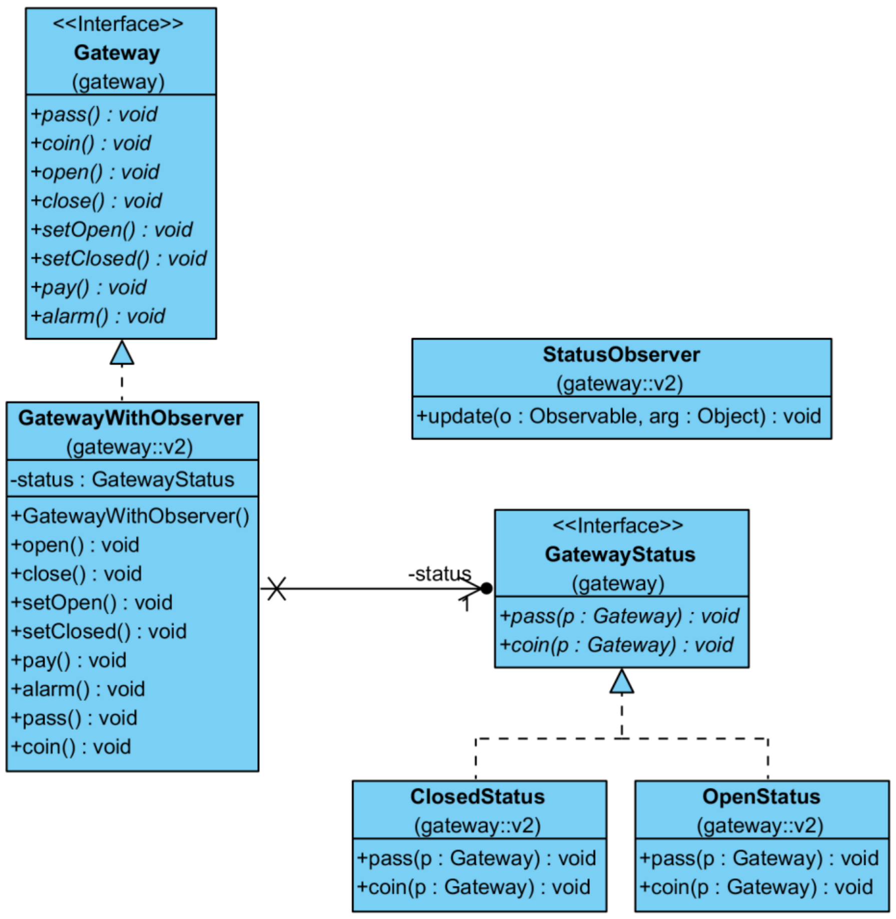


Diagram 11. Observer. Visual Paradigm.

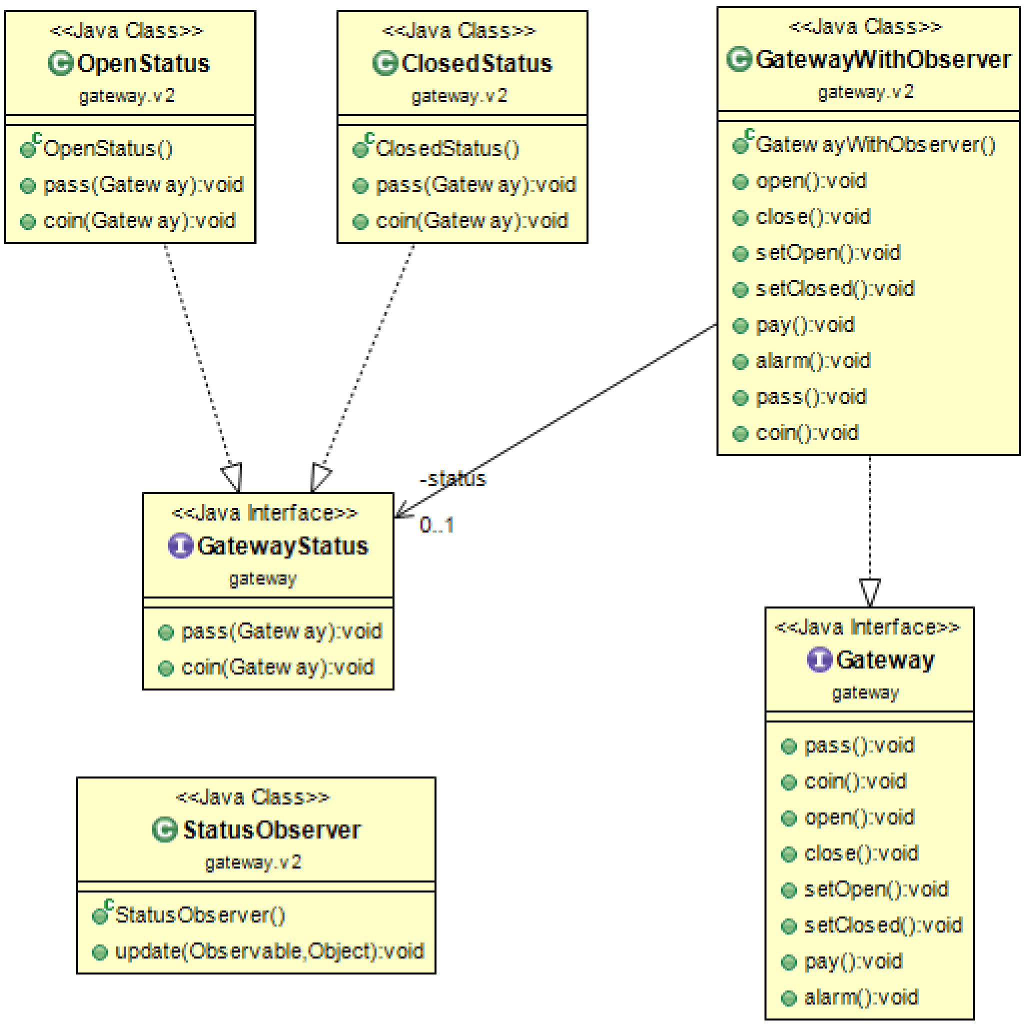


Diagram 12. Observer. The ObjectAid UML Explorer

3.4 Strategy

3.4.1 Pattern definition

The strategy is a behavioural pattern. The primary intent of this pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [1]

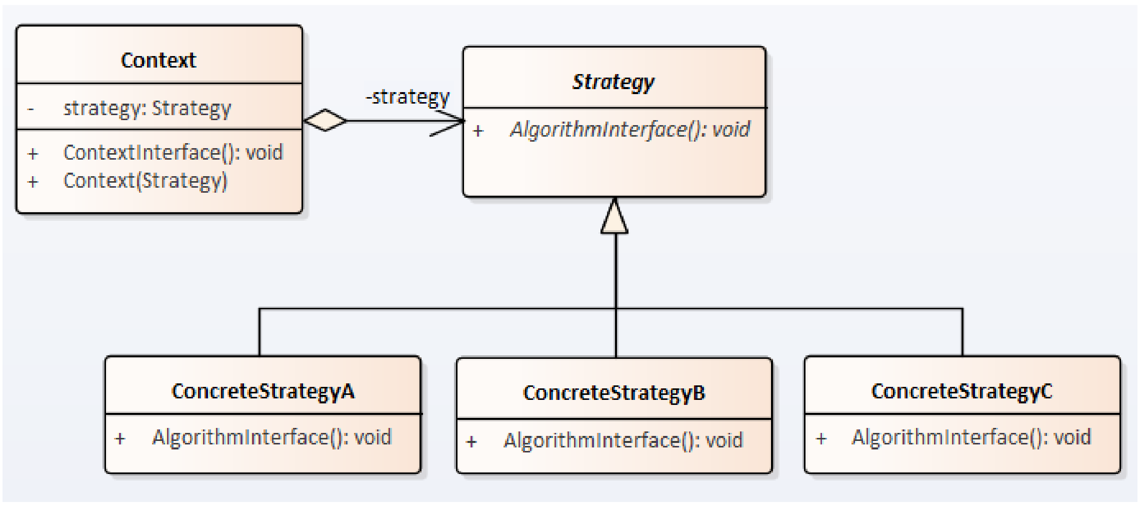

Context
- strategy: Strategy
+ ContextInterface(): void
+ Context(Strategy)
-strategy
Strategy
+ AlgorithmInterface(): void
ConcreteStrategyA
+ AlgorithmInterface(): void
ConcreteStrategyB
+ AlgorithmInterface(): void
ConcreteStrategyC
+ AlgorithmInterface(): void

Diagram 13. Strategy pattern

3.4.2 Java project description

For strategy pattern analysis I use TestNG test framework project [11].

In TestNG project strategy pattern is represented with InvocationStrategy interface. This interface is realised by 4 concrete classes:

- InvokeBeforeInvocationWithoutContextStrategy,
- InvokeBeforeInvocationWithContextStrategy,
- InvokeAfterInvocationWithoutContextStrategy,
- InvokeAfterInvocationWithContextStrategy

InvocationStrategy interface and its implementations are nested classes of InvokedMethodListenerInvoker class.

InvokedMethodListenerInvoker class stores map of strategies as private static property and retrieves required for listener type strategy using the private obtainStratefyFor method.

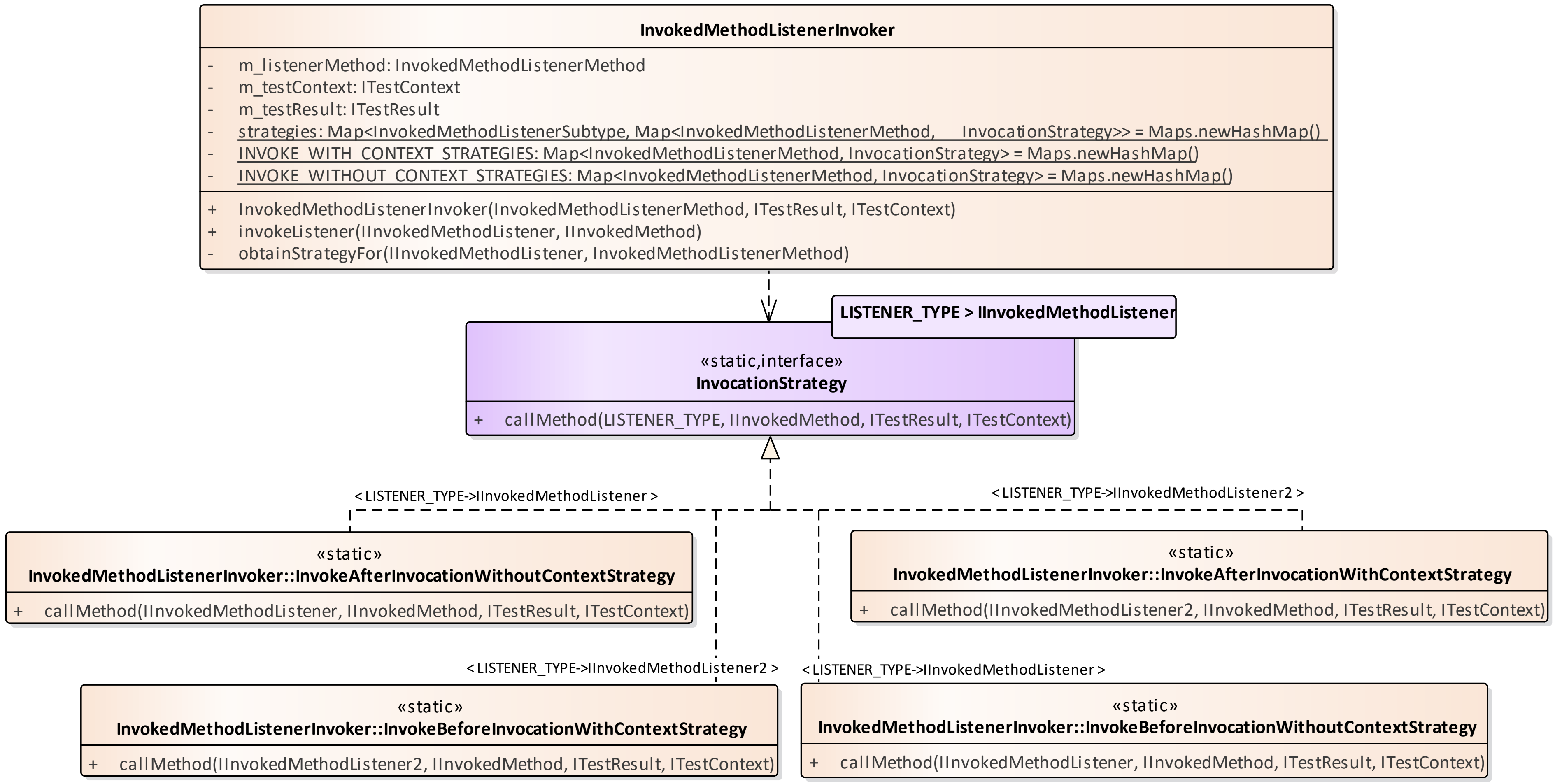
3.4.3 Pattern UML presentation analysis

All CASE tools identified realisation relationships between InvocationStrategy and four realisations of this class.

Enterprise Architect identified that InvokedMethodListenerInvoker has a dependency relationship with InvocationStrategy. CASE tools marked that strategy classes are nested classes for InvokedMethodListenerInvoker by changing the name of the class to format “outerClassName::nestedClassName”. All design pattern parts were found, and Enterprise Architect gets 3 points.

Visual paradigm shows that strategy classes are nested classes or are “part-of” InvokedMethodListenerInvoker using containment association, but the CASE tool does not show the dependency between InvocationStrategy and InvokedMethodListenerInvoker. One part of design pattern was not found, and Visual paradigm gets 2 points.

The ObjectAid UML Explorer identified nested classes and marked them with containment association. This CASE tool was able to locate that InvokedMethodListenerInvoker has two private map variable, which store references to InvocationStrategy classes. It indicated this relationship using association with zero to many multiplicities and label with the property name. Representation of this pattern is different from Enterprise Architect, but all parts of design pattern were found, and CASE tool gets 3 points.



InvokedMethodListenerInvoker
- m_listenerMethod: InvokedMethodListenerMethod
- m_testContext: ITestContext
- m_testResult: ITestResult
- strategies: Map<InvokedMethodListenerSubtype, Map<InvokedMethodListenerMethod, InvocationStrategy>> = Maps.newHashMap()
- INVOKE_WITH_CONTEXT_STRATEGIES: Map<InvokedMethodListenerMethod, InvocationStrategy> = Maps.newHashMap()
- INVOKE_WITHOUT_CONTEXT_STRATEGIES: Map<InvokedMethodListenerMethod, InvocationStrategy> = Maps.newHashMap()
+ InvokedMethodListenerInvoker(InvokedMethodListenerMethod, ITestResult, ITestContext)
+ invokeListener(IInvokedMethodListener, IInvokedMethod)
- obtainStrategyFor(IInvokedMethodListener, InvokedMethodListenerMethod)
LISTENER_TYPE > IInvokedMethodListener
«static,interface»
InvocationStrategy
+ callMethod(LISTENER_TYPE, IInvokedMethod, ITestResult, ITestContext)
< LISTENER_TYPE->IInvokedMethodListener >
< LISTENER_TYPE->IInvokedMethodListener2 >
«static»
InvokedMethodListenerInvoker::InvokeAfterInvocationWithoutContextStrategy
+ callMethod(IInvokedMethodListener, IInvokedMethod, ITestResult, ITestContext)
«static»
InvokedMethodListenerInvoker::InvokeAfterInvocationWithContextStrategy
+ callMethod(IInvokedMethodListener2, IInvokedMethod, ITestResult, ITestContext)
< LISTENER_TYPE->IInvokedMethodListener2 >
< LISTENER_TYPE->IInvokedMethodListener >
«static»
InvokedMethodListenerInvoker::InvokeBeforeInvocationWithContextStrategy
+ callMethod(IInvokedMethodListener2, IInvokedMethod, ITestResult, ITestContext)
«static»
InvokedMethodListenerInvoker::InvokeBeforeInvocationWithoutContextStrategy
+ callMethod(IInvokedMethodListener, IInvokedMethod, ITestResult, ITestContext)

Diagram 14. Strategy pattern. Enterprise Architect

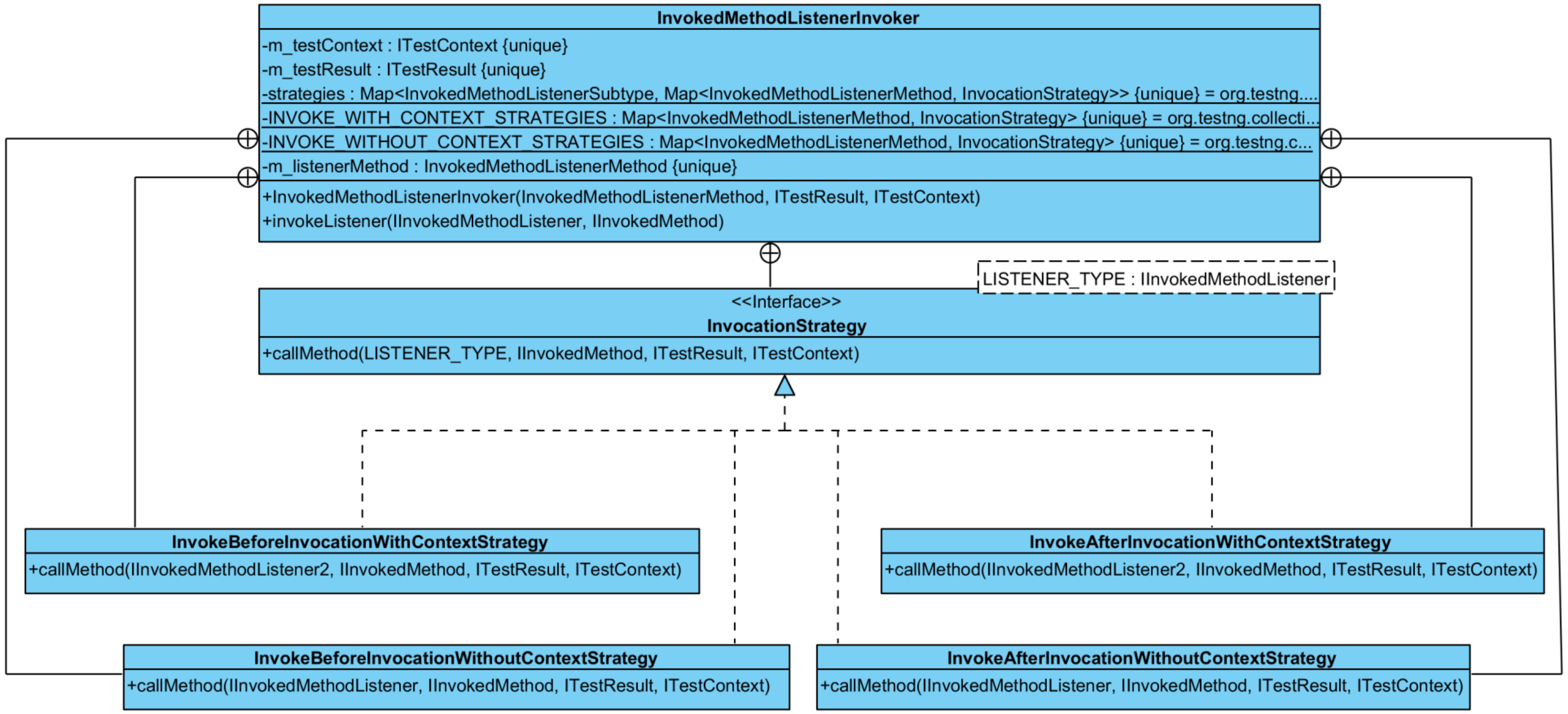
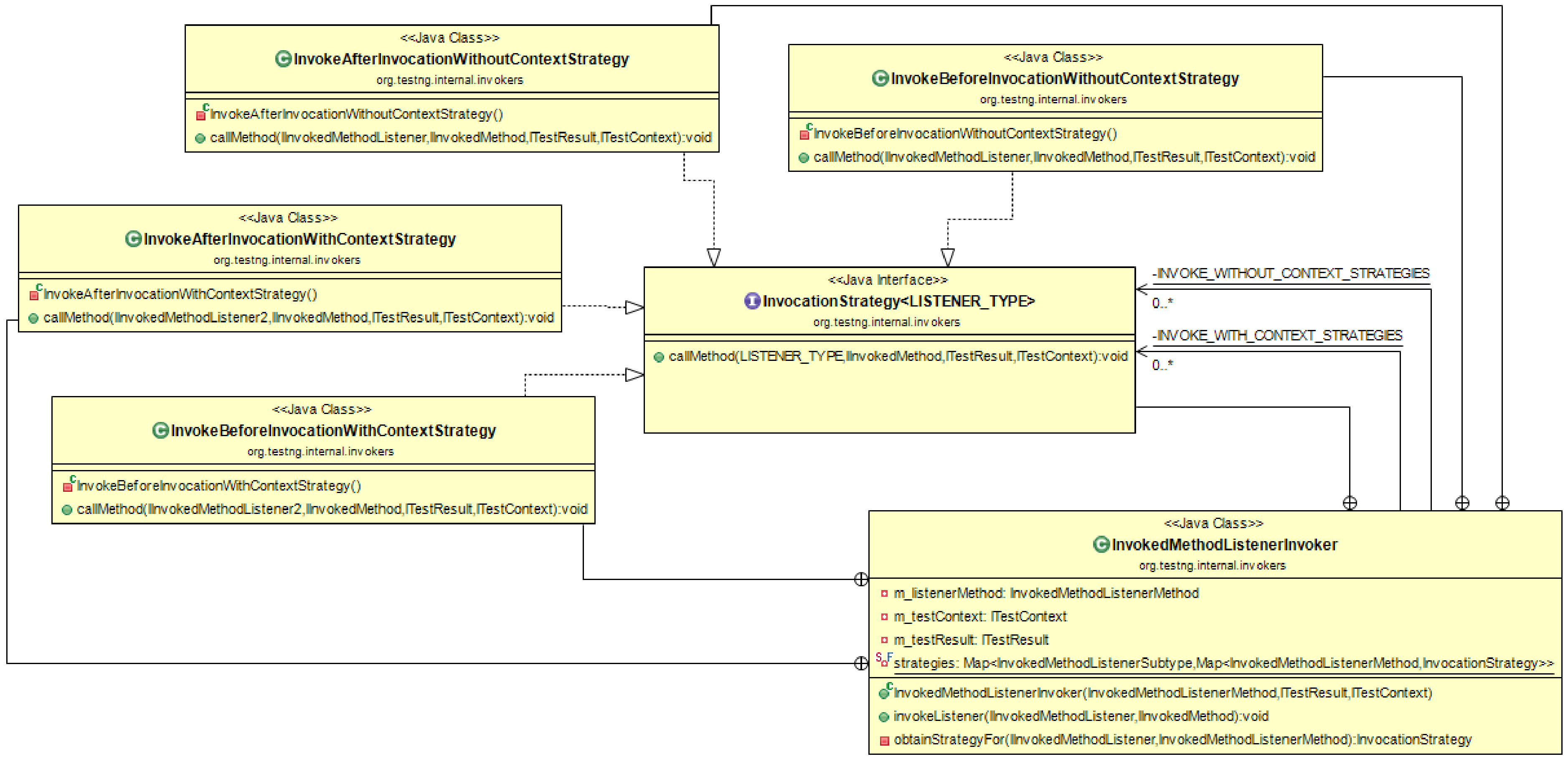

