Analysis of GoF Design Patterns used in Knowledge Processing Systems

STEFAN NADSCHLÄGER AND JOSEF KÜNG, Institute for Application Oriented Knowledge Processing (FAW), Johannes Kepler University, Linz

To increase the quality of knowledge processing systems and provide help to software developers, selected existing knowledge processing systems are analysed for the occurrence of used object-oriented design patterns (especially from the Gang-of-Four catalogue). This analysis intends to draw attention to the lack of good software design in the area of knowledge processing systems and at the same time provides a smaller catalogue of design patterns with proven usage in practice, to support development. The design patterns were identified manually in a structured analysis by reverse engineering the source code, supported by a design pattern detection tool. As a result, Gang-of-Four design patterns, suitable for developing custom knowledge processing systems, are presented and discussed.

CCS Concepts: • **Software and its engineering → Patterns**; *Software architectures*; • **Computing methodologies →** Knowledge representation and reasoning;

Additional Key Words and Phrases: Design Patterns, Knowledge Processing Systems, Software Architecture

ACM Reference Format:
STEFAN NADSCHLÄGER and JOSEF KÜNG. 2017. Analysis of GoF Design Patterns used in Knowledge Processing Systems. 1, 1 (October 2017), 22 pages. https://doi.org/10.1145/3147704.3147711

1 INTRODUCTION

This paper intends to contribute to an architectural guideline for knowledge processing systems (KPS). The authors started to create such a guideline [11] with the intention to improve the quality and document best practices of knowledge processing system implementations. In contrast to their general design pattern collection, this paper provides a list of applied object-oriented design patterns in existing knowledge processing systems. Even though they will not suffice to develop one's own custom knowledge processing system, reverse engineering existing systems helps to identify best practices and learn about their architecture.

As the name already implies, knowledge processing systems process *knowledge*. Knowledge, in contrast to *data* and *information*, is represented containing some relationship between data/information: For example, IF-THEN constructs, or IS-A hierarchies, as well as modern and more complex formats like *ontologies*. This knowledge has to be stored and processed to get new knowledge, or insights (a process called *reasoning* or *inference*). Examples of types of knowledge processing systems are *Expert Systems*, *CASE-based Reasoning systems*, *intelligent tutoring systems*, etc.

The area of knowledge processing systems is large. In order to curtail it, this paper primarily handles *expert systems* (ES) and *rule engines* (RE), particularly the systems JavaDON (Tomić et al. [14]), OpenRules (OpenRules

Author's address: STEFAN NADSCHLÄGER and JOSEF KÜNG, Institute for Application Oriented Knowledge Processing (FAW), Johannes Kepler University, Linz.

XXXX-XXXX/2017/10-ART
https://doi.org/10.1145/3147704.3147711

Website [13]), Drools (Drools Website [19]) and NxBRE (Dossot [6]) will be analyzed. They all allow to use the source code and deal with the same kind of knowledge processing system. Also, the number of design patterns is high; therefore, the analysis will be restricted to the "Gang-of-Four" patterns (GoF[1], Gamma et al. [9]). Another reason to focus on these patterns is that this book is commonly considered as the first and most significant one (even though the concept of design patterns existed before). Therefore, the object-oriented patterns described in the book are famous and well-known amongst developers. This paper has a strong focus on expert systems / rule engines, but the identified patterns can also be used in other types of knowledge processing systems.

Besides a structured manual analysis via reverse engineering the source code, a tool was used to support the identification process. Tools were developed to identify Gang-of-Four [9] design patterns in Java: pattern4 [17] FINDER [18], or DPFinder [5] to name just a few in scientific context. Unfortunately only a few of them are available for free usage and are still executable within modern system environments. Therefore, they are just used as a basic support.

In this paper the term "quality" focuses on the aspects of clean code: *Complicatedness* (not to be confused with *complexity*) should be reduced, *maintainability* should be increased and the *readability* should be improved.

The analysis of used design patterns in knowledge processing systems supports the development of custom knowledge processing systems and intends to increase the quality of their implementation. Although the usage of design patterns does not automatically mean a better quality (see Khomh and Gueheneuce [10], or Zhang and Budgen [20]), studies exist (for example Alghamdi and Qureshi [2]) that show the positive influence of documented best practices on system maintainability. This list is intended as a basic catalogue of design patterns for software engineers and researchers who have to develop custom knowledge processing systems. Moreover, this paper intends to draw attention to the lack of design concepts for knowledge processing systems. It is analysed if design patterns have been used at all in knowledge processing systems and if yes, which ones. The selection of the patterns is based on the criteria list defined in section 2.