InvokedMethodListenerInvoker
-m_testContext : ITestContext {unique}
-m_testResult : ITestResult {unique}
-strategies : Map<InvokedMethodListenerSubtype, Map<InvokedMethodListenerMethod, InvocationStrategy>> {unique} = org.testng....
-INVOKE_WITH_CONTEXT_STRATEGIES : Map<InvokedMethodListenerMethod, InvocationStrategy> {unique} = org.testng.collecti...
-INVOKE_WITHOUT_CONTEXT_STRATEGIES : Map<InvokedMethodListenerMethod, InvocationStrategy> {unique} = org.testng.c...
-m_listenerMethod : InvokedMethodListenerMethod {unique}
+InvokedMethodListenerInvoker(InvokedMethodListenerMethod, ITestResult, ITestContext)
+invokeListener(IInvokedMethodListener, IInvokedMethod)
LISTENER_TYPE : IInvokedMethodListener
<<Interface>>
InvocationStrategy
+callMethod(LISTENER_TYPE, IInvokedMethod, ITestResult, ITestContext)
InvokeBeforeInvocationWithContextStrategy
+callMethod(IInvokedMethodListener2, IInvokedMethod, ITestResult, ITestContext)
InvokeAfterInvocationWithContextStrategy
+callMethod(IInvokedMethodListener2, IInvokedMethod, ITestResult, ITestContext)
InvokeBeforeInvocationWithoutContextStrategy
+callMethod(IInvokedMethodListener, IInvokedMethod, ITestResult, ITestContext)
InvokeAfterInvocationWithoutContextStrategy
+callMethod(IInvokedMethodListener, IInvokedMethod, ITestResult, ITestContext)

Diagram 15. Strategy pattern. Visual Paradigm



<<Java Class>>
InvokeAfterInvocationWithoutContextStrategy
org.testng.internal.invokers
InvokeAfterInvocationWithoutContextStrategy()
callMethod(IInvokedMethodListener,IInvokedMethod,ITestResult,ITestContext):void
<<Java Class>>
InvokeBeforeInvocationWithoutContextStrategy
org.testng.internal.invokers
InvokeBeforeInvocationWithoutContextStrategy()
callMethod(IInvokedMethodListener,IInvokedMethod,ITestResult,ITestContext):void
<<Java Class>>
InvokeAfterInvocationWithContextStrategy
org.testng.internal.invokers
InvokeAfterInvocationWithContextStrategy()
callMethod(IInvokedMethodListener2,IInvokedMethod,ITestResult,ITestContext):void
<<Java Interface>>
InvocationStrategy<LISTENER_TYPE>
org.testng.internal.invokers
callMethod(LISTENER_TYPE,IInvokedMethod,ITestResult,ITestContext):void
-INVOKE_WITHOUT_CONTEXT_STRATEGIES
0..*
-INVOKE_WITH_CONTEXT_STRATEGIES
0..*
<<Java Class>>
InvokeBeforeInvocationWithContextStrategy
org.testng.internal.invokers
InvokeBeforeInvocationWithContextStrategy()
callMethod(IInvokedMethodListener2,IInvokedMethod,ITestResult,ITestContext):void
<<Java Class>>
InvokedMethodListenerInvoker
org.testng.internal.invokers
m_listenerMethod: InvokedMethodListenerMethod
m_testContext: ITestContext
m_testResult: ITestResult
strategies: Map<InvokedMethodListenerSubtype,Map<InvokedMethodListenerMethod,InvocationStrategy>>
InvokedMethodListenerInvoker(InvokedMethodListenerMethod,ITestResult,ITestContext)
invokeListener(IInvokedMethodListener,IInvokedMethod):void
obtainStrategyFor(IInvokedMethodListener,InvokedMethodListenerMethod):InvocationStrategy

Diagram 16. Strategy pattern. The ObjectAid UML Explorer for Eclipse

3.5 Iterator

3.5.1 Pattern definition

An iterator is a behavioural pattern used to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. [1]

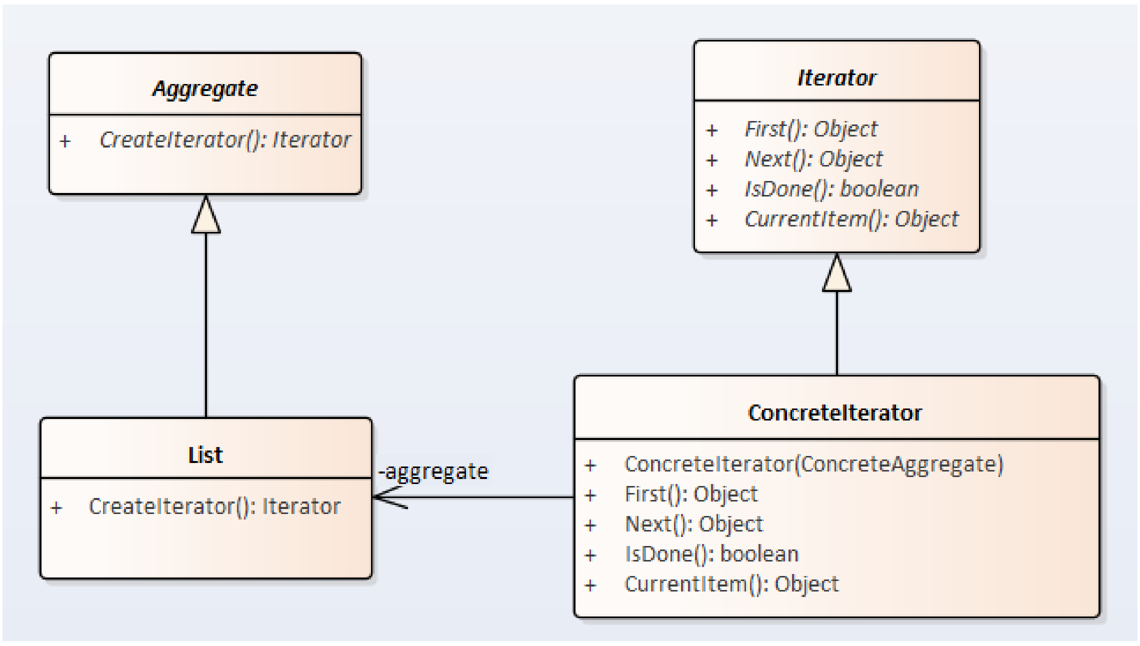


Diagram 17. Iterator pattern

3.5.2 Java project description

For iterator pattern analysis I use TestNG test framework project [11].

As iterator interface TestNG project uses java.util.Iterator interface defined in Java core library. There are three implementations of iterator interface in this project.

- OneToTwoDimArrayIterator
- OneToTwoDimIterator
- ArrayIterator

MethodInvocationHelper class is instantiating these iterators during invokeDataProvider method execution.

3.5.3 Pattern UML presentation analysis

Enterprise Architect identified that three concrete iterators are implementing iterator interface, but it could not determine dependency relationship between iterators and MethodInvocationHelper class. One part of design pattern is missing, and this CASE tool gets 2 points.

Visual Paradigm does not show that concrete iterators implement iterator interface and does not see dependency between iterators and MethodInvocationHelper class. This CASE tool could not identify any parts of the pattern and gets 0 points.

The ObjectAid UML Explorer identified a dependency between iterators and MethodInvocationHelper class but does not show that concrete iterators implement iterator interface. One most important part of design pattern is missing, that is why CASE tool gets 1 point.

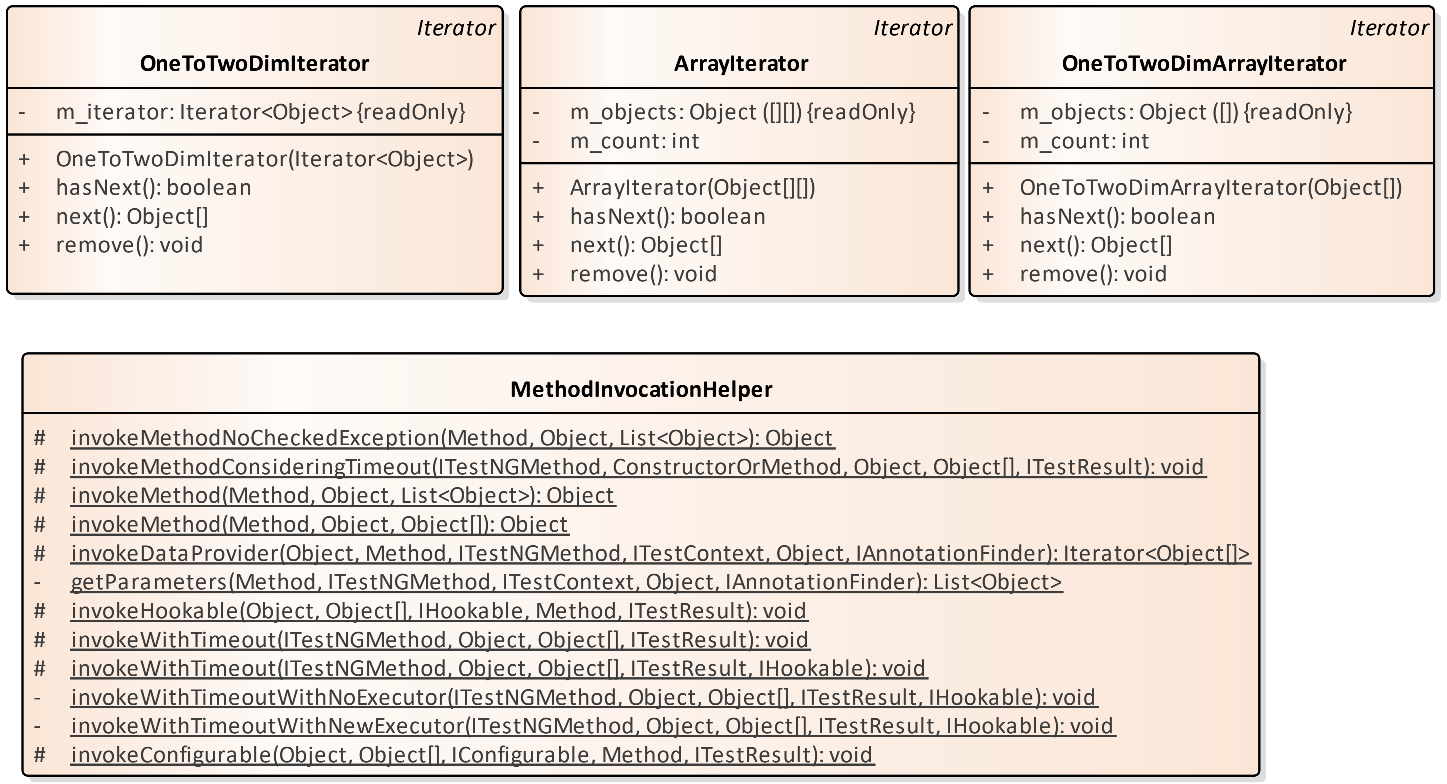

Iterator
OneToTwoDimIterator
- m_iterator: Iterator<Object> {readOnly}
+ OneToTwoDimIterator(Iterator<Object>)
+ hasNext(): boolean
+ next(): Object[]
+ remove(): void
Iterator
ArrayIterator
- m_objects: Object ([][]) {readOnly}
- m_count: int
+ ArrayIterator(Object[][])
+ hasNext(): boolean
+ next(): Object[]
+ remove(): void
Iterator
OneToTwoDimArrayIterator
- m_objects: Object ([]) {readOnly}
- m_count: int
+ OneToTwoDimArrayIterator(Object[])
+ hasNext(): boolean
+ next(): Object[]
+ remove(): void
MethodInvocationHelper
# invokeMethodNoCheckedException(Method, Object, List<Object>): Object
# invokeMethodConsideringTimeout(ITestNGMethod, ConstructorOrMethod, Object, Object[], ITestResult): void
# invokeMethod(Method, Object, List<Object>): Object
# invokeMethod(Method, Object, Object[]): Object
# invokeDataProvider(Object, Method, ITestNGMethod, ITestContext, Object, IAnnotationFinder): Iterator<Object[]>
- getParameters(Method, ITestNGMethod, ITestContext, Object, IAnnotationFinder): List<Object>
# invokeHookable(Object, Object[], IHookable, Method, ITestResult): void
# invokeWithTimeout(ITestNGMethod, Object, Object[], ITestResult): void
# invokeWithTimeout(ITestNGMethod, Object, Object[], ITestResult, IHookable): void
- invokeWithTimeoutWithNoExecutor(ITestNGMethod, Object, Object[], ITestResult, IHookable): void
- invokeWithTimeoutWithNewExecutor(ITestNGMethod, Object, Object[], ITestResult, IHookable): void
# invokeConfigurable(Object, Object[], IConfigurable, Method, ITestResult): void

Diagram 18. Iterator pattern. Enterprise Architect

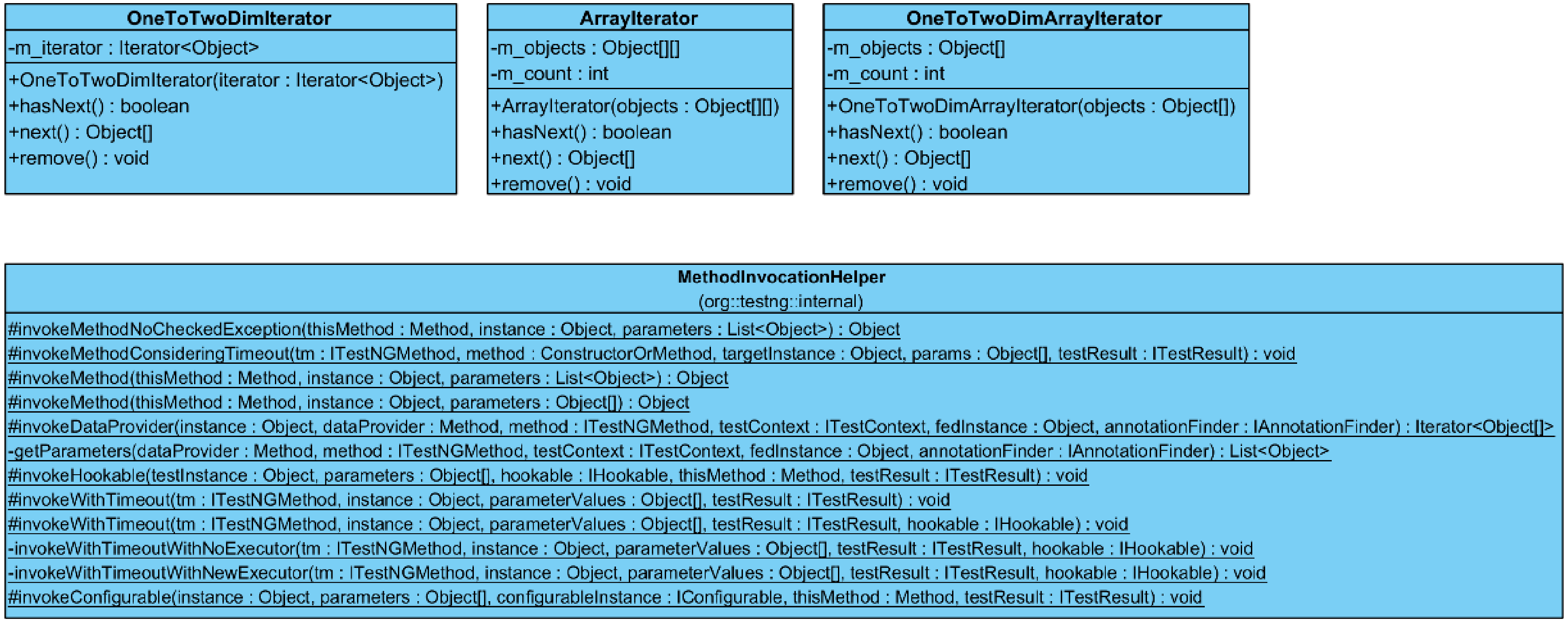
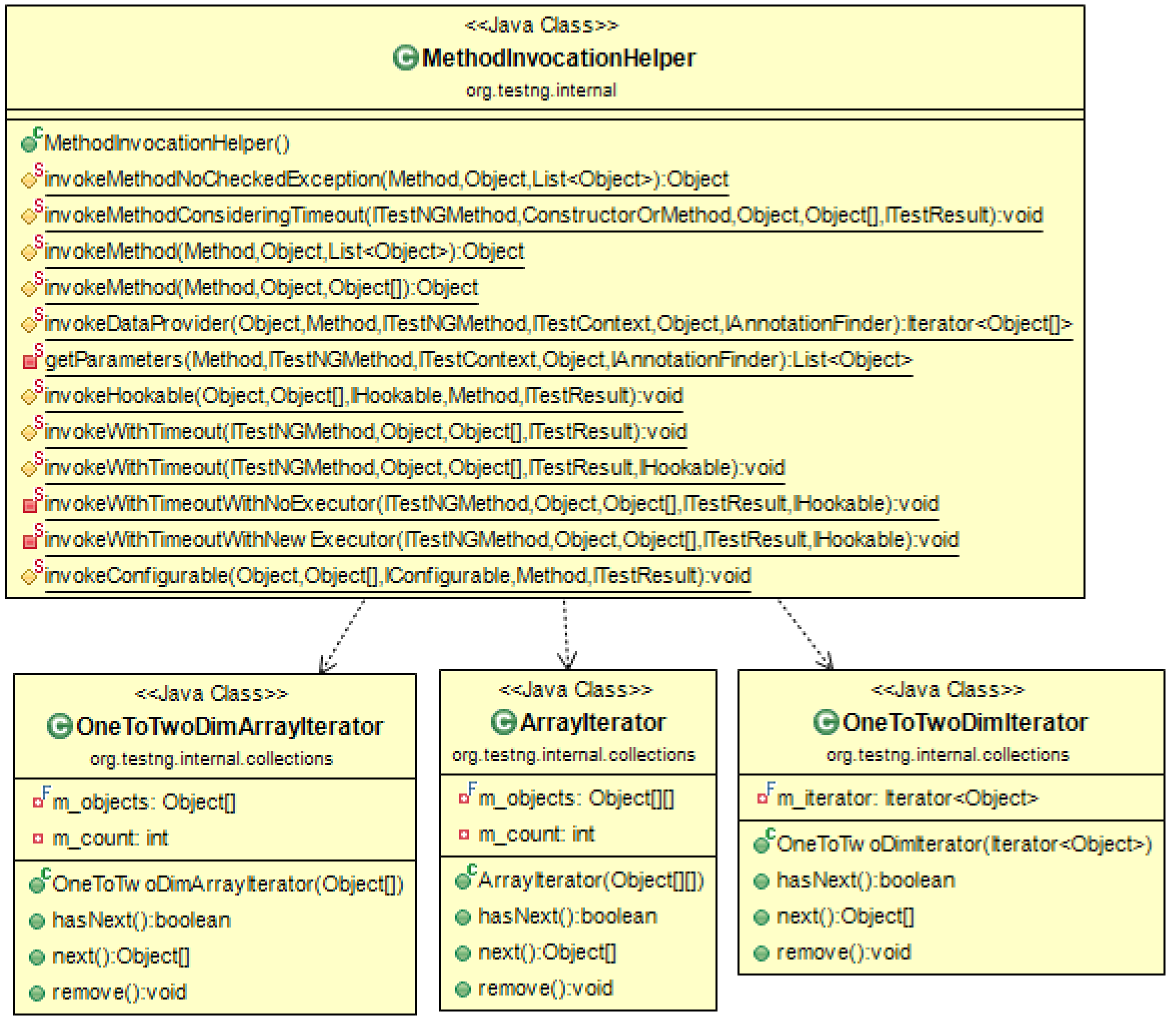

OneToTwoDimIterator
-m_iterator : Iterator<Object>
+OneToTwoDimIterator(iterator : Iterator<Object>)
+hasNext() : boolean
+next() : Object[]
+remove() : void
ArrayIterator
-m_objects : Object[][]
-m_count : int
+ArrayIterator(objects : Object[][])
+hasNext() : boolean
+next() : Object[]
+remove() : void
OneToTwoDimArrayIterator
-m_objects : Object[]
-m_count : int
+OneToTwoDimArrayIterator(objects : Object[])
+hasNext() : boolean
+next() : Object[]
+remove() : void
MethodInvocationHelper
(org::testng::internal)
#invokeMethodNoCheckedException(thisMethod : Method, instance : Object, parameters : List<Object>) : Object
#invokeMethodConsideringTimeout(tm : ITestNGMethod, method : ConstructorOrMethod, targetInstance : Object, params : Object[], testResult : ITestResult) : void
#invokeMethod(thisMethod : Method, instance : Object, parameters : List<Object>) : Object
#invokeMethod(thisMethod : Method, instance : Object, parameters : Object[]) : Object
#invokeDataProvider(instance : Object, dataProvider : Method, method : ITestNGMethod, testContext : ITestContext, fedInstance : Object, annotationFinder : IAnnotationFinder) : Iterator<Object[]>
-getParameters(dataProvider : Method, method : ITestNGMethod, testContext : ITestContext, fedInstance : Object, annotationFinder : IAnnotationFinder) : List<Object>
#invokeHookable(testInstance : Object, parameters : Object[], hookable : IHookable, thisMethod : Method, testResult : ITestResult) : void
#invokeWithTimeout(tm : ITestNGMethod, instance : Object, parameterValues : Object[], testResult : ITestResult) : void
#invokeWithTimeout(tm : ITestNGMethod, instance : Object, parameterValues : Object[], testResult : ITestResult, hookable : IHookable) : void
-invokeWithTimeoutWithNoExecutor(tm : ITestNGMethod, instance : Object, parameterValues : Object[], testResult : ITestResult, hookable : IHookable) : void
-invokeWithTimeoutWithNewExecutor(tm : ITestNGMethod, instance : Object, parameterValues : Object[], testResult : ITestResult, hookable : IHookable) : void
#invokeConfigurable(instance : Object, parameters : Object[], configurableInstance : IConfigurable, thisMethod : Method, testResult : ITestResult) : void

Diagram 19. Iterator pattern. Visual Paradigm



<<Java Class>>
MethodInvocationHelper
org.testng.internal
MethodInvocationHelper()
invokeMethodNoCheckedException(Method,Object,List<Object>):Object
invokeMethodConsideringTimeout(ITestNGMethod,ConstructorOrMethod,Object,Object[],ITestResult):void
invokeMethod(Method,Object,List<Object>):Object
invokeMethod(Method,Object,Object[]):Object
invokeDataProvider(Object,Method,ITestNGMethod,ITestContext,Object,IAnnotationFinder):Iterator<Object[]>
getParameters(Method,ITestNGMethod,ITestContext,Object,IAnnotationFinder):List<Object>
invokeHookable(Object,Object[],IHookable,Method,ITestResult):void
invokeWithTimeout(ITestNGMethod,Object,Object[],ITestResult):void
invokeWithTimeout(ITestNGMethod,Object,Object[],ITestResult,IHookable):void
invokeWithTimeoutWithNoExecutor(ITestNGMethod,Object,Object[],ITestResult,IHookable):void
invokeWithTimeoutWithNew Executor(ITestNGMethod,Object,Object[],ITestResult,IHookable):void
invokeConfigurable(Object,Object[],IConfigurable,Method,ITestResult):void
<<Java Class>>
OneToTwoDimArrayIterator
org.testng.internal.collections
m_objects: Object[]
m_count: int
OneToTw oDimArrayIterator(Object[])
hasNext():boolean
next():Object[]
remove():void
<<Java Class>>
ArrayIterator
org.testng.internal.collections
m_objects: Object[][]
m_count: int
ArrayIterator(Object[][])
hasNext():boolean
next():Object[]
remove():void
<<Java Class>>
OneToTwoDimIterator
org.testng.internal.collections
m_iterator: Iterator<Object>
OneToTw oDimIterator(Iterator<Object>)
hasNext():boolean
next():Object[]
remove():void

Diagram 20. Iterator pattern. The ObjectAid UML Explorer

3.6 Bridge

3.6.1 Pattern definition

The bridge is a structural pattern used to decouple an abstraction from its implementation so that the two can vary independently. [1]

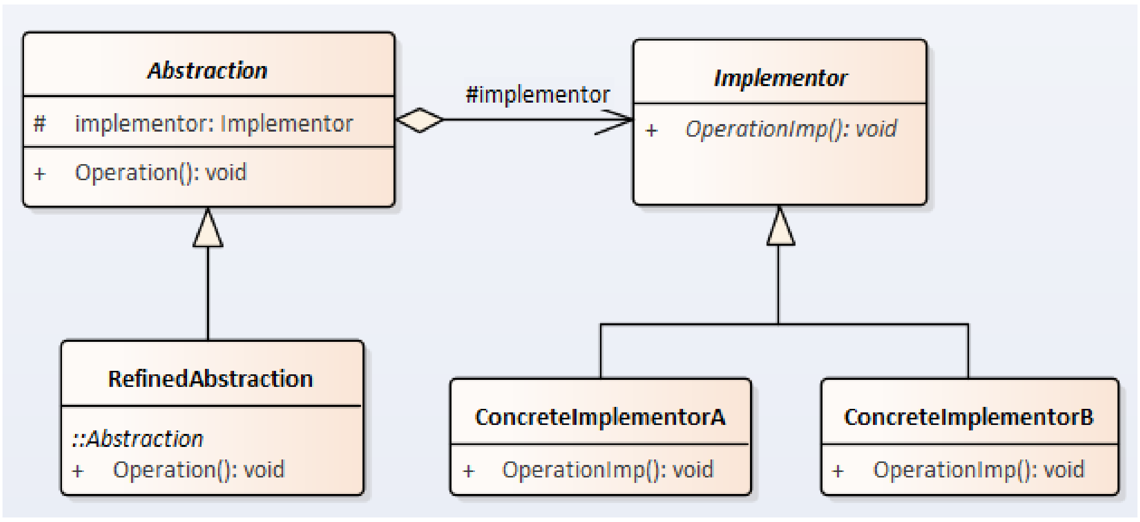


Diagram 21. Bridge pattern

3.6.2 Java project description

For bridge pattern analysis I use TestNG test framework project [link].

In the TestNG project, bridge pattern is used to decouple annotation finder and test methods implementations. In this case, IAnnotationFinder acts as a bridge between BaseTestMethod and its implementations. The project contains only one implementation of IAnnotationFinder, but other implementations can be added easily without significant changes in BaseTestMethod or its implementations.

3.6.3 Pattern UML presentation analysis

All CASE tools determined that BaseTestMethod have four methods that extend it. As expected, CASE tools marked this relationship with generalisation arrow.

All CASE tools identified that IAnnotationFinder is an interface and have one realisation called JDK15AnnotationFinder. CASE tools marked this relationship using realisation arrow.

All CASE tools identified an association between BaseTestMethod and IAnnothationFinder interface as well. It is represented by association arrow, but aggregation arrow was expected.

Enterprise Architect and The ObjectAid UML Explorer discovered dependency relationships between FactoryMethod, ConfigurationMethod, TestNGMethod and IAnnotationFinder. It is expected because in some methods these classes use IAnnotationFinder instance stored in BaseTestMethod. It means that these CASE tools identified all expected parts and got 3 points. Visual paradigm could not identify this dependency and gets 2 points.

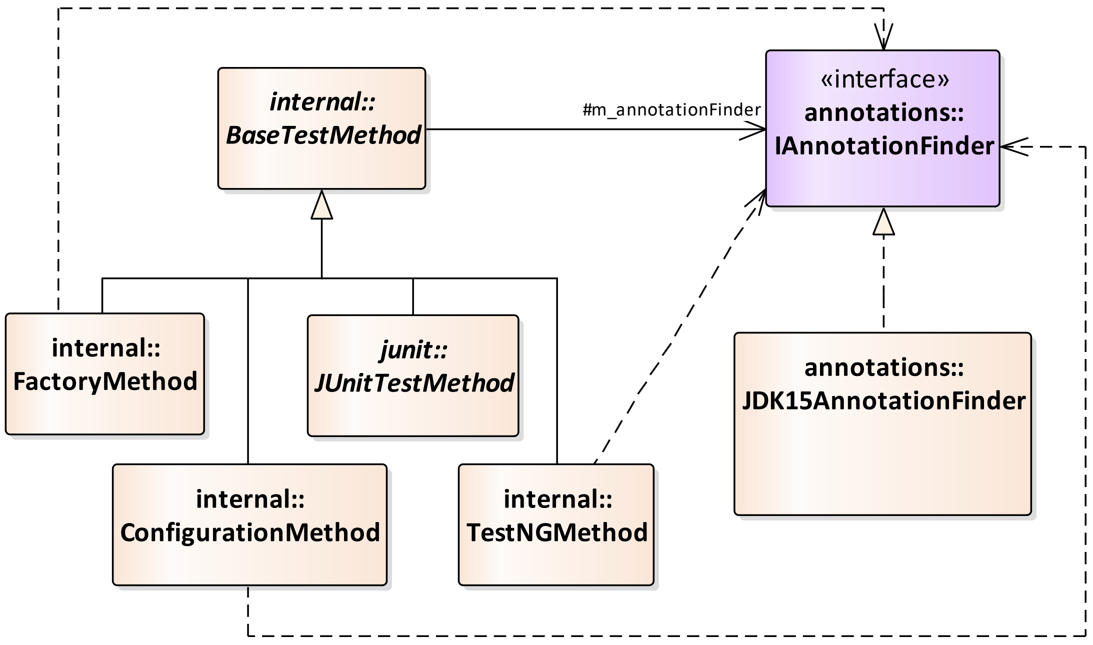


Diagram 22. Bridge. Enterprise Architect.

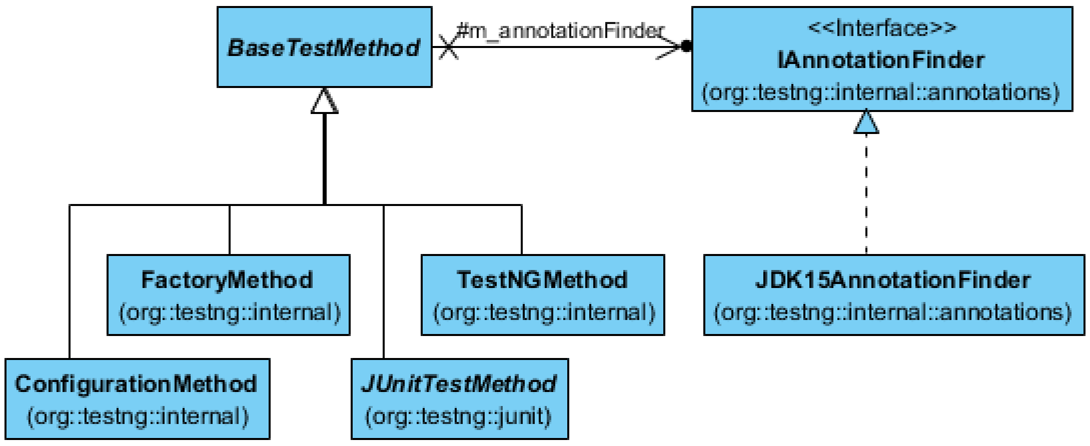


Diagram 23. Bridge. Visual Paradigm

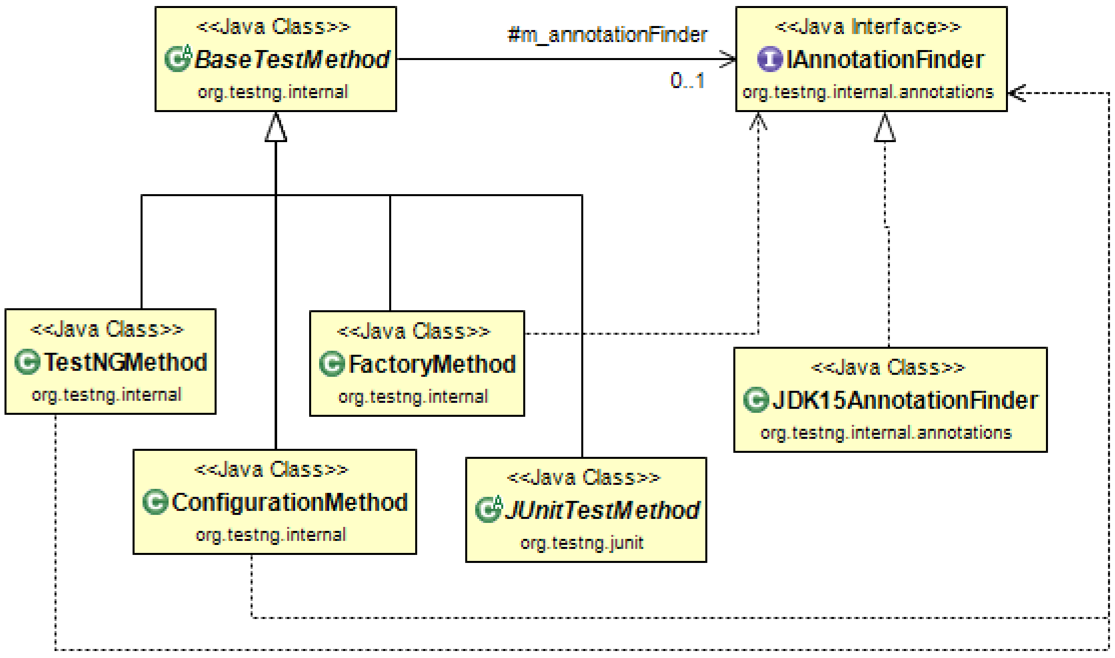


Diagram 24. Bridge. The ObjectAid UML Explorer for Eclipse.

3.7 Adapter

3.7.1 Pattern definition

The adapter is the structural pattern used to convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces. [1]

Adapter pattern can be applied to the class by using multiple inheritances or to the object by composing object instance.

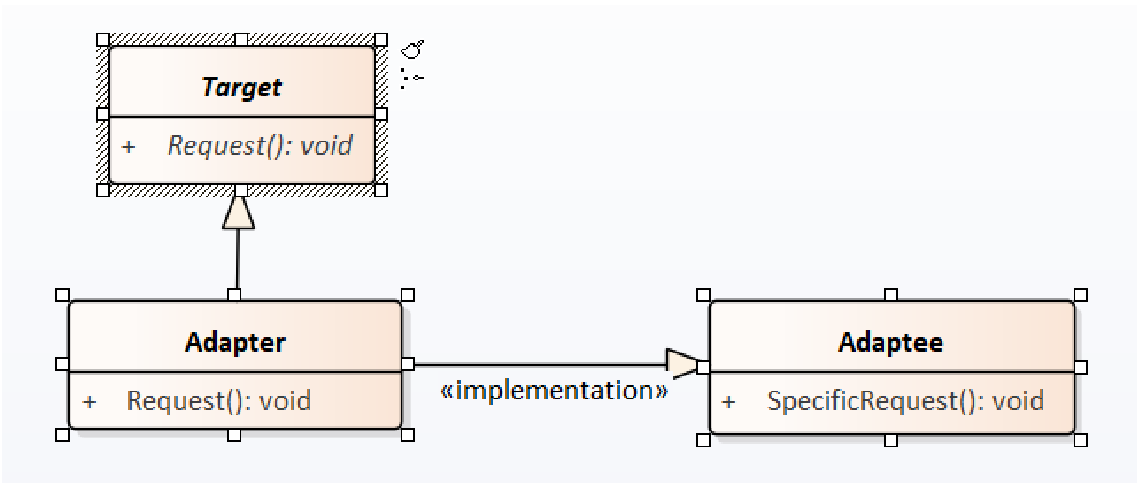


Diagram 25. Adapter pattern using multiple inheritances.

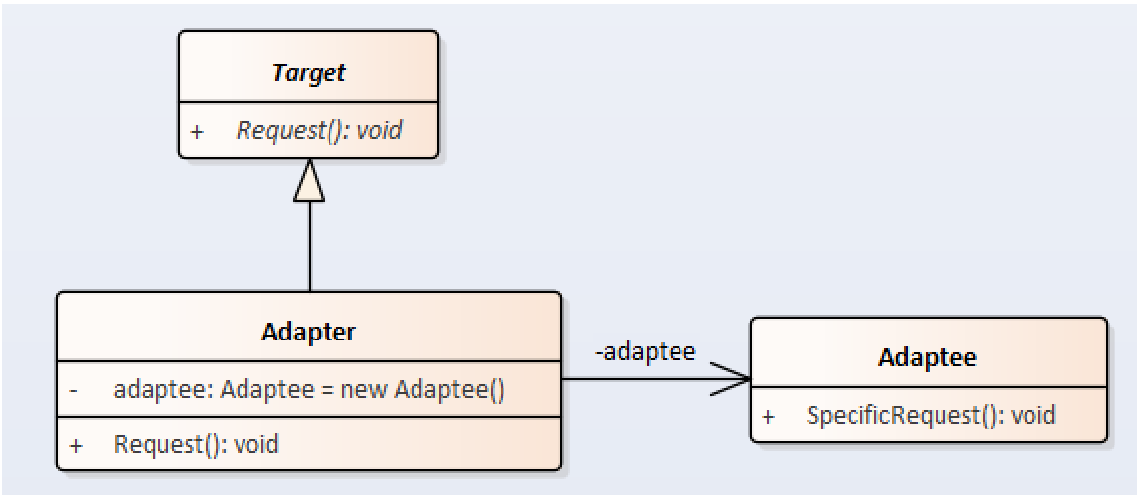


Diagram 26. Adapter pattern by composing object instance.

3.7.2 Java project description

For adapter design pattern analysis, adapter pattern implementation from “Design patterns implemented in Java” project in GitHub [12] is used.

In the project, we have a captain who has row skill. Captain can row the rowing boat. Then requirements change, and captain needs to operate a fishing boat that can only sail using his rowing ability. In this case, fishing boat adapter will help. The adapter is implementing rowing boat interface but using fishing boat sailing functionality for rowing.

3.7.3 Pattern UML presentation analysis

All CASE tools found realisation dependency between FishingBoatAdapter and RowingBoat interface.

All CASE tools found an association between FishingBoatAdapter and FishingBoat. There is only one difference between representations by different CASE tools. Enterprise architect and Visual paradigm show it as a solid line with variable name value and shows variable with type FishingBoat inside the class box. The ObjectAid UML Explorer only indicates an association with the label.

All CASE tools identified all design pattern parts, and each gets 3 points.

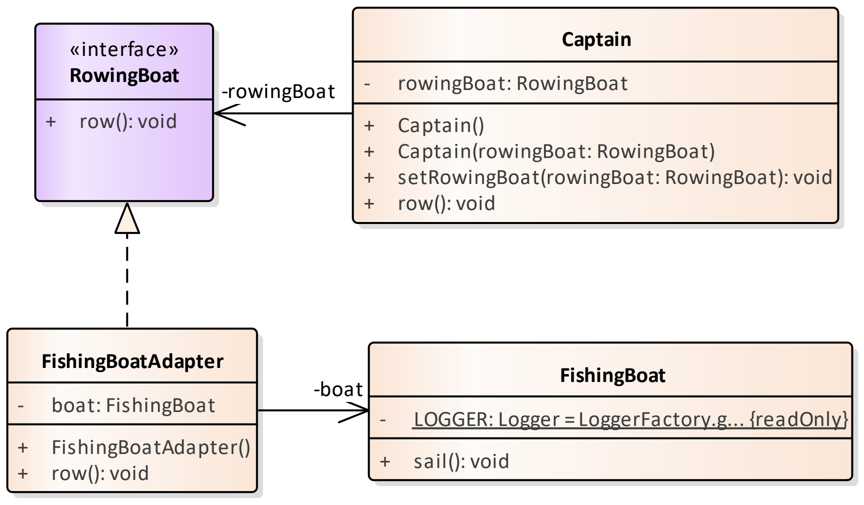


Diagram 27. Adapter. Enterprise Architect

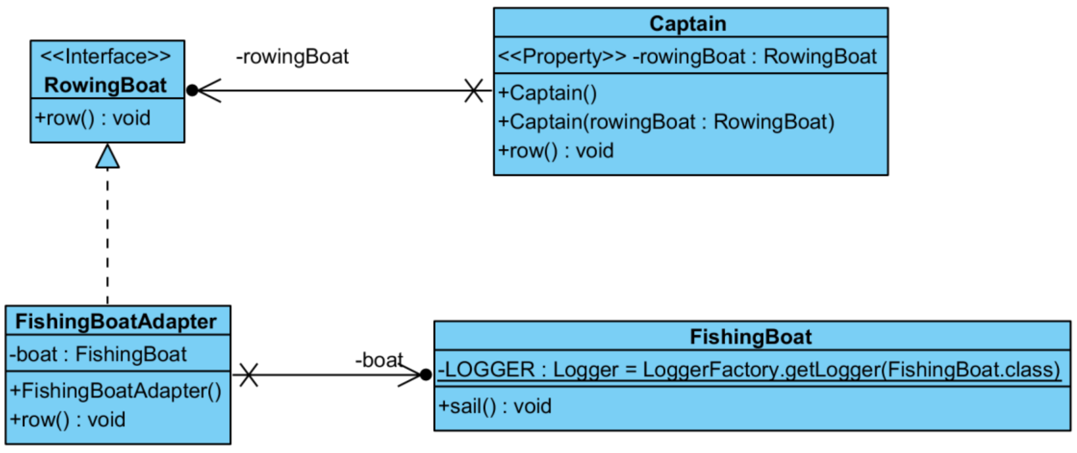


Diagram 28. Adapter. Visual Paradigm

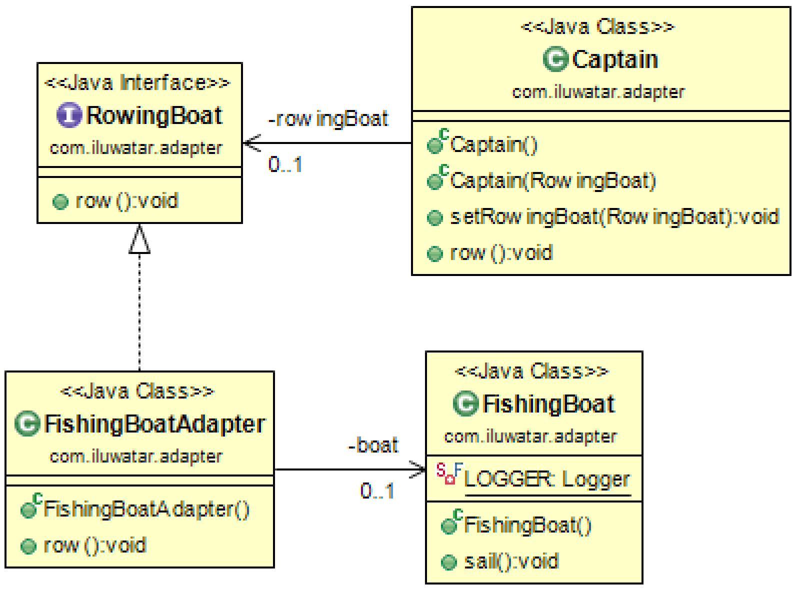


Diagram 29. Adapter. The ObjectAid UML Explorer

3.8 Composite

3.8.1 Pattern definition

Composite is a structural pattern used to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. [1]

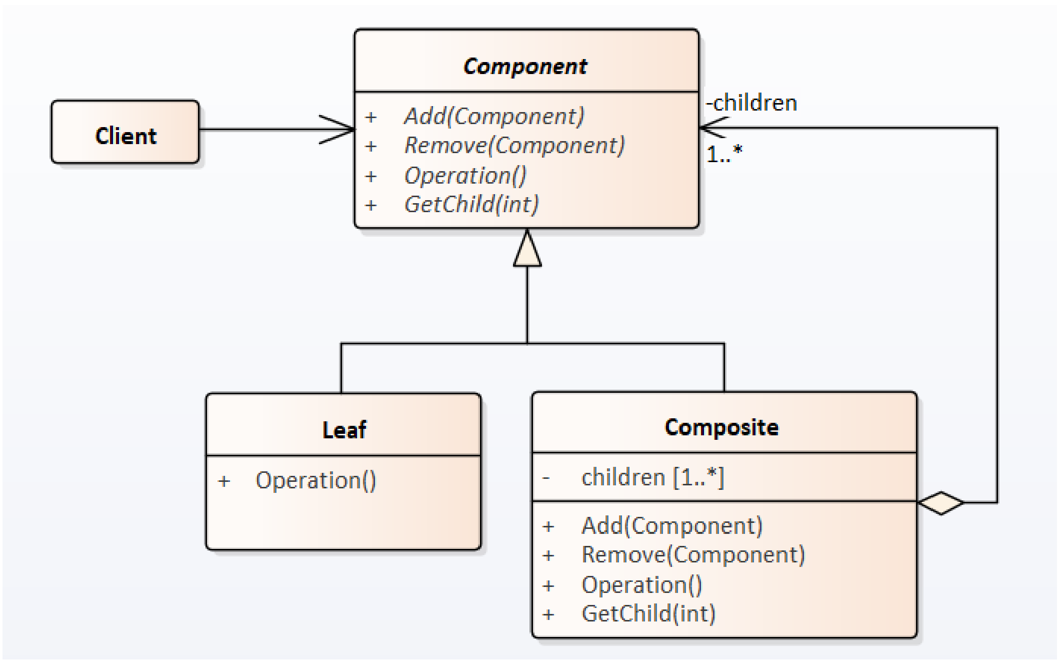


Diagram 30. Composite pattern

3.8.2 Java project description

Small java application that the author wrote as an exercise in Software Architecture and Design course (IDU1550) [16] is used for composite design pattern analysis.