The result of the analysis and hence its contribution is a narrower list of object-oriented design patterns that have already successfully been used in knowledge processing systems. The patterns provide small reusable building blocks, in contrast to a reference architecture. This guarantees the software developer enough freedom in the overall system design. Moreover, it gives researchers, not that familiar with design patterns, the chance to implement evolutionary prototypes in higher quality.

Section 2 shows an overview on existing knowledge processing system frameworks, section 3 describes the methodology of analysing the systems. In section 4 the identified design patterns are presented, section 5 discusses the result of the analysis and section 6 concludes the paper.

2 KNOWLEDGE PROCESSING SYSTEMS

In this section, a general overview on existing knowledge processing systems and their architecture will be provided.

2.1 Knowledge Processing System Architecture

The common representation of a knowledge processing system architecture is presented in figure 1. It only shows the three main components that are described in literature in a high-level overview (e.g., Akerkar and Sajja [1], or Beierle and Kern-Isberner [4]).

[1]A name given to the authors of the book: Gamma, Helm, Johnson and Vlissides.

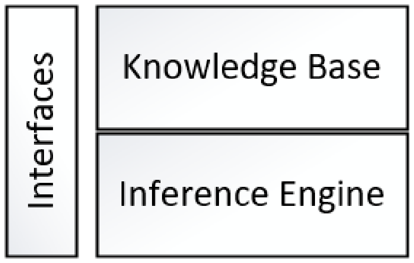


Fig. 1. A common representation of a knowledge processing system architecture.

Figure 1 shows a major problem in knowledge processing system design. This figure only contains three major components of the systems but is generally accepted as the architecture description.

Interfaces These can either be graphical, or remote communication interfaces (for example, web services) to access the functionality of a knowledge processing system.

Knowledge Base The knowledge base contains all the knowledge (in form of *facts* and *rules*) of the system.

Inference Engine The inference engine encapsulates the actual knowledge processing algorithm.

Working Memory The working memory is commonly not shown in figures but is an essential component. It functions as a runtime storage for inferred facts. This component is seen as part of the *inference engine* and normally not shown in the architecture overview, like the one in figure 1.

Explanation Module Optionally, some knowledge processing systems also provide an explanation module. This module is responsible for creating human readable text that explains, how certain knowledge was deduced. This component is also normally not shown in the architecture overview.

2.2 Existing Frameworks

Table 1 shows a selection of existing knowledge processing system frameworks that are found when one searches for a so-called *expert system shell* (this can be compared to a knowledge processing *framework*). The table lists whether the frameworks are currently actively developed (first column), the source code is available (second column), if they apply to the JSR-94 specification (third column, see the Java rule engine specification [15]) and in which programming language they are implemented (fourth column). JSR-94 defines the architecture of a rule engine implemented in Java and therefore relates to the architecture of expert systems. Nevertheless, it cannot be applied to the development of knowledge processing systems in general. It is strictly tied to the Java programming language and only handles the maintenance and evaluation of rules.

The *usage* column provides information on the popularity of the frameworks. It is a subjective estimation based on the following criteria:

- Date of last change to the source code: An old date indicates lost interest in the framework.
- Entries on http://www.openhub.net, a platform that scans open-source frameworks and provides statistical information. Also the community activity is shown.

The values in columns *openhub* and *last commit* are both taken from the openhub platform.

Table 1. Common knowledge processing system frameworks.

	active	source available	jsr-94	language	usage	openhub	last commit
JavaDON		✓		java	no known active usage	no entry	4 years ago
NxBRE	✓	✓		c#	mainly research	425K LOC	2 years ago
JESS	✓		✓	java	active usage	no entry	?
CLIPS	✓	✓		c	active usage	no entry	?
JBoss Drools	✓	✓	✓	java	active usage	1.63M LOC	1 day ago
OpenRules	✓	✓	✓	java	active usage	no analysis	?
ILog JRules	✓		✓	java	active usage	no entry	?
Pyke		✓		python	no known active usage	21.7K LOC	8 years ago
OpenExpert		✓		php	no known active usage	68.9K LOC	8 years ago
DTRules		✓		java	no known active usage	77.9K LOC	5 years ago
Infosapient				java	no known active usage	no entry	?
JEOps				java	no known active usage	no entry	?
JLisa		✓	✓	java	no known active usage	46K LOC	13 years ago
Mandarax		✓		java	no known active usage	36.8K LOC	10 years ago
OpenCyc	✓			java	active usage	80.7K LOC	?