This application purpose to draw the Sierpinski triangle [19]. SierpinskiTriangle stores information about points that indicate triangle tops and between which program draw lines. SierpinskiUhend stores information about parent triangle and child triangles.

3.8.3 Pattern UML presentation analysis

Diagrams generated by all CASE tools contain generalisation relationship between SierpinskiUhend and SierpinskiTriangle, but none of them includes composition line between this classes. Instead, diagrams show association line with variable name as a label and zero to more multiplicity. It is counted as missing part of design pattern, and each CASE tool gets 2 points.

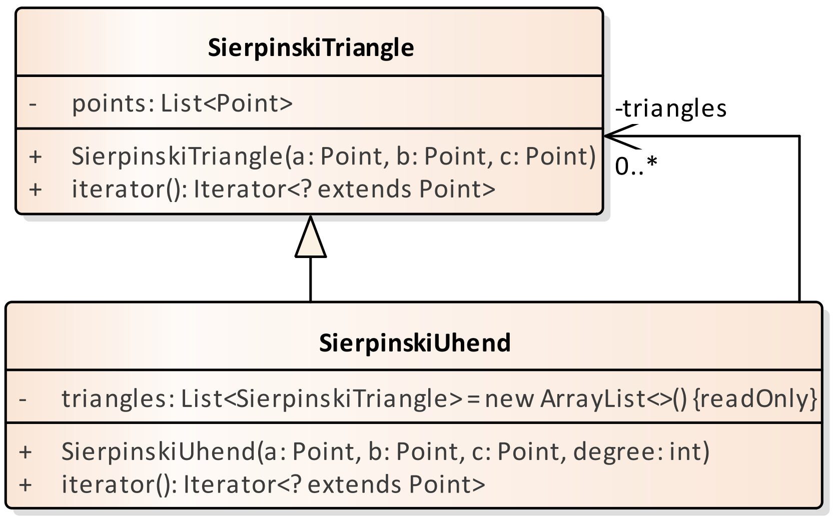

SierpinskiTriangle
- points: List<Point>
+ SierpinskiTriangle(a: Point, b: Point, c: Point)
+ iterator(): Iterator<? extends Point>
-triangles
0..*
SierpinskiUhend
- triangles: List<SierpinskiTriangle> = new ArrayList<>() {readOnly}
+ SierpinskiUhend(a: Point, b: Point, c: Point, degree: int)
+ iterator(): Iterator<? extends Point>

Diagram 31. Composite. Enterprise Architect

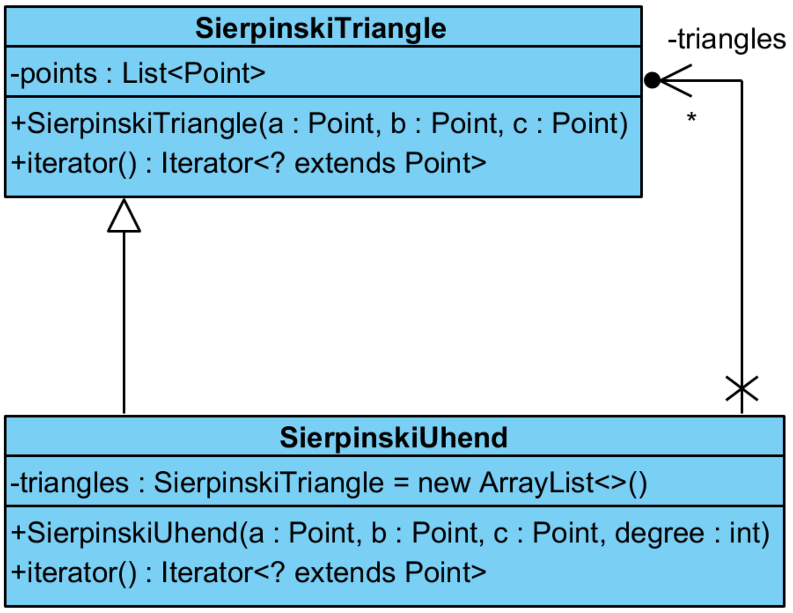

SierpinskiTriangle
-points : List<Point>
+SierpinskiTriangle(a : Point, b : Point, c : Point)
+iterator() : Iterator<? extends Point>
-triangles
*
SierpinskiUhend
-triangles : SierpinskiTriangle = new ArrayList<>()
+SierpinskiUhend(a : Point, b : Point, c : Point, degree : int)
+iterator() : Iterator<? extends Point>

Diagram 32. Composite. Visual Paradigm

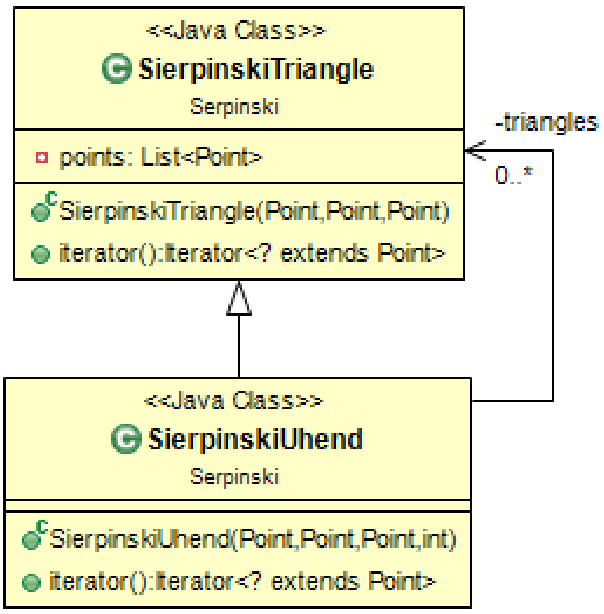

<<Java Class>>
SierpinskiTriangle
Serpinski
points: List<Point>
SierpinskiTriangle(Point,Point,Point)
iterator():Iterator<? extends Point>
-triangles
0..*
<<Java Class>>
SierpinskiUhend
Serpinski
SierpinskiUhend(Point,Point,Point,int)
iterator():Iterator<? extends Point>

Diagram 33. Composite. The ObjectAid UML Explorer

3.9 Template method

3.9.1 Pattern definition

Template method is the behavioural pattern used to Define the skeleton of an algorithm in operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [1]

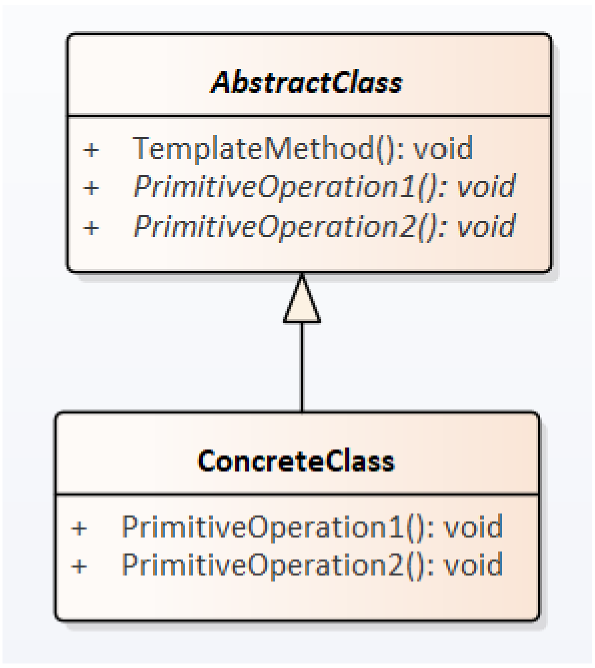

AbstractClass
+ TemplateMethod(): void
+ PrimitiveOperation1(): void
+ PrimitiveOperation2(): void
ConcreteClass
+ PrimitiveOperation1(): void
+ PrimitiveOperation2(): void

Diagram 34. Template method pattern

3.9.2 Java project description

For template method pattern analysis I use TestNG test framework project [11].

TestNG project contains abstract class BaseMultiSuitePanel that contains two abstract methods called getHeader and getContent. Sever subclasses implement this abstract class and its abstract methods.

3.9.3 Pattern UML presentation analysis

Diagrams generated by each CASE tool contain BaseMultiSuirePanel class box, and its abstract methods names have italic formatting. Each CASE tool was able to find generalisation relationship between BaseMultiSuirePanel abstract class and its implementations.

All design pattern parts were identified correctly, and all CASE tools get 3 points.

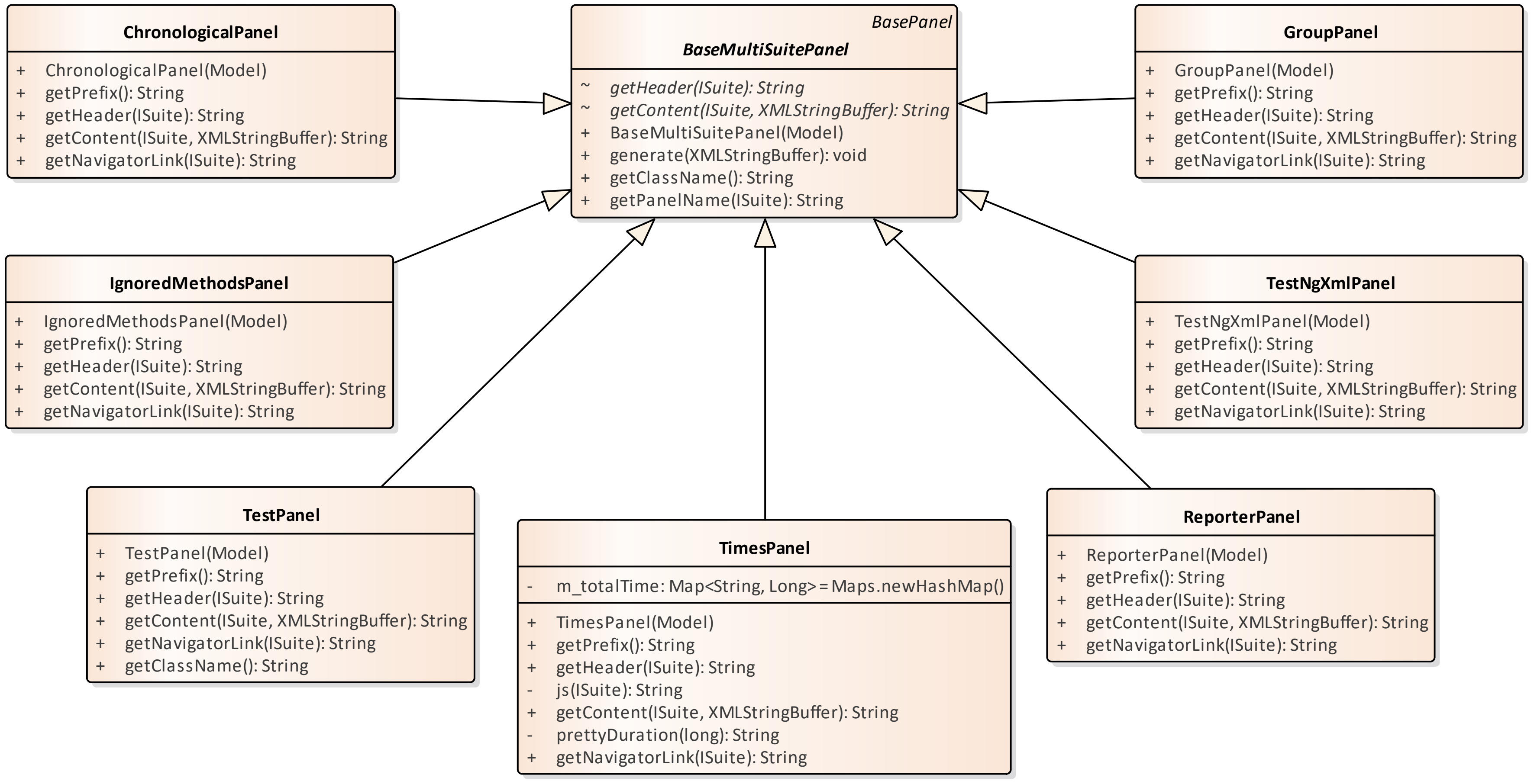

BasePanel
BaseMultiSuitePanel
~ getHeader(ISuite): String
~ getContent(ISuite, XMLStringBuffer): String
+ BaseMultiSuitePanel(Model)
+ generate(XMLStringBuffer): void
+ getClassName(): String
+ getPanelName(ISuite): String
ChronologicalPanel
+ ChronologicalPanel(Model)
+ getPrefix(): String
+ getHeader(ISuite): String
+ getContent(ISuite, XMLStringBuffer): String
+ getNavigatorLink(ISuite): String
GroupPanel
+ GroupPanel(Model)
+ getPrefix(): String
+ getHeader(ISuite): String
+ getContent(ISuite, XMLStringBuffer): String
+ getNavigatorLink(ISuite): String
IgnoredMethodsPanel
+ IgnoredMethodsPanel(Model)
+ getPrefix(): String
+ getHeader(ISuite): String
+ getContent(ISuite, XMLStringBuffer): String
+ getNavigatorLink(ISuite): String
TestNgXmlPanel
+ TestNgXmlPanel(Model)
+ getPrefix(): String
+ getHeader(ISuite): String
+ getContent(ISuite, XMLStringBuffer): String
+ getNavigatorLink(ISuite): String
TestPanel
+ TestPanel(Model)
+ getPrefix(): String
+ getHeader(ISuite): String
+ getContent(ISuite, XMLStringBuffer): String
+ getNavigatorLink(ISuite): String
+ getClassName(): String
TimesPanel
- m_totalTime: Map<String, Long> = Maps.newHashMap()
+ TimesPanel(Model)
+ getPrefix(): String
+ getHeader(ISuite): String
- js(ISuite): String
+ getContent(ISuite, XMLStringBuffer): String
- prettyDuration(long): String
+ getNavigatorLink(ISuite): String
ReporterPanel
+ ReporterPanel(Model)
+ getPrefix(): String
+ getHeader(ISuite): String
+ getContent(ISuite, XMLStringBuffer): String
+ getNavigatorLink(ISuite): String

Diagram 35. Template method. Enterprise Architect

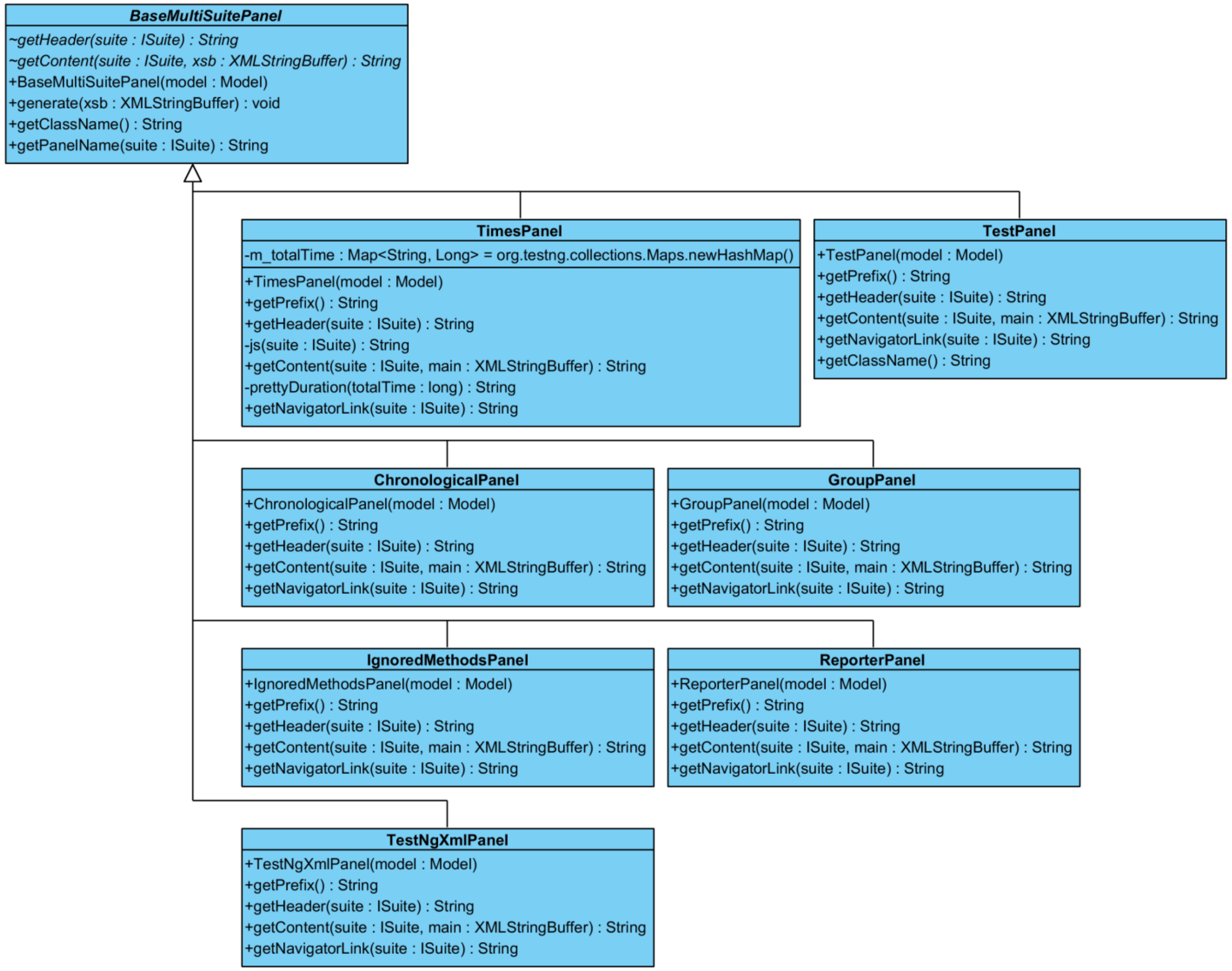

BaseMultiSuitePanel
~getHeader(suite : ISuite) : String
~getContent(suite : ISuite, xsb : XMLStringBuffer) : String
+BaseMultiSuitePanel(model : Model)
+generate(xsb : XMLStringBuffer) : void
+getClassName() : String
+getPanelName(suite : ISuite) : String
TimesPanel
-m_totalTime : Map<String, Long> = org.testng.collections.Maps.newHashMap()
+TimesPanel(model : Model)
+getPrefix() : String
+getHeader(suite : ISuite) : String
-js(suite : ISuite) : String
+getContent(suite : ISuite, main : XMLStringBuffer) : String
-prettyDuration(totalTime : long) : String
+getNavigatorLink(suite : ISuite) : String
TestPanel
+TestPanel(model : Model)
+getPrefix() : String
+getHeader(suite : ISuite) : String
+getContent(suite : ISuite, main : XMLStringBuffer) : String
+getNavigatorLink(suite : ISuite) : String
+getClassName() : String
ChronologicalPanel
+ChronologicalPanel(model : Model)
+getPrefix() : String
+getHeader(suite : ISuite) : String
+getContent(suite : ISuite, main : XMLStringBuffer) : String
+getNavigatorLink(suite : ISuite) : String
GroupPanel
+GroupPanel(model : Model)
+getPrefix() : String
+getHeader(suite : ISuite) : String
+getContent(suite : ISuite, main : XMLStringBuffer) : String
+getNavigatorLink(suite : ISuite) : String
IgnoredMethodsPanel
+IgnoredMethodsPanel(model : Model)
+getPrefix() : String
+getHeader(suite : ISuite) : String
+getContent(suite : ISuite, main : XMLStringBuffer) : String
+getNavigatorLink(suite : ISuite) : String
ReporterPanel
+ReporterPanel(model : Model)
+getPrefix() : String
+getHeader(suite : ISuite) : String
+getContent(suite : ISuite, main : XMLStringBuffer) : String
+getNavigatorLink(suite : ISuite) : String
TestNgXmlPanel
+TestNgXmlPanel(model : Model)
+getPrefix() : String
+getHeader(suite : ISuite) : String
+getContent(suite : ISuite, main : XMLStringBuffer) : String
+getNavigatorLink(suite : ISuite) : String

Diagram 36. Template method. Visual Paradigm

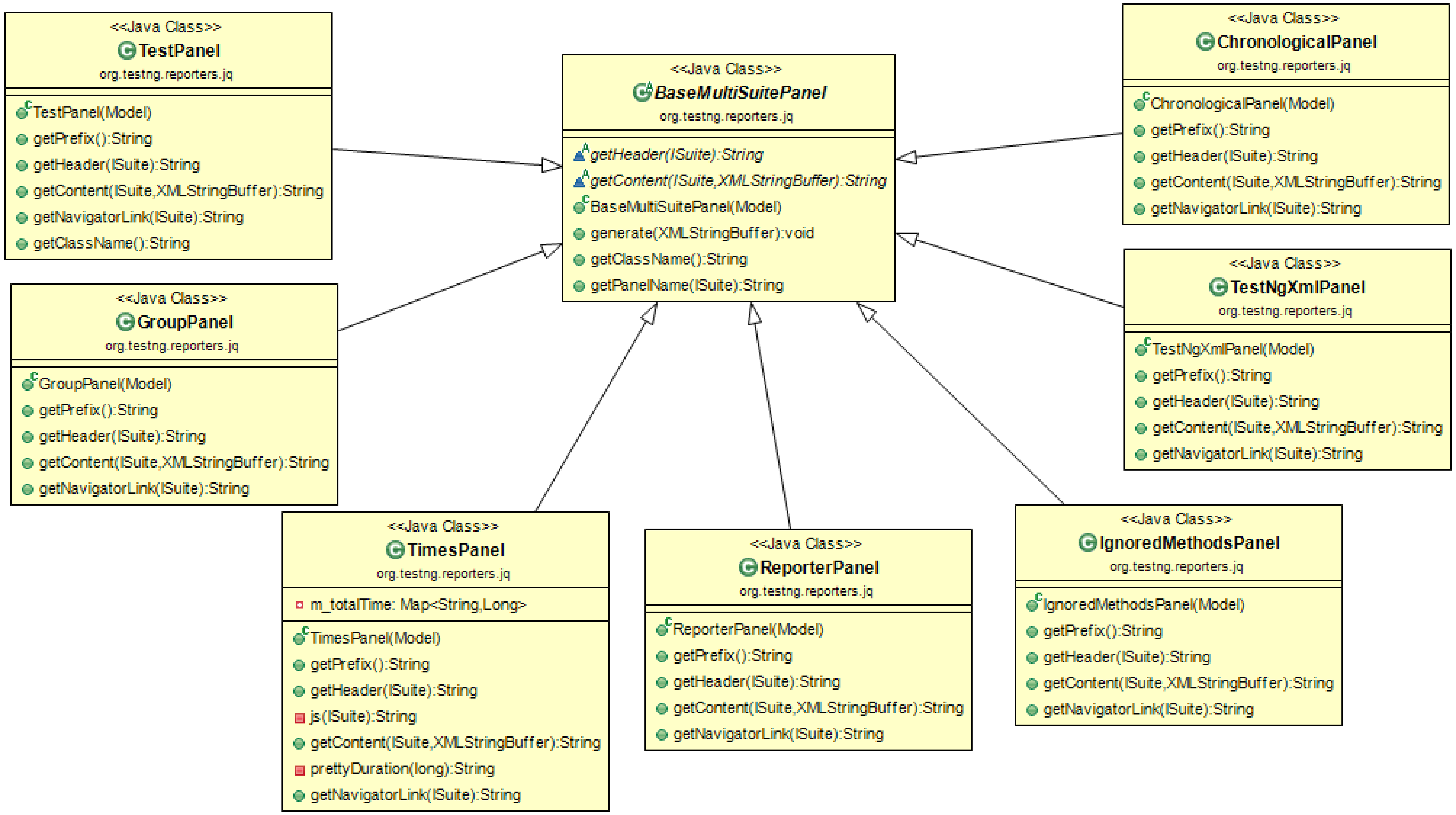


Diagram 37. Template method. The ObjectAid UML Explorer

3.10 Command

3.10.1 Pattern definition

Command is behavioural design pattern and is used to encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. [1]

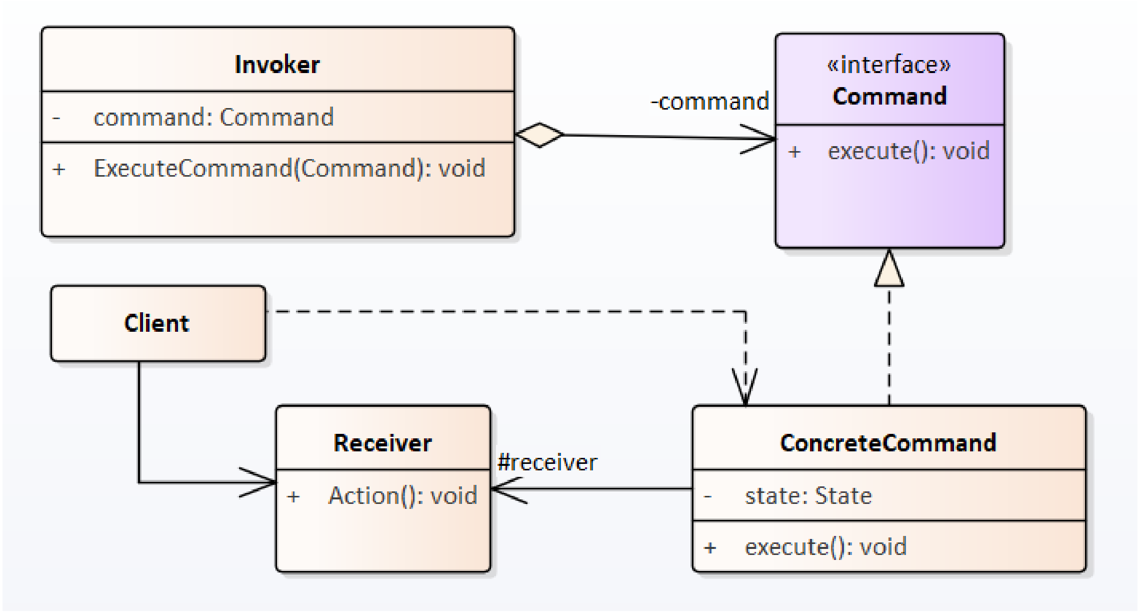


Diagram 38. Command pattern

3.10.2 Java project description

Project from “Design patterns implemented in Java” project in GitHub [12] is used for command design pattern analysis. This project contains example implementation for many design patterns.

There is a wizard (invoker) who can cast a spell (command). Wizard can cast two spells: invisibility spell and shrink spell (concrete commands). Spells store state and target. It allows to revert spells or execute them again on the same target. Wizard cast spells on a target (receiver). Goblin is the only one available target.

3.10.3 Pattern UML presentation analysis

All CASE tools correctly identified

- generalisation between Command class and concrete spells, Target class and Goblin class
- association between concrete spells and Target class

Enterprise architect and The ObjectAid UML Explorer recognised that Wizard has a dependency relationship with Target. Visual paradigm could not find dependency between them.

All CASE tools see redoStack and undoStack properties for Wizard, but only The ObjectAid UML Explorer displays it as association with Command class with zero or more multiplicity.

The ObjectAid UML Explorer determined all parts of design pattern and got 3 points. Enterprise Architect could not find one relationship and gets 2 points. Visual Paradigm could not identify two separate relationships and gets 1 point.

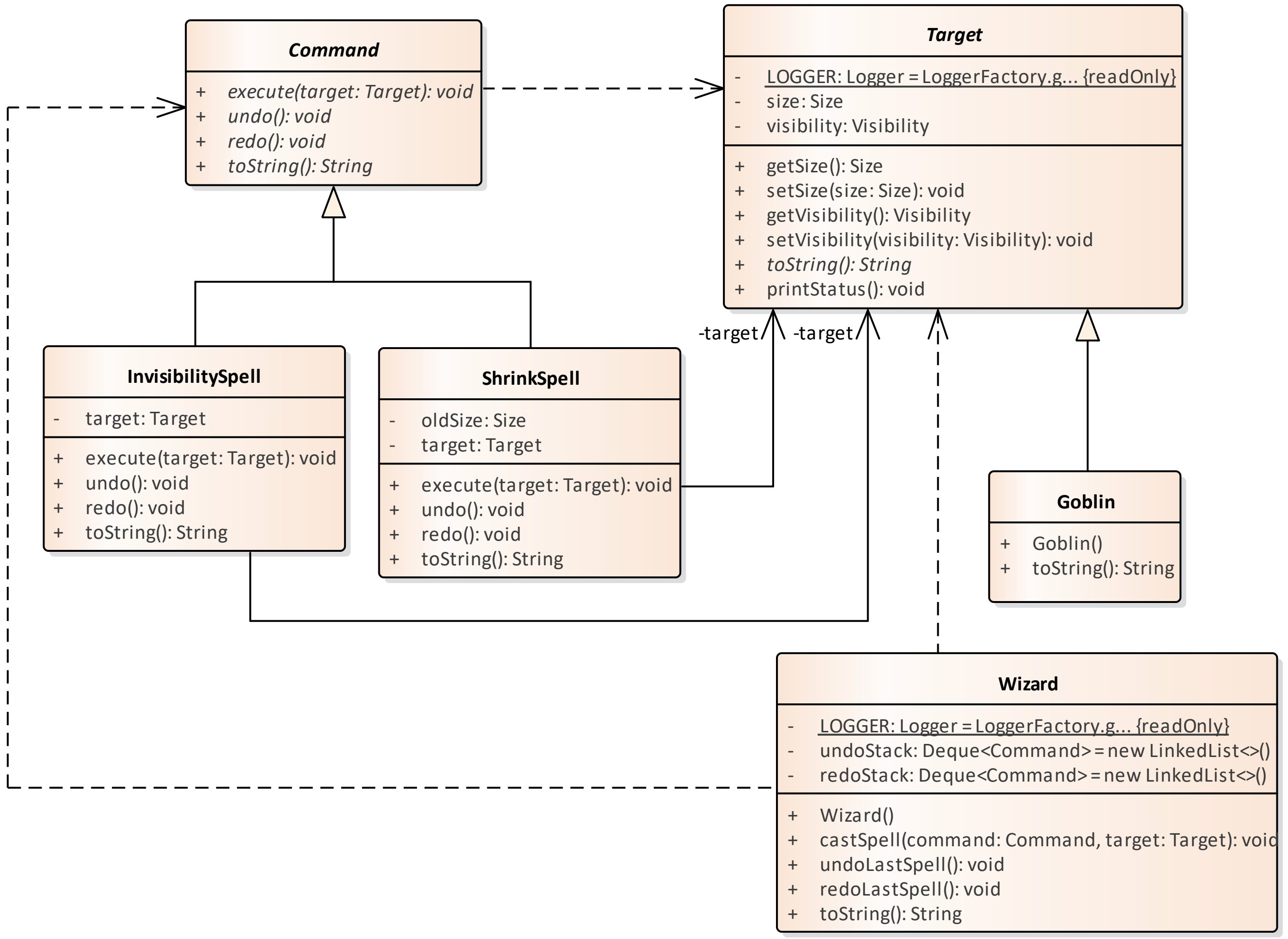


Diagram 39. Command. Enterprise Architect

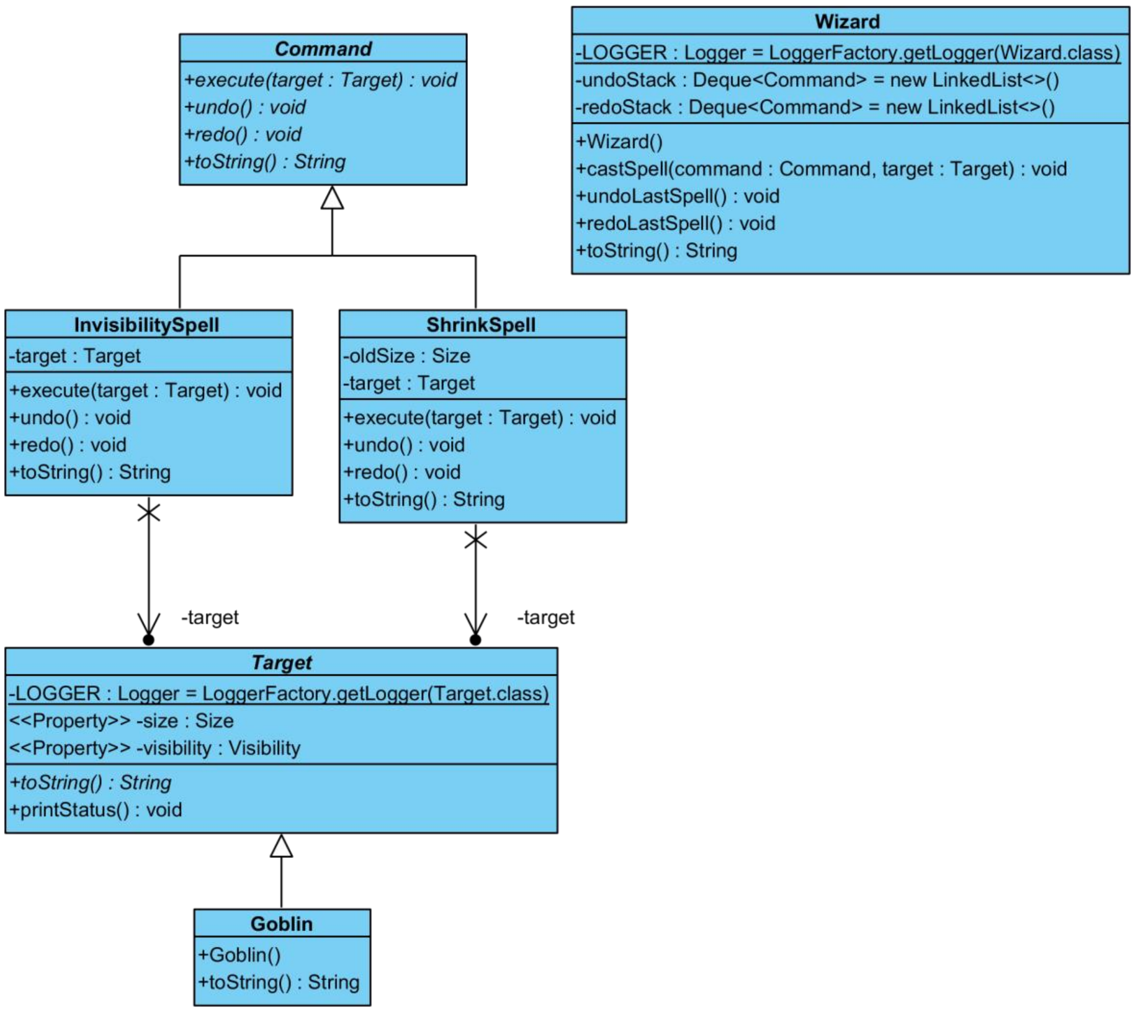

Command
+execute(target : Target) : void
+undo() : void
+redo() : void
+toString() : String
Wizard
-LOGGER : Logger = LoggerFactory.getLogger(Wizard.class)
-undoStack : Deque<Command> = new LinkedList<>()
-redoStack : Deque<Command> = new LinkedList<>()
+Wizard()
+castSpell(command : Command, target : Target) : void
+undoLastSpell() : void
+redoLastSpell() : void
+toString() : String
InvisibilitySpell
-target : Target
+execute(target : Target) : void
+undo() : void
+redo() : void
+toString() : String
ShrinkSpell
-oldSize : Size
-target : Target
+execute(target : Target) : void
+undo() : void
+redo() : void
+toString() : String
-target
-target
Target
-LOGGER : Logger = LoggerFactory.getLogger(Target.class)
<<Property>> -size : Size
<<Property>> -visibility : Visibility
+toString() : String
+printStatus() : void
Goblin
+Goblin()
+toString() : String

Diagram 40. Command. Visual Paradigm

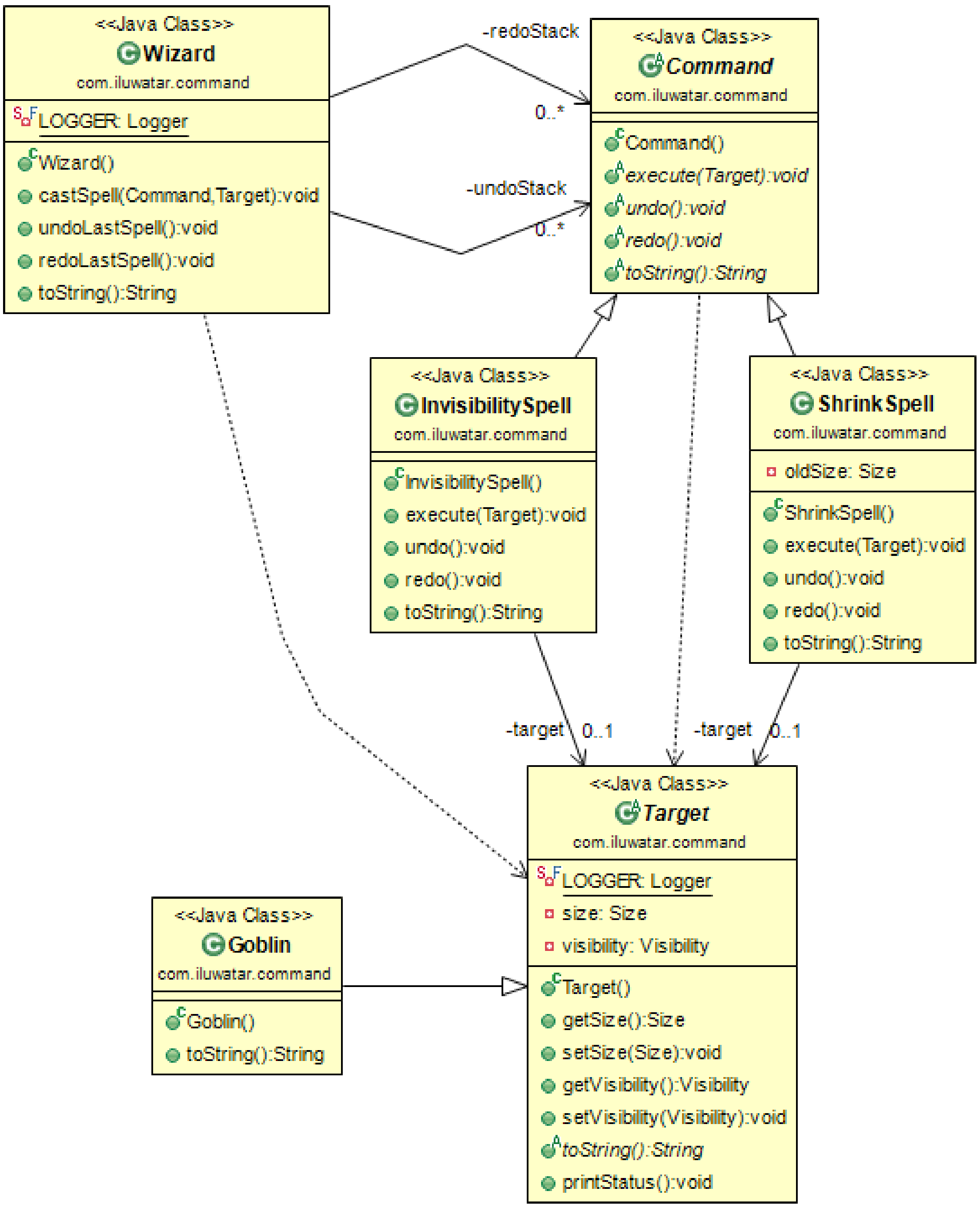


Diagram 41. Command. The ObjectAid UML Explorer

3.11 Builder

3.11.1 Pattern definition

The builder is the creational pattern used to separate the construction of a complex object from its representation so that the same construction process can create different representations. [1]

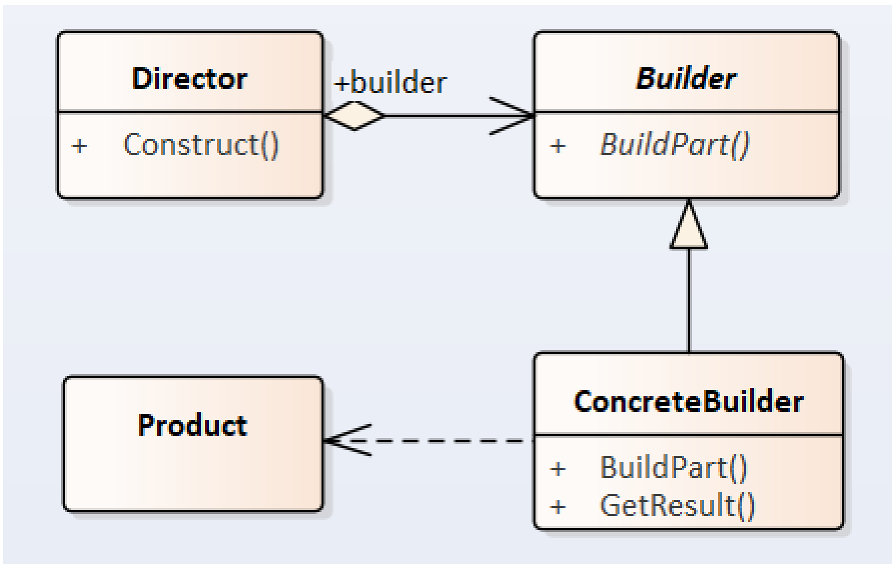


Diagram 42. Builder pattern

3.11.2 Java project description

TestNG test framework project [11] is used for builder pattern analysis.

The builder is implemented as a nested class for Arguments class. Arguments class is nested inside AbstractParallelWorker class. TestRunner calls arguments builder when the createWorkers function is executed.

In this implementation of builder pattern, builder interface is not used. Because of that, expect that director will have association or dependency relationship with concrete builder implementation.

3.11.3 Pattern UML presentation analysis

All CASE tools identified that builder has an association with Arguments and stores instance of Arguments class as private property.

Visual Paradigm and The ObjectAid UML explorer could identify nesting of classes. Enterprise architect recognised that Arguments are nested inside AbstractParallelWorker but could not determine that Builder is nested inside Arguments.