Even though many such systems exist, a lot of them have been discontinued. Moreover, it is not always possible to adapt a framework to one's needs. This makes the development of custom knowledge processing systems necessary.

The majority of systems listed in table 1 is implemented in an object-oriented language (mainly Java). Therefore, an analysis of applied object-oriented design patterns documents suitable patterns for their implementation. Even though the Gang-of-Four patterns are considered application domain independent, some of them may be more appropriate to be used in the core components of a knowledge processing system than others. This paper intends to find these patterns, narrowing the number of patterns in the catalogue to a few relevant and therefore making the selection of suitable patterns easier for a software engineer.

3 METHODOLOGY

Besides a manual analysis, a Java tool ("Design pattern detection tool, *pattern4*") was used to support the identification of design patterns. It is described in Tsantalis et al. [17] and is freely available [16]. It is implemented in the Java programming language and detects Gang-of-Four design patterns in Java code via similarity scoring between graph vertices. The tool provides a graphical user interface and needs a directory containing the bytecode files of a Java application. It scans these files and provides a list of found design patterns together with the classes that participate in the pattern. Because such a tool can never be 100% correct, the personal estimation of the authors is favoured over results from the analysis tool.

The following list shows the **criteria for selecting a pattern**:

- The patterns are from the Gang-of-Four catalogue.
- Patterns are relevant for the implementation of core knowledge processing components.
- Patterns can commonly be used for custom knowledge processing system implementation and they are not specific to solve a system specific problem in one framework. Only generally applicable patterns will be considered. Suitable design patterns may vary for different inference algorithms and this analysis focuses on the whole system architecture.

- The patterns are either identified via manual selection or with the support of the tool.

The **steps for manual analysis** are described in the following paragraph. It is a precondition that the analyst is familiar with the Gang-of-Four patterns.

(1) Download and set up the source code of the framework in an IDE of your choice.
(2) Run the tool to get an overview of possible interesting locations in the source code.
(3) Identify the core components of a knowledge processing system (as described in section 2.1): Knowledge representation in the form of *rules* and *facts*, a *knowledge base*, a *working memory* and an *inference engine*.
(4) Start searching from the classes that are located in the identified components and that were suggested by the analysis tool.
(5) Identify parts of these components that deviate from simple object-oriented programming standards (e.g., simple inheritance, information hiding mechanisms, etc.). Select those that seem to work together in a more advanced structure.
(6) Check, if these structures match an existing Gang-of-Four pattern.

For the analysis, the knowledge processing systems JavaDON, Drools, OpenRules and NxBRE were selected, because they implement the same kind of knowledge processing system and don't rely on external components in other programming languages (as, for example, JLisa [3] relies on a Lisp runtime). Moreover, their source code is available.

In another publication by the authors [12], a design pattern language for knowledge processing systems is presented. The basis for the identification of patterns were also the frameworks JavaDON, Drools and NxBRE. These three systems were selected because they use an object-oriented programming language and the two Java implementations differ in the application of JSR-94. As the Java example applying JSR-94, JBoss Drools was chosen because it is the most mature and widely used framework. As an alternative to a Java implementation NxBRE was chosen. JavaDON was selected in favor of DTRules or Mandarax because the active development of this framework wasn't stopped that long ago.

In the course of pattern identification the central classes and components of the systems were identified. The starting point for the manual analysis are the following classes, grouped by five core components (see Nadschläger and Küng [12]). Also referenced classes were considered for the analysis:

(1) *Fact*:
    - *Drools*: `Fact` and `Field`
    - *JavaDON*: `Facts`, `KnowledgeChunk`, `Attribute`, `KnowledgeElement`, `FramE`, `Slot` and `Domain`
    - *NxBRE*: `Fact`, `Atom` and `IPredicate`
(2) *Rule*:
    - *Drools*: `Rule`, `Condition` and `Consequence`
    - *JavaDON*: `Rule`, `Clause`, `KnowledgeElement` and `KnowledgeChunk`
    - *NxBRE*: `Implication`, `Query`, `AtomGroup` and `Atom`
(3) *Knowledge Base*:
    - *Drools*: `KieBase`, `KnowledgeBase`, `InternalKnowledgeBase`, `KnowledgeBaseImpl` and `ReteooBuilder`
    - *JavaDON*: `KnowledgeBase`, `DomainKnowledge` and `Knowledge`
    - *NxBRE*: `FactBase` and `ImplicationBase`
(4) *Working Memory*:
    - *Drools*: `StatefulKnowledgeSessionImpl`, `InternalAgenda`, `ReteAgenda`, `PropagationList`, `RetePropagationList` and `ReteWorkingMemory`
    - *JavaDON*: `WorkingMemory`, `WMElement` and `KnowledgeChunk`
    - *NxBRE*: `AbstractWorkingMemory`, `WorkingMemory` and `FactBase`
(5) *Inference Engine*:

- *Drools*: `StatelessKnowledgeSessionImpl`
- *JavaDON*: `InferenceEngine`
- *NxBRE*: `IEImpl`, `IWorkingMemory` and `ImplicationBase`

In the following section, the identified design patterns and how they are used to support the implementation of a custom knowledge processing system, will be presented. This list only contains patterns relevant for the implementation of the core components presented before. Used design patterns for detailed inference algorithms, or detail implementations of knowledge storage, will not be handled, as these would exceed the boundaries of this paper. Moreover, they would not add any additional benefit in the implementation of a generic knowledge processing system because the architecture described in [12] assumes that these are single encapsulated components, not influencing the whole system architecture.

As an additional result, the analysis intends to show the lack of a common software architecture (and the associated inconsistent implementation) in existing knowledge processing system frameworks. The authors are highly convinced that a consistent architecture should also lead to a similar design pattern usage.

4 DESIGN PATTERNS ANALYSIS

In this section the identified patterns will be presented and their occurrence in the single systems will be described via selected examples.

The design pattern detection tool could not be used to verify the results in NxBRE as it only can scan Java source code.

The tool did not identify any design patterns in JavaDON, where all classes in the `don` package were scanned (including subpackages). It may be due to the reverse engineering that certain data structures, relevant for the identification, got lost.

In OpenRules the following packages were scanned:

- `com.openrules.ruleengine`: 2 patterns found, but they did not relate to the core components.
- `com.openrules.java`: 0 patterns found.
- `com.openrules.rules`: 31 patterns found, only 2 patterns could be verified to take part in core components.
- The other packages did not contain source code directly contributing to core components.

In Drools the `drools-core` package was scanned. 558 design patterns in total were identified by the pattern analysis tool. Only 11 of them could be mapped to central classes of a knowledge processing system.

For better understanding of the patterns, their UML diagrams can be found in the appendix A.

4.1 Factory Method

"**Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. Gamma et al. [9]"

In Drools, the `Agenda` is a central component. It plays an important role in the *working memory* component of a knowledge processing system. Nevertheless, its implementation can vary depending on the knowledge representation and the inference algorithm (e.g., Rete algorithm by Forgy [8], or Phreak [7]). This pattern is shown in figure 2.

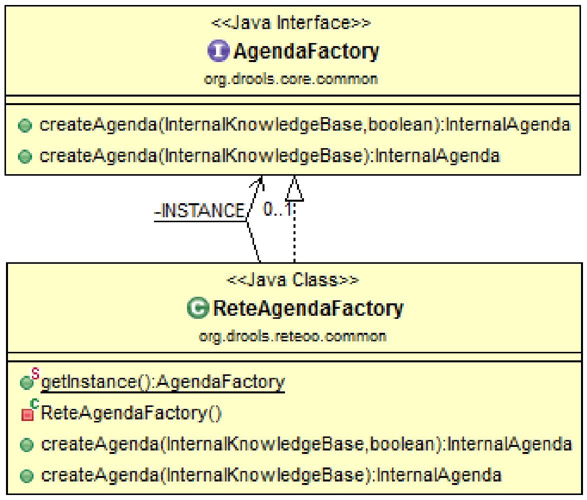

<<Java Interface>>
AgendaFactory
org.drools.core.common
createAgenda(InternalKnowledgeBase,boolean):InternalAgenda
createAgenda(InternalKnowledgeBase):InternalAgenda
-INSTANCE
0..1
<<Java Class>>
ReteAgendaFactory
org.drools.reteoo.common
getInstance():AgendaFactory
ReteAgendaFactory()
createAgenda(InternalKnowledgeBase,boolean):InternalAgenda
createAgenda(InternalKnowledgeBase):InternalAgenda

Fig. 2. Example for Factory Method in Drools.

Other occurrences of Factory Method in Drools are `AgendaGroupFactory`, but also the `WorkingMemoryFactory` that creates the *working memory* itself.

In JavaDON no Factory Method could be found, it was never intended to support multiple inference algorithms. Also in OpenRules and NxBRE no reasonable usage of the Factory Method could be identified.