Enterprise Architect and Visual Paradigm could not identify that TestRunner class have a dependency on Builder, Arguments and AbstractParallelWorker classes and get 2 points. The ObjectAid UML Explorer identifies dependencies for TestRunner class correctly and gets 3 points.

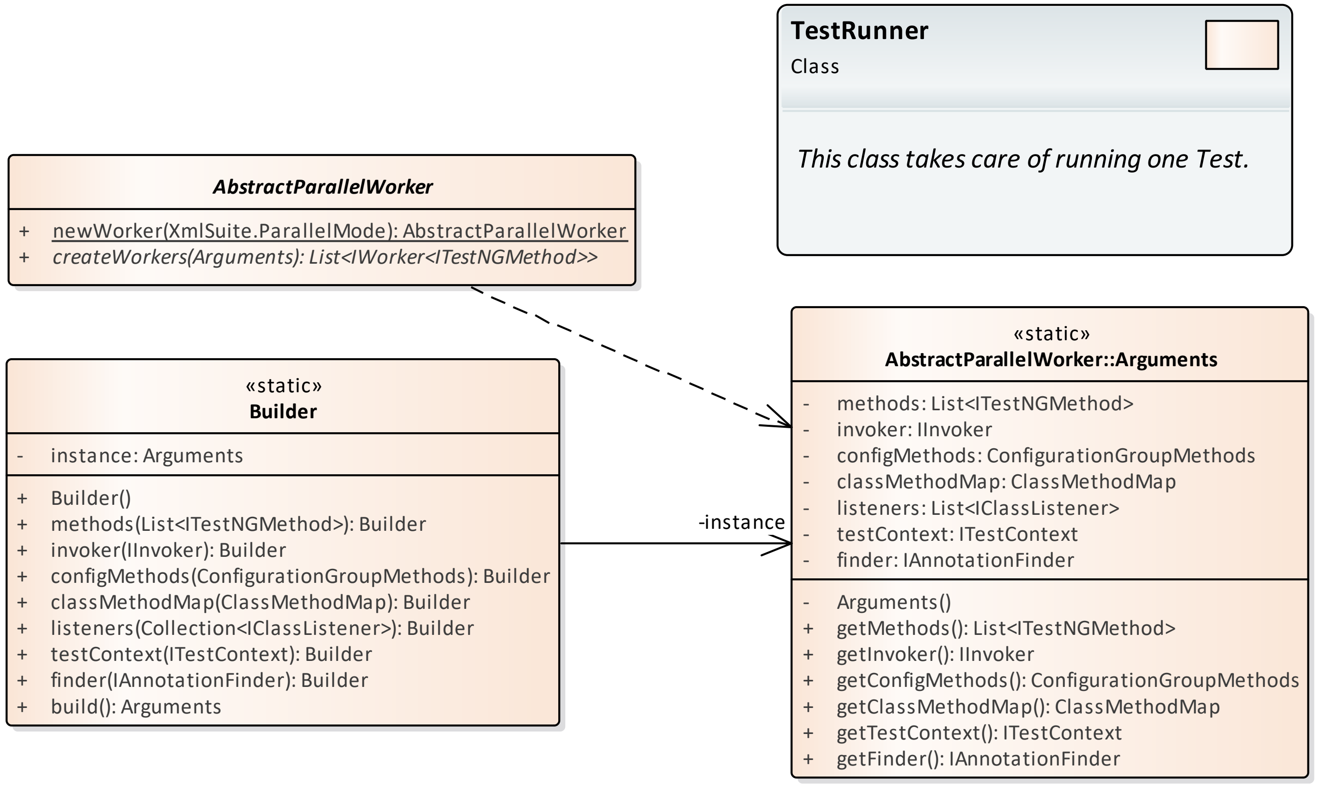
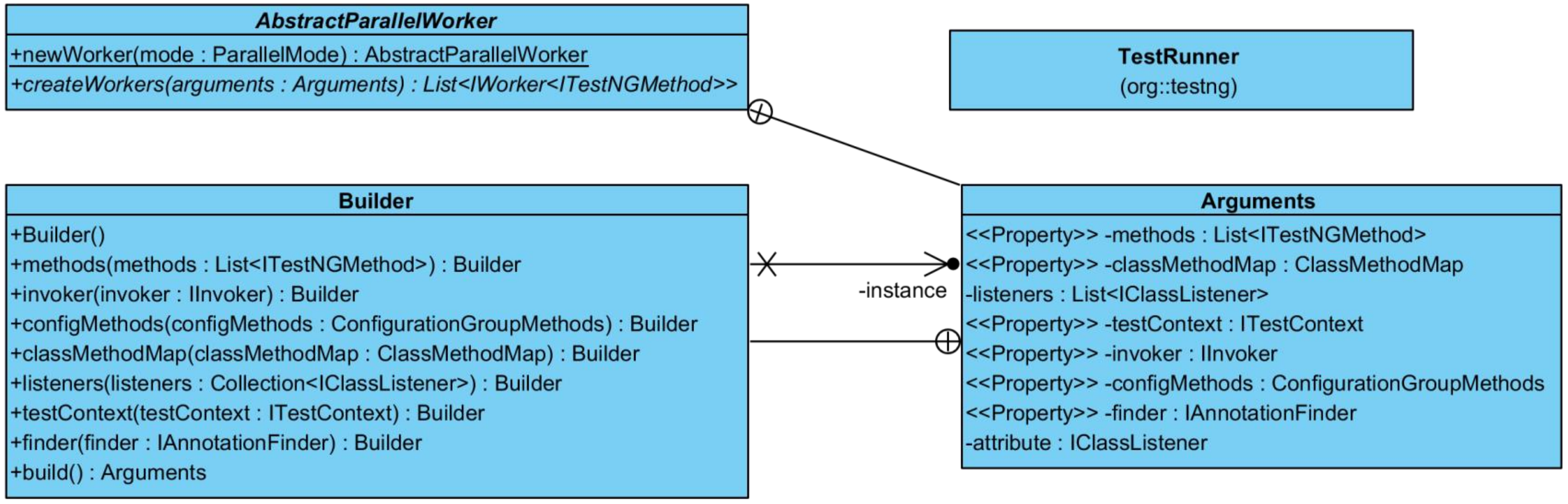
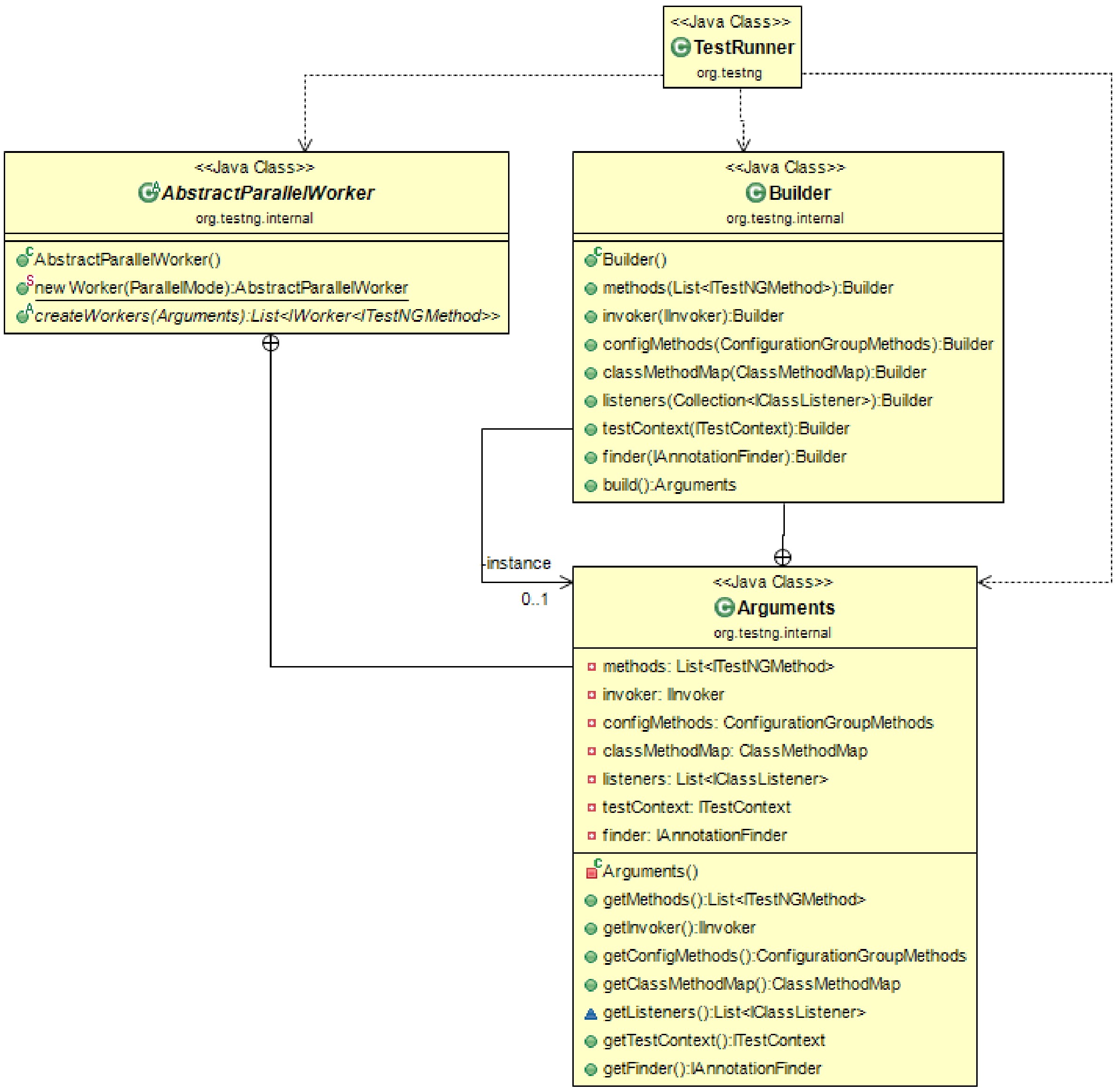

TestRunner
Class
This class takes care of running one Test.
AbstractParallelWorker
+ newWorker(XmlSuite.ParallelMode): AbstractParallelWorker
+ createWorkers(Arguments): List<IWorker<ITestNGMethod>>
«static»
Builder
- instance: Arguments
+ Builder()
+ methods(List<ITestNGMethod>): Builder
+ invoker(IInvoker): Builder
+ configMethods(ConfigurationGroupMethods): Builder
+ classMethodMap(ClassMethodMap): Builder
+ listeners(Collection<IClassListener>): Builder
+ testContext(ITestContext): Builder
+ finder(IAnnotationFinder): Builder
+ build(): Arguments
-instance
«static»
AbstractParallelWorker::Arguments
- methods: List<ITestNGMethod>
- invoker: IInvoker
- configMethods: ConfigurationGroupMethods
- classMethodMap: ClassMethodMap
- listeners: List<IClassListener>
- testContext: ITestContext
- finder: IAnnotationFinder
- Arguments()
+ getMethods(): List<ITestNGMethod>
+ getInvoker(): IInvoker
+ getConfigMethods(): ConfigurationGroupMethods
+ getClassMethodMap(): ClassMethodMap
+ getTestContext(): ITestContext
+ getFinder(): IAnnotationFinder

Diagram 43. Builder. Enterprise Architect



AbstractParallelWorker
+newWorker(mode : ParallelMode) : AbstractParallelWorker
+createWorkers(arguments : Arguments) : List<IWorker<ITestNGMethod>>
TestRunner
(org::testng)
Builder
+Builder()
+methods(methods : List<ITestNGMethod>) : Builder
+invoker(invoker : IInvoker) : Builder
+configMethods(configMethods : ConfigurationGroupMethods) : Builder
+classMethodMap(classMethodMap : ClassMethodMap) : Builder
+listeners(listeners : Collection<IClassListener>) : Builder
+testContext(testContext : ITestContext) : Builder
+finder(finder : IAnnotationFinder) : Builder
+build() : Arguments
-instance
Arguments
<<Property>> -methods : List<ITestNGMethod>
<<Property>> -classMethodMap : ClassMethodMap
-listeners : List<IClassListener>
<<Property>> -testContext : ITestContext
<<Property>> -invoker : IInvoker
<<Property>> -configMethods : ConfigurationGroupMethods
<<Property>> -finder : IAnnotationFinder
-attribute : IClassListener

Diagram 44. Builder. Visual paradigm



<<Java Class>>
TestRunner
org.testng
<<Java Class>>
AbstractParallelWorker
org.testng.internal
AbstractParallelWorker()
new Worker(ParallelMode):AbstractParallelWorker
createWorkers(Arguments):List<IWorker<ITestNGMethod>>
<<Java Class>>
Builder
org.testng.internal
Builder()
methods(List<ITestNGMethod>):Builder
invoker(IInvoker):Builder
configMethods(ConfigurationGroupMethods):Builder
classMethodMap(ClassMethodMap):Builder
listeners(Collection<IClassListener>):Builder
testContext(ITestContext):Builder
finder(IAnnotationFinder):Builder
build():Arguments
-instance
0..1
<<Java Class>>
Arguments
org.testng.internal
methods: List<ITestNGMethod>
invoker: IInvoker
configMethods: ConfigurationGroupMethods
classMethodMap: ClassMethodMap
listeners: List<IClassListener>
testContext: ITestContext
finder: IAnnotationFinder
Arguments()
getMethods():List<ITestNGMethod>
getInvoker():IInvoker
getConfigMethods():ConfigurationGroupMethods
getClassMethodMap():ClassMethodMap
getListeners():List<IClassListener>
getTestContext():ITestContext
getFinder():IAnnotationFinder

Diagram 45. Builder. The ObjectAid UML Explorer

3.12 State

3.12.1 Pattern definition

State is behavioural design pattern used to allow an object to alter its behaviour when its internal state changes. The object will appear to change its class. [1]

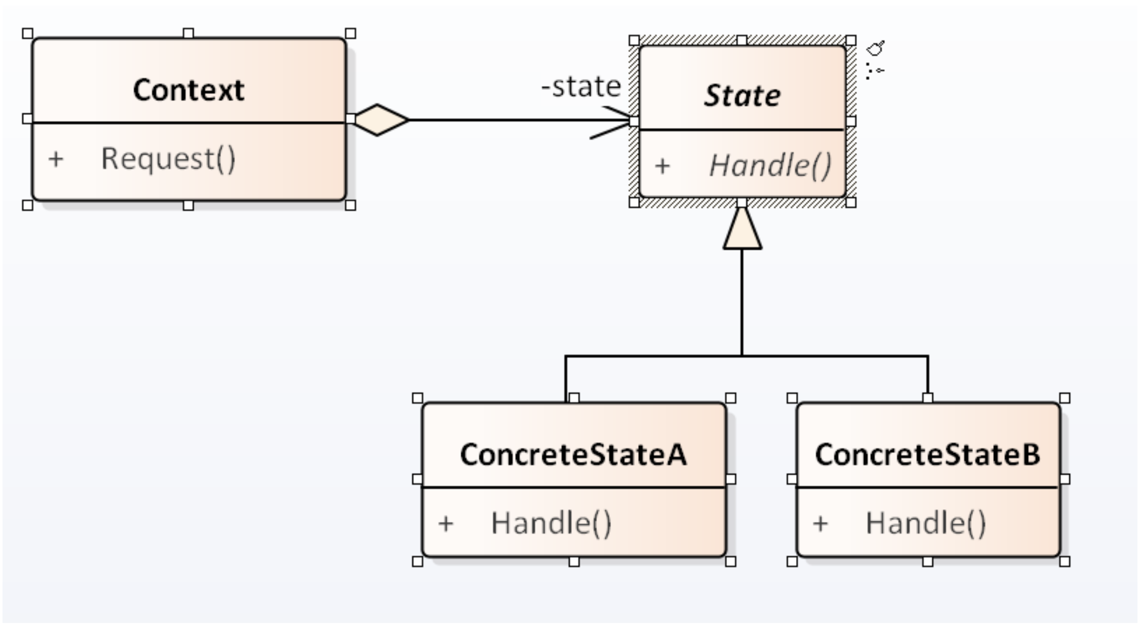


Diagram 46. State pattern

3.12.2 Java project description

Small java application that the author wrote as an exercise in Software Architecture and Design course (IDU1550) [16]is used for state pattern analysis.

This application implements logic for the gateway that opens only when a person pays for entrance and closes when one person passes the gate. If a person tries to pass without paying, an alarm is triggered.

3.12.3 Pattern UML presentation analysis

All CASE tools identified a dependency between concrete state implementations (ClosedStatus and OpenStatus) and GatewayStatus interface. All of them have the same set of methods.

All CASE tools identified that GatewayImpl class have private property with GatewayStatus type called status, but none of them identified an association between this classes as aggregation.

Each CASE tool differently identified multiplicity for this association. Enterprise Architect does not have any multiplicity. Visual paradigm identified it as precisely one instance. The ObjectAid UML Explorer classified multiplicity as zero instances or one instance. In this case, the Visual Paradigm is the CASE tools that identified multiplicity correctly because the status property has an initial value that is not null.

All CASE tools identified all main parts of design pattern, and all CASE tools get 3 points.

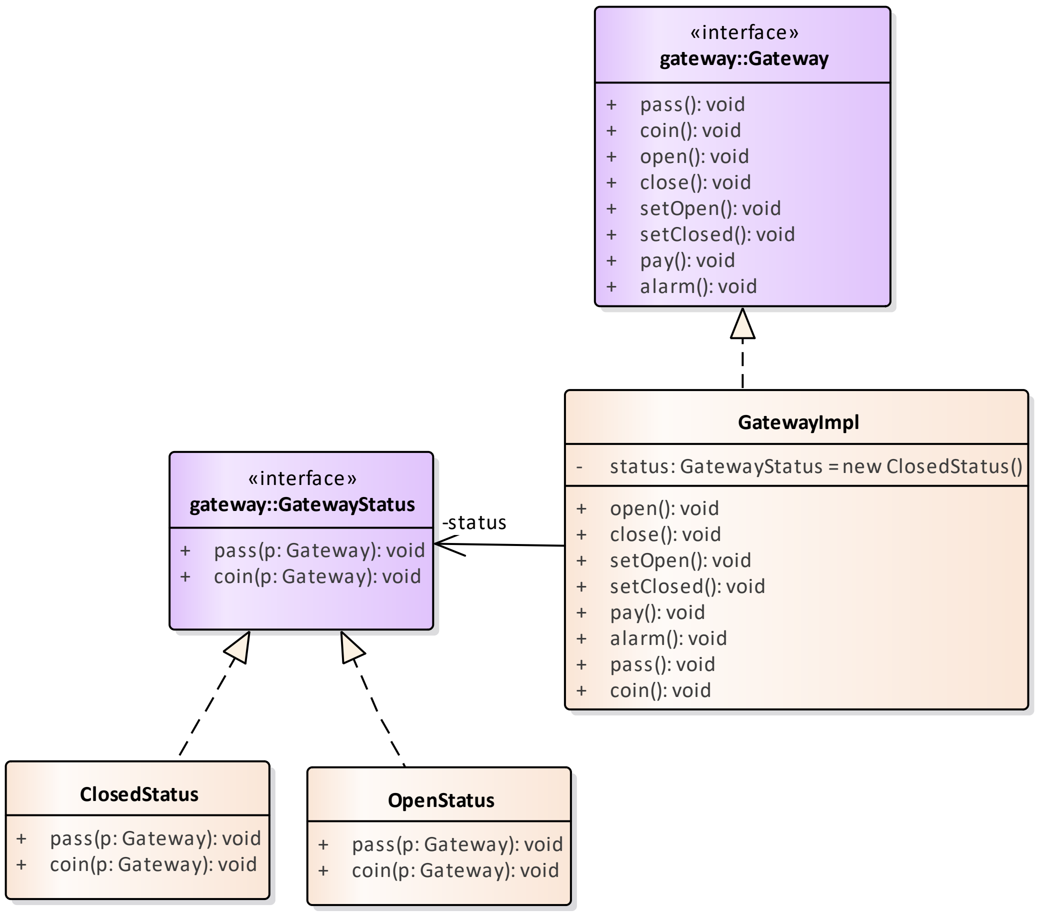


Diagram 47. State. Enterprise Architect

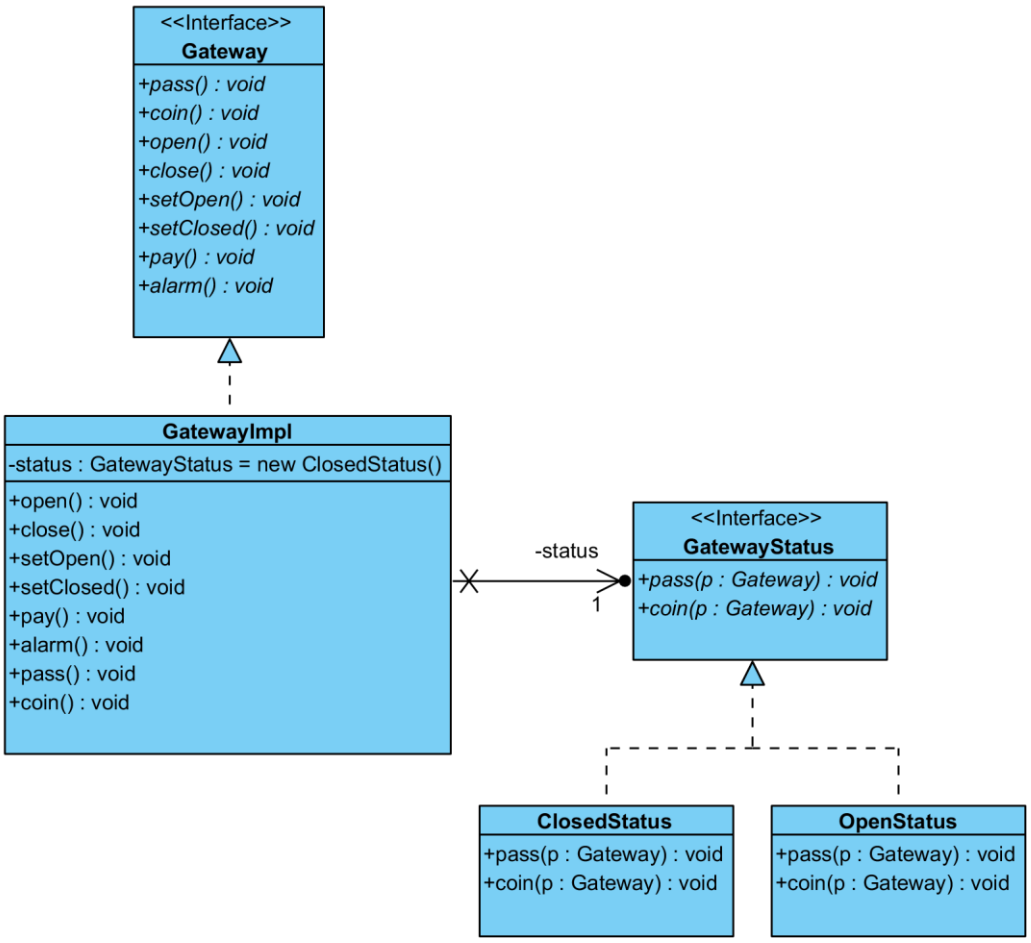


Diagram 48. State. Visual Paradigm

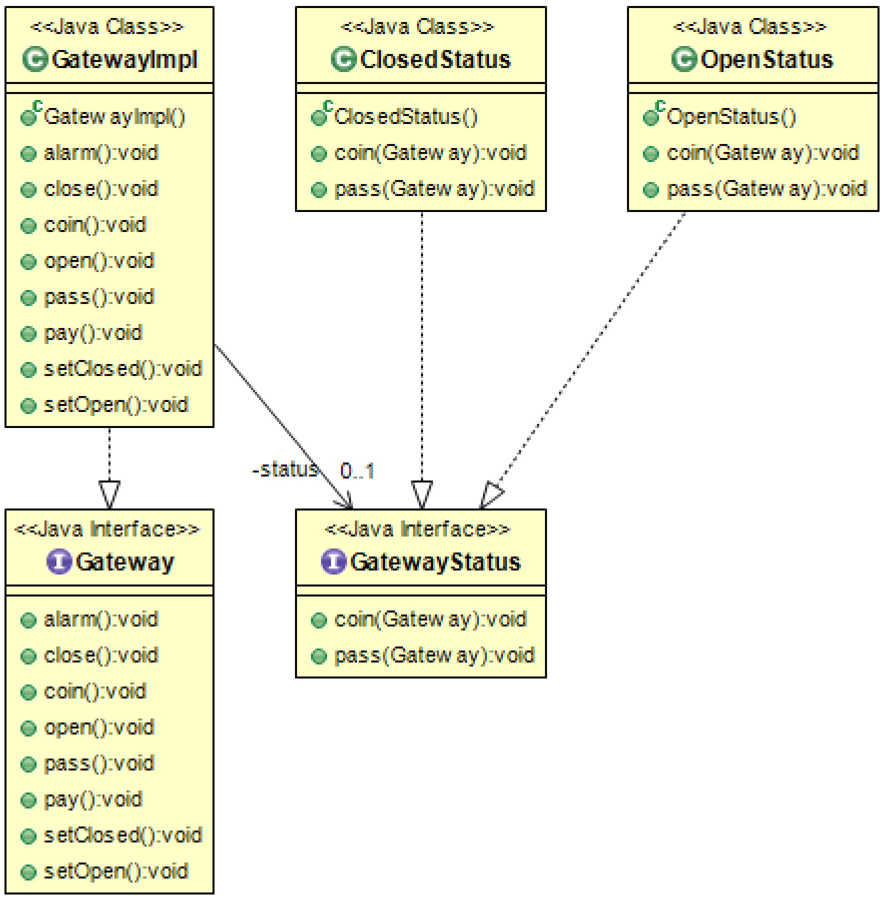


Diagram 49. State. The ObjectAid UML Explorer

3.13 Summary for reverse engineering to class diagram

This section contains summary table based on analysis of design patterns representation for three CASE tools used for analysis and voting table for each CASE tool pair [20]. All GOF design patterns are counted as equally important.

The table contains assessment for each pattern and CASE tool pair. Assessment scale is based on following values:

0 – could not identify any parts of a design pattern

1 – more than one expected part is missing

2 – identify almost everything except one part

3 – identify all parts of a design pattern

Table 1. Points summary table for GOF pattern representation assessment of CASE tools

GOF pattern	**Enterprise Architect**	**Visual Paradigm**	**The ObjectAid UML Explorer for Eclipse**
Singleton	3	3	3
Factory method	2	2	2
Observer	2	0	0
Strategy	3	2	3
Iterator	2	0	1
Bridge	3	2	3
Adapter	3	3	3
Composite	2	2	2
Template method	3	3	3
Command	2	1	3
Builder	2	2	3
State	3	3	3
Total	30	23	29

Table 2. Voting table for GOF pattern representation assessment of CASE tools

	Enterprise Architect	**The ObjectAid UML Explorer for Eclipse**	**Visual Paradigm**	**Votes**
Enterprise Architect	0	2	5	7
The ObjectAid UML Explorer for Eclipse	2	0	5	7
Visual Paradigm	0	0	0	0

In table 2, CASE tool pairs are compared with each other for each GOF pattern. If one CASE tool is better than another is, then it gets 1 point. Usually, the sum of points for each pair is added to the table. Then all point for each row sums up to get a final grade. The more correct approach based on minimum FAS (Feedback Arc Set) [5] tries to find such a reordering of the voting table so that the sum of cells below the main diagonal is minimal. In this case, both approaches give the same reordering.

We can see that Enterprise Architect and The ObjectAid UML Explorer are better that Visual paradigm for five GOF design patterns. Enterprise Architect is better than The ObjectAid UML Explorer for two GOF design patterns, but The ObjectAid UML Explorer is better than Enterprise Architect for two GOF design patterns as well.

As a result, Enterprise Architect and The ObjectAid UML Explorer have same grades in the voting table, but Enterprise Architect does not have any 0 points in Table 1 and have more total points. Based on that, Enterprise Architect is the best CASE tool for three selected CASE tools.

These results are based on the assumption that the reverse engineering of each GOF design pattern is equally important. If the importance varies, then one would have to weigh the points and votes in tables 1 and 2. However, since it is difficult to find information about the relative importance different patterns (and varying importance might depend on the individual development project), this additional weighing was left out of the approach used in this thesis.

4 Sequence diagram reverse engineering

Analysis of sequence diagram reverse engineering is focused on basic reverse engineering functionality, such as identifying class instances used in reverse engineered method, identifying messages sent to that class instances, representing them correctly. Simple java project was used for that purpose.

The scope of this analysis is limited and more complex projects (with asyncronouse calls or loops) can reveal other strengths and weaknesses of used CASE tool. This can influence the current ranking of CASE tools and change results.

4.1 Java project description

Small java application that author wrote as an exercise in Software Architecture and Design course (IDU1550) [16] is used to analyse sequence diagrams functionality.

This application implements logic for the gateway, that opens only when a person pays for entrance and closes when one person passes the gate. If a person tries to pass without paying, an alarm is triggered. Gateway interface has two realisations. The first realisation is using State GOF pattern. The second realisation is using State GOF pattern and Observer GOF pattern to notify about payments and alarms.

For reverse engineering of the sequence diagram based on this application, main() method is used. This method is executing the same sequence of actions for each realisation of Gateway interface.

```
public class Main {

    public static void main(String[] args) {
        System.out.println("-------- v1 ----------");
        Gateway gate = new GatewayImpl();
        start(gate);

        System.out.println("\n-------- v2 ----------");
        GatewayWithObserver gate2 = new GatewayWithObserver();
        StatusObserver observer = new StatusObserver();
        gate2.addObserver(observer);
        start(gate2);
    }
    private static void start(Gateway gate) {
        gate.pass();
        gate.coin();
        gate.coin();
        gate.pass();
        gate.pass();
    }
}
```

4.2 Analysis of sequence diagrams

4.2.1 Enterprise Architect

Enterprise Architect requires executing the code in order to reverse engineer the code to sequence diagram. It is using debugging functionality to record execution steps. To record each function call user requires going through each class and marking it with recording marker. It is time consuming and might be hard to setup for libraries.

Lifeline in sequence diagram created by Enterprise Architect contains only concrete classes. It uses anonymous instances for these classes. In this case, a lifeline has ":className" naming format. When named instances are used lifeline name have format "instanceFormat:className". Sequence diagram does not have numbering for messages.

Sequence diagram does not contain information about initialisation of StatusObserver and GatewayImpl classes but contains a message that triggers the creation of GatewayWithObserver class. It is related to the fact GatewayWithObserver have a constructor defined in executed code. StatusObserver and GatewayImpl classes use the default constructor, and there is no way to set recorder marker for this type of constructor.

In UML sequence diagrams when the object is created, it should be shown on the same height as creation message [21] and use dashed line for the message. When GatewayWithObserver is created for instance lifeline is not shown on the same level as creation message, and a solid line is used.

Sequence diagram contains all execution of methods marked for recording. However, the original code includes calls to classes and methods that are part of Java core libraries. For example, GatewayWithObserver class extends java.util.Observable [22] class. The main method calls an addObserver method on GatewayWithObserver, but it is not recorded, because you cannot set recorder marker on external library code.

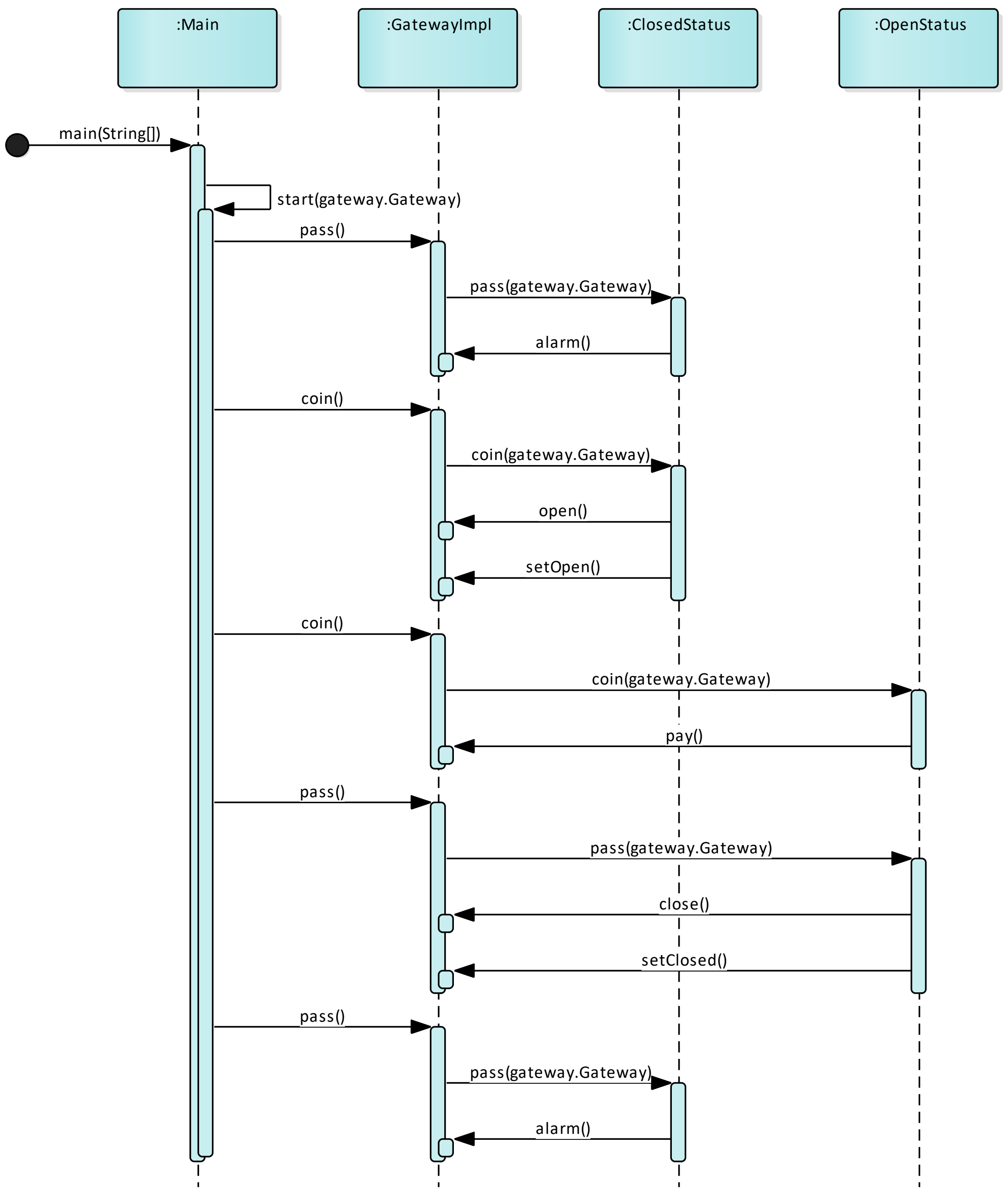


Diagram 50. GatewayImpl execution. Enterprise Architect sequence diagram.

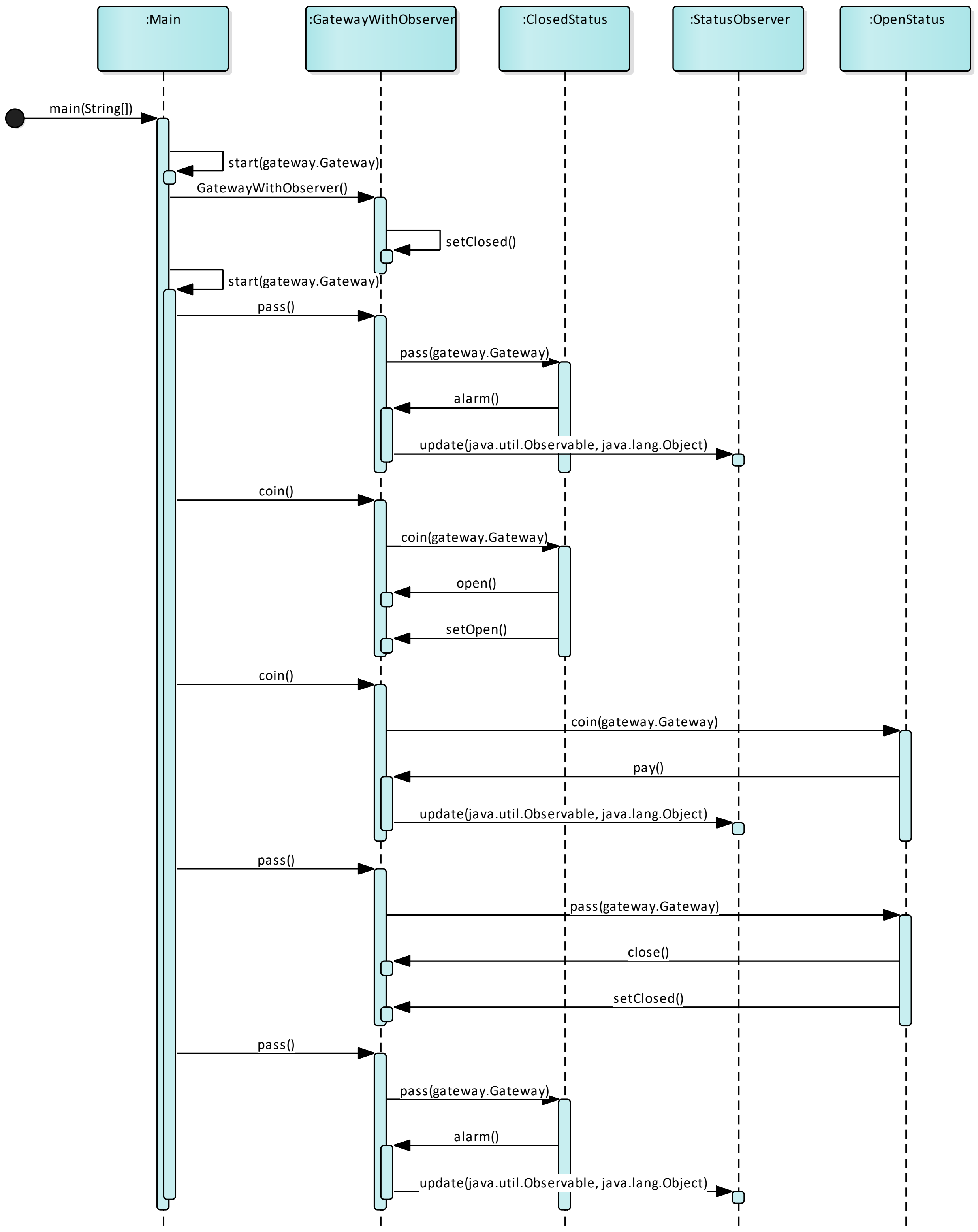


Diagram 51. GatewayWithObserver execution. Enterprise Architect sequence diagram.

4.2.2 Visual Paradigm

The visual paradigm can generate sequence diagrams without code execution. Initially, it creates sequence diagram only for a specified method and does not go deeper (Diagram 56).

On generated diagram lifelines use named instances, and instance name is equal to the variables used in the code. The diagram contains numbering for messages, and it uses nested numbering by default, but the user can change it to single level numbering.

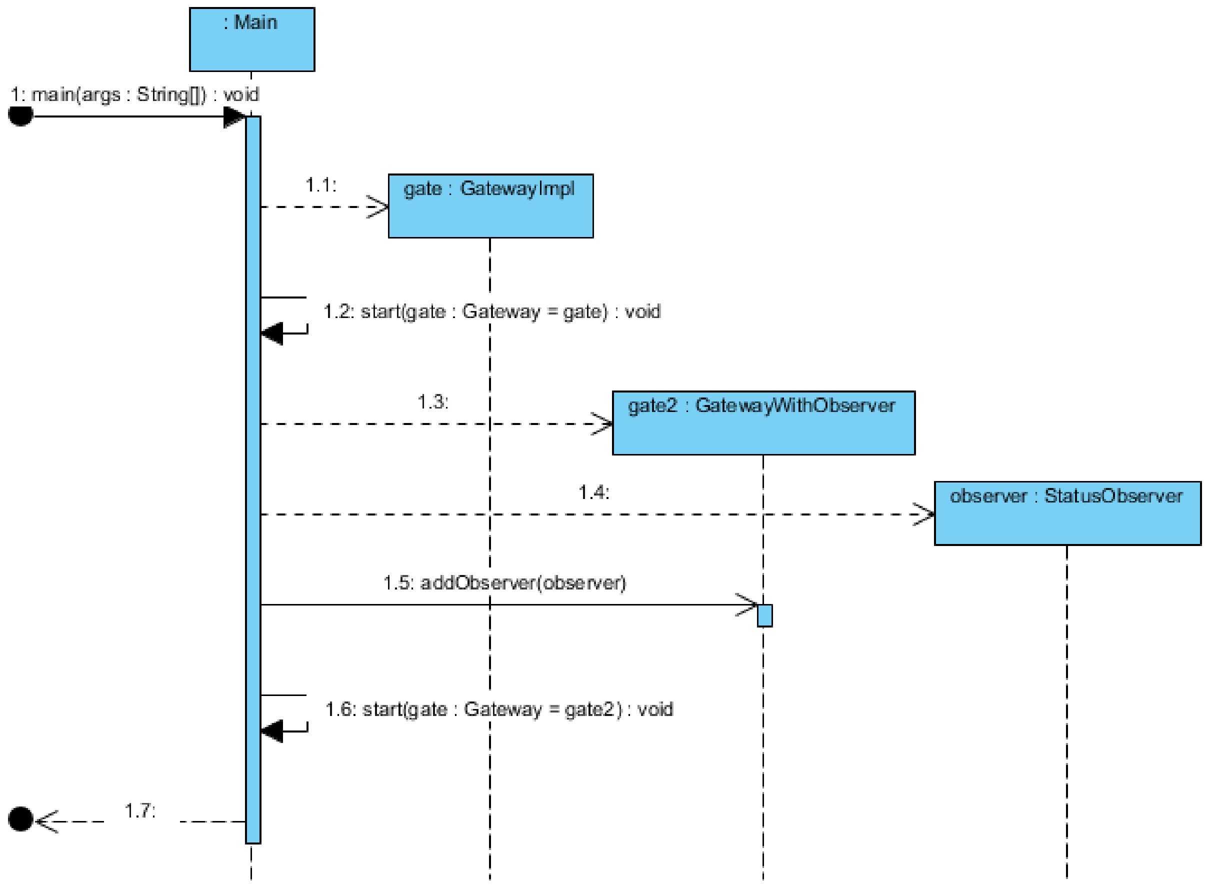
Visual paradigm was able to find creation messages for all classes used in the main method. It is showing it with dashed line and instance placed on the same level as creation message.

Visual paradigm was able to find and display call to addObserver method that is part of extended java.util.Observable [23] class, but it does not show calls to System.out.println() method.

If the user needs to go deeper into the code, then right-click on a method that needs to be reverse engineered in sequence diagram and select "Instant Reverse Java Code". The author tried this with both start method calls (Diagram 57).

After that numbering on diagram got broken. On diagram with a deeper level of code, details numbering becomes mixed. Visual paradigm uses nested numbering before start method call and single level numbering after it.

Visual paradigm was unable to understand that in message number 1.2 instance of GatewayImpl class was passed to start method and in message number 11 instance of GatewayWithObserver was used. Instead, it created new lifelines for two instances of Gateway interface used by two start method calls. Because pass() and coin() messages are passed to Gateway interface, there is no way to go even deeper in code and see on the same diagram that pass() and coin() methods do. To see that pass() and coin() methods do user need to create a new sequence diagram for each class method.



: Main
1: main(args : String[]) : void
1.1:
gate : GatewayImpl
1.2: start(gate : Gateway = gate) : void
1.3:
gate2 : GatewayWithObserver
1.4:
observer : StatusObserver
1.5: addObserver(observer)
1.6: start(gate : Gateway = gate2) : void
1.7:

Diagram 52. Main execution, one level deep. Visual paradigm sequence diagram.

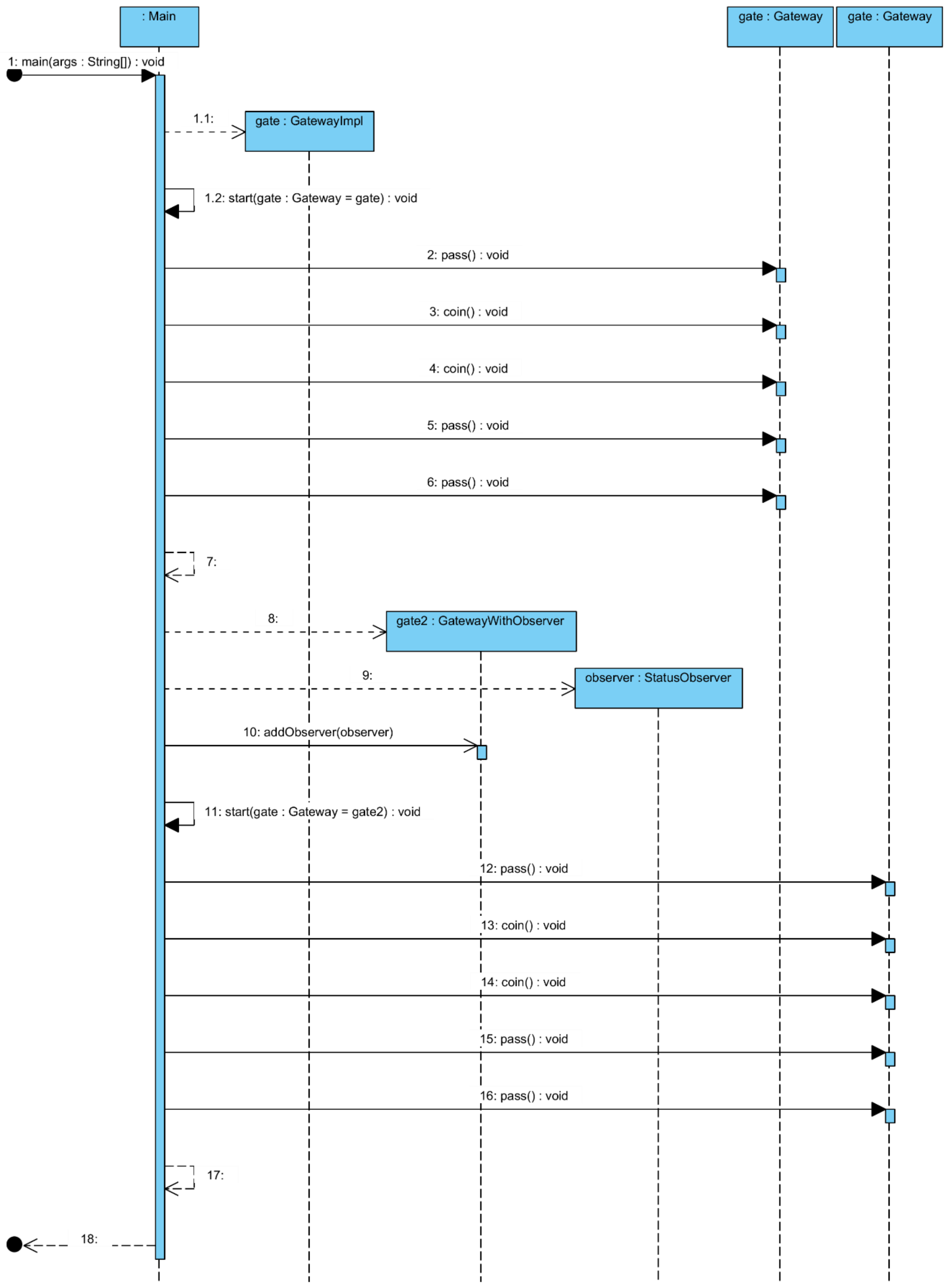


Diagram 53. Main execution, two levels deep. Visual paradigm sequence diagram.

4.2.3 The ObjectAid UML Explorer for Eclipse

The ObjectAid UML Explorer for Eclipse does not need to execute code to generate sequence diagram. The user can drag and drop required method on the diagram, then right-click on the method name and select “Add Called Operations”. This action will show only calls for methods, which are one level deep (Diagram 58). If the user needs to go deeper in the code repeating of the same operation on required method will help.

For lifelines name The ObjectAid UML Explorer uses named classes in case if the variable was created for a class in the main method and anonymous instances for classes that do not have variables created in the specified method. Sequence diagram does not contain numbering for messages.

The CASE tool found all initialisation calls for all classes initialised in the main method. It displays it using a solid line, and lifeline boxes are not shown on the same level as creation message.

The CASE tool found all calls to user code and to Java core libraries including a call to the addObserver method for GatewayWithObserver class that is part of extended java.util.Observable [23] class and println method call for java.io.PrintStream.

When tried to go more in-depth for the start method, the CASE tool could not understand that instance of the concrete class was passed to it and created to a lifeline for Gateway interface instance and showed all calls to it instead of concrete class.

The addObserver method call is marked as critical by the CASE tool. It means that this code can be executed only by one thread at the time. It is valid marking because addObserver [22] is a synchronised method [24].

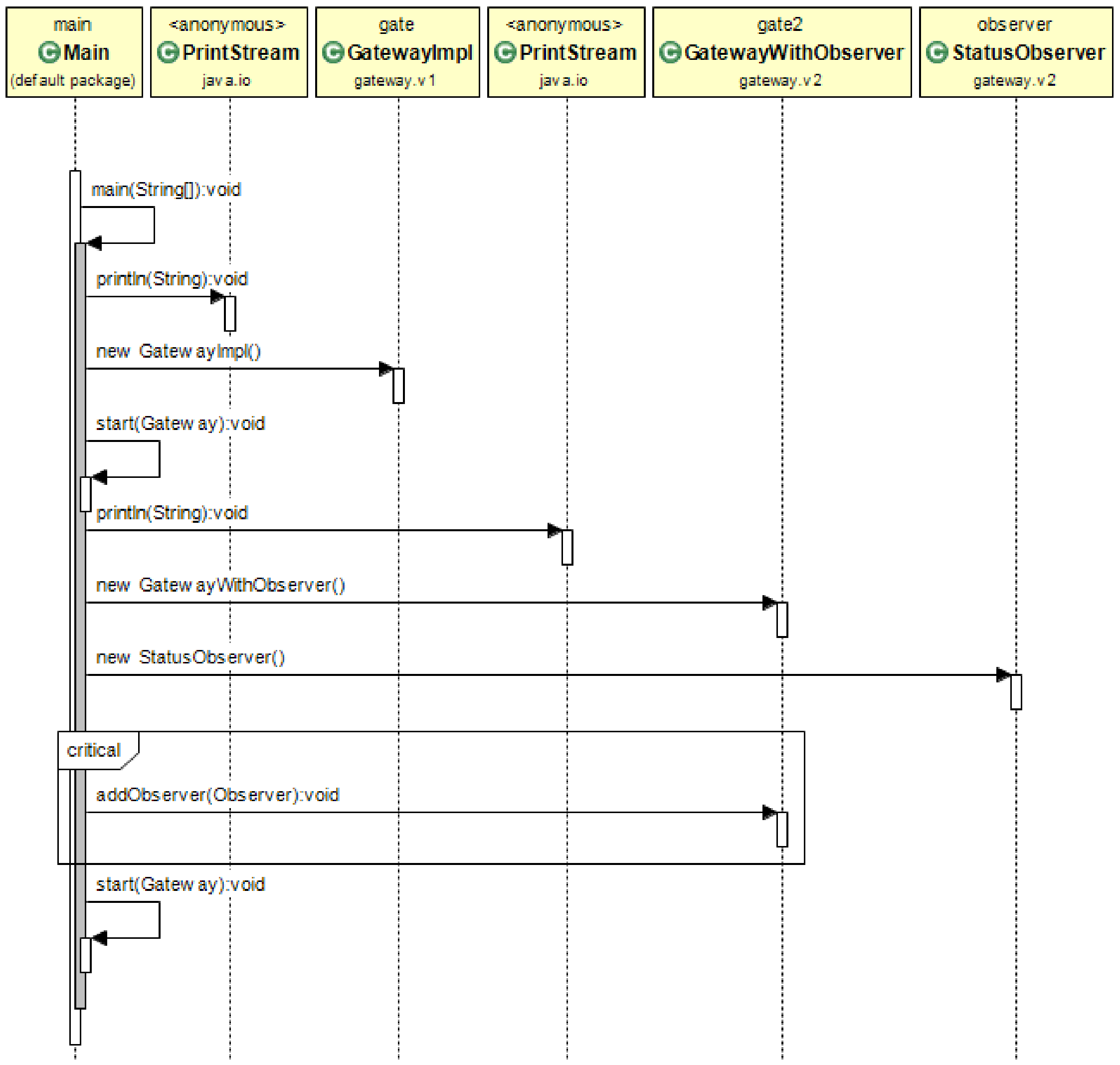


Diagram 54. Main function one level deep. The ObjectAid UML Explorer for Eclipse sequence diagram.

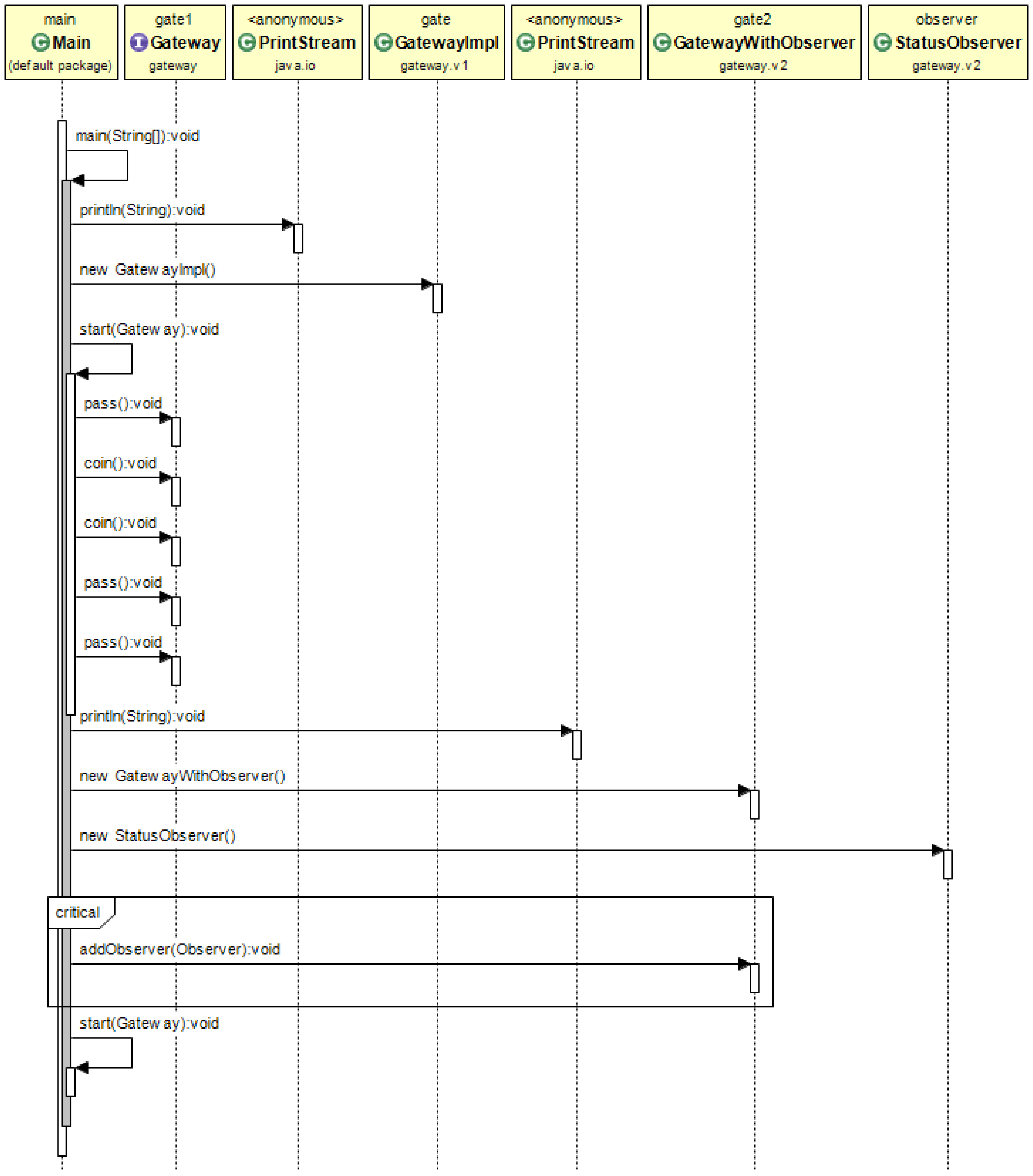


Diagram 55. Main function two levels deep. The ObjectAid UML Explorer for Eclipse sequence diagram.

4.3 Summary for reverse engineering to sequence diagram

This section contains table summarising abilities of each CASE tool for sequence diagram reverse engineering and voting table for each CASE tool pair [20]. All abilities counted as equally important.

Symbols meaning:

+ - functionality supported,

– - functionality not supported,

+/– - functionality partially supported

Table 3. Sequence diagram reverse engineering abilities of CASE tools

	Enterprise Architect	**Visual Paradigm**	**The ObjectAid UML Explorer for Eclipse**
Does not require code execution	**–**	**+**	**+**
Lifeline have named instances	**–**	**+**	**+**
Lifeline have anonymous instances	**+**	**+**	**+**
Can record calls to external libraries	**–**	**+/–** Calls to methods of extended classes only	**+**
Initialisation using constructors defined in code is shown	**+**	**+**	**+**
Initialisation using default constructors is shown	**–**	**+**	**+**
Instance creation represented as per UML definition	**–**	**+**	**–**
Messages numbering	**–**	**+/–** Broken when going to deeper levels.	**–**
Can see calls to concrete instances when method accepts interface	**+**	**–**	**–**

Table 4. Voting table for sequence diagram reverse engineering abilities of CASE tools

	Visual Paradigm	**The ObjectAid UML Explorer for Eclipse**	**Enterprise Architect**	**Votes**
Visual Paradigm	0	2	6	8
The ObjectAid UML Explorer for Eclipse	1	0	4	5
Enterprise Architect	1	1	0	2

Results in table 4 show that Visual Paradigm is a CASE tool with the best set of functions for reverse engineering of methods to sequence diagrams and its representation of sequence diagrams is closest to the standard.

5 Combined results for reverse engineering abilities

This section contains information combined from summaries from chapter 3 and 4. It provides a selection of a best CASE tool from selected CASE tool.

Class diagrams are more helpful for overall system understanding and are considered more important for software maintenance [25] than sequence diagrams. Because of that 1 point for class diagrams reverse engineering in Table 2 is equal to 2 points in Table 5.

Table 5. Voting table for combined results for reverse engineering abilities

	Enterprise Architect	**Visual Paradigm**	**The ObjectAid UML Explorer for Eclipse**	**Votes**
Enterprise Architect	0	11	5	16
Visual Paradigm	6	0	2	8
The ObjectAid UML Explorer for Eclipse	8	11	0	19

Even if Enterprise Architect was the best for class diagrams reverse engineering, and Visual paradigm was the best for sequence diagrams reverse engineering, both of this CASE tools are weak in another reverse engineering ability. It allowed The ObjectAid UML Explorer to get a best total score for both reverse engineering abilities. If the class diagrams are to be considered more important than sequence diagrams, then the ranking of these 3 CASE tools will hold.

6 Summary

This thesis does not include all possible variation of CASE tools and does not analyse all design patterns. It contains a proposed approach for evaluation of CASE tools based on their reverse engineering capabilities. As a result, this approach identifies the best CASE tool from the selection.

The goal of this bachelor thesis “Analysis and selection of best CASE tools based on reverse engineering and patterns detection for java project” is to identify which of three selected CASE tools is the best for reverse engineering of Java-based software to UML model-based documentation.

Three selected CASE tools are Enterprise Architect 13.0, Visual paradigm 15.0 Standard edition and The ObjectAid UML Explorer v1.2.2.

The analysis is based on two CASE tools reverse engineering capabilities.

The first capability is reverse engineering of Java code to class diagrams and the ability to represent GOF design patterns in generated class diagrams. The analysis was based on twelve out of twenty-three GOF design patterns.

Results in Table 1 show that Enterprise Architect is the best CASE tool for reverse engineering Java code to class diagrams. Enterprise Architect represented all GOF patterns with enough details on UML diagrams to identify patterns. The ObjectAid UML Explorer is close to Enterprise Architect, but it could not represent observer and iterator design patterns. Even the fact that it shows some patterns with more details than Enterprise Architect does not help it to be the best. Visual Paradigm is the worst of three selected CASE tools. This CASE tool had issues with factory method, observer, iterator and command design pattern representations.

Another CASE tools capability is reverse engineering of Java code to a sequence diagram.

For reverse engineering of methods to sequence diagrams, Enterprise Architect is the worst choice. It is hard to setup because it requires executing code and setup recording markers. This CASE tool cannot record calls to external libraries, like java core libraries and calls to default constructors.

Visual Paradigm and The ObjectAid UML Explorer are both good in the generation of sequence diagrams using reverse engineering. However, Visual Paradigm is better in the representation of lifelines for newly created instances, and it has numbering for messages. It allows calling Visual Paradigm the best for sequence diagrams for reverse engineering.

By the combined results of the analysis given in Table 5, it can be concluded that The ObjectAid UML Explorer is the best from selected CASE tools for documentation of java code using reverse engineering because it showed good results for both analysed capabilities. It was almost as good as Enterprise Architect in the representation of GOF design patterns and better than Enterprise Architect in reverse engineering of sequence diagrams. Visual Paradigm was a bit better than The ObjectAid UML Explorer in sequence diagrams reverse engineering, but Visual Paradigm was worse than it in class diagram reverse engineering.

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software,” 1995.

[2] “Enterprise Architect,” Sparx Systems Pty Ltd., [Online]. Available: http://sparxsystems.com/products/ea/.

[3] “The Object Management Group,” [Online]. Available: https://www.omg.org.

[4] “GangOfFour,” [Online]. Available: http://wiki.c2.com/?GangOfFour.

[5] T. Veskioja, Stable marriage problem and college admission, 2005.

[6] E. J. Chikofsky and J. Cross, “Reverse engineering and design recovery: a taxonomy,” *IEEE Software,* vol. 7, no. 1, pp. 13-17, 1990.

[7] L. C. Briand, “Software documentation: how much is enough?,” in *Seventh European Conference on Software Maintenance and Reengineering*, 2003.

[8] “Visual paradigm,” [Online]. Available: https://www.visual-paradigm.com/.

[9] “The ObjectAid UML Explorer for Eclipse,” [Online]. Available: http://www.objectaid.com/home.

[10] J. Vassiljeva, “All project with reverse engineered UML diagrams,” [Online]. Available: https://bitbucket.org/Camio/loputoo/src/master/.

[11] “TestNG testing framework source code,” [Online]. Available: https://github.com/cbeust/testng/tree/6.14.3.

[12] “Design patterns implemented in Java,” [Online]. Available: https://github.com/iluwatar/java-design-patterns.

[13] “Design patterns implemented in Java, Singleton,” [Online]. Available: https://github.com/iluwatar/java-design-patterns/tree/master/singleton.

[14] “Design patterns implemented in Java, Adapter,” [Online]. Available: https://github.com/iluwatar/java-design-patterns/tree/master/adapter.

[15] “Design patterns implemented in Java, Command,” [Online]. Available: https://github.com/iluwatar/java-design-patterns/tree/master/command.

[16] A. Torim, “IDU1550 Tarkvara arhitektuur ja disain,” [Online]. Available: https://moodle.hitsa.ee/enrol/index.php?id=14074.

[17] J. Vassiljeva, “Gateway java application code,” [Online]. Available: https://bitbucket.org/Camio/loputoo/src/master/gateway/.

[18] J. Vassiljeva, “Sierpienski implementation, source code,” [Online]. Available: https://bitbucket.org/Camio/loputoo/src/master/sierpinski/.

[19] Wikipedia, “Sierpinski triangle,” [Online]. Available: https://en.wikipedia.org/wiki/Sierpinski_triangle.

[20] L. Võhandu, K. Raidma and T. Veskioja, “Subjektiivsetest hinnangutest objektiivsete tulemusteni,” [Online]. Available: http://maurus.ttu.ee/ained/IDN5120/doc/11/IDN5120_Loeng1_2013_2sept.ppt.

[21] C. Larman, “15.7 Basic Sequence Diagram Notation,” in *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 2 ed., 2004.

[22] “Source code for java.util.Observable,” [Online]. Available: http://book2s.com/java/src/package/java/util/observable.html.

[23] “Standard Edition 8 API Specification, java.util.Observable,” Java™ Platform, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Observable.html.

[24] “The Java™ Tutorials. Synchronized Methods,” Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html.

[25] S. Cozzetti B. de Souza, N. Anquetil and K. M. de Oliveira, “A study of the documentation essential to software maintenance,” in *23rd annual international conference on Design of communication: documenting & designing for pervasive information,* Coventry, United Kingdom, 2005.

[26] “Enterprise Architect - Use Patterns,” [Online]. Available: http://www.sparxsystems.com/resources/developers/use_uml_patterns.html.

[27] P. M. I. P. Ghulam Rasool, “Pages Evaluation of design pattern recovery tools,” *Procedia Computer Science,* vol. 3, pp. 813-819, 2011.

[28] Z. Marco, “Data mining techniques for design pattern detection,” 2012.

[29] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., 2002.

[30] Osman, Hafeez and Chaudron, Michel, “Correctness and Completeness of CASE Tools in Reverse Engineering Source Code into UML Model,” *The GSTF Journal on Computing (JoC),* 2012.

[31] D. Cutting and J. Noppen, “An extensible benchmark and tooling for comparing reverse engineering approaches.,” *International Journal on Advances in Software,* vol. 8, pp. 115-124, 2015.

[32] B. Bellay and H. Gall, “A comparison of four reverse engineering tools,” in *Proceedings of the Fourth Working Conference on Reverse Engineering,* Amsterdam, 1997.

[33] N. Shi and R. A. Olsson, “Reverse Engineering of Design Patterns from Java Source Code,” *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06),* pp. 123-134, 2006.

[34] “Programming Tutorials by SourceTricks,” [Online]. Available: http://www.sourcetricks.com/2013/04/composite-pattern.html.