4.2 Singleton

"**Intent:** Ensure a class only has one instance, and provide a global point of access to it. Gamma et al. [9]"

In Drools, also the Singleton pattern is used in the `AgendaFactory`. Only one factory for the whole application is needed. This reduces the amount of (unnecessary) instances. Other occurrences in Drools are, for example, `ConflictResolver` that is responsible for defining the correct rule evaluation order. Another example, where the Singleton pattern is used, is `RuleActivationListenerFactory` that creates listeners to be notified, whenever a rule is activated. Unfortunately, the implementation of the last one is not a correct Singleton, as it keeps the constructor with public visibility and therefore allows the creation of multiple instances.

No usage of the Singleton pattern could be found in JavaDON, OpenRules or NxBRE.

4.3 Adapter

"**Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Gamma et al. [9]"

In Drools, this pattern is used, for example, in the *knowledge base*. It can use different variants of a `SessionCache` for caching existing sessions. In order to hide this cache from the users of the *knowledge base*, the Adapter pattern is used, where `KnowledgeBaseImpl` functions as the adapter and `SessionCache` is the adaptee (see figure 3).

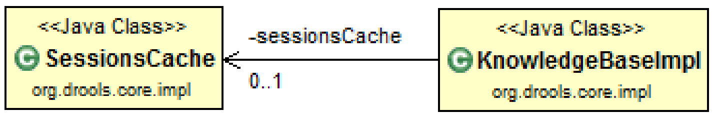


Fig. 3. Example for ADAPTER in Drools.

In NxBRE the rules can either be defined in XML, in the code, or via Microsoft Visio. Therefore, `IRuleBaseAdapter` exists, to be able to handle them all simultaneously.

In JavaDON and OpenRules, no ADAPTER could be identified.

4.4 DECORATOR

"**Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Gamma et al. [9]"

In Drools, the DECORATOR can be found in the Agenda. `ScheduledAgendaItem` is a decorator for `AgendaItem` (an entry in the agenda). This pattern is presented in figure 4.

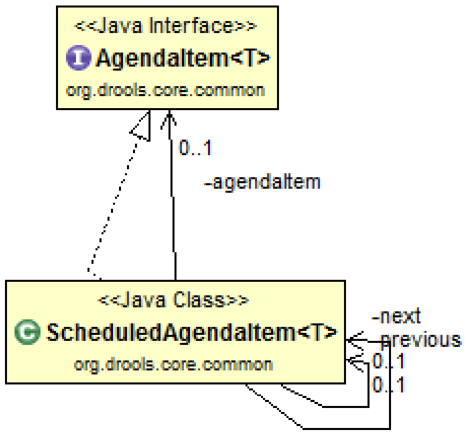


Fig. 4. An example for a DECORATOR in Drools.

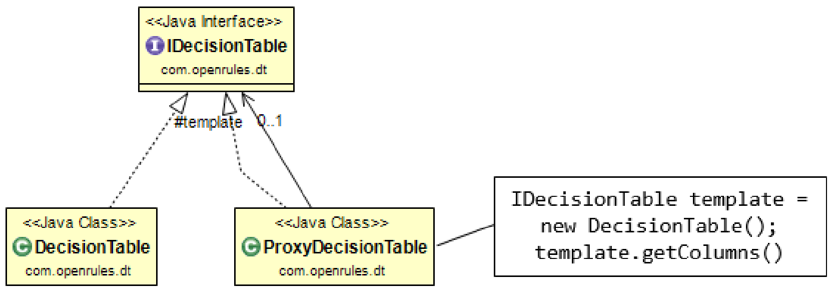
The DECORATOR pattern also supports extensibility of the knowledge processing system. Often, object composition should be favoured over inheritance. In case of the identified example it is a good way to create facts with additional functionality (e.g., a scheduled `AgendaItem`).

This pattern is not used in JavaDON, NxBRE and OpenRules.

4.5 PROXY

"**Intent:** Provide a surrogate or placeholder for another object to control access to it. Gamma et al. [9]"

In OpenRules, rules can come from different sources, but mainly they are extracted from Microsoft Excel sheets. `ProxyDecisionTable` is a *rule* construct that allows the usage, hiding the concrete source of the rules (see figure 5).



```
IDecisionTable template =
  new DecisionTable();
 template.getColumns()
```

Fig. 5. Example for Proxy in OpenRules.

4.6 Template Method

"**Intent:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. Gamma et al. [9]"

In Drools, an example for the usage of the Template Method pattern can be found in `WorkingMemoryLogger`. It defines the basic logic of the logger, but leaves the implementation of the `logEventCreated` method to the user.

In NxBRE the Template Method is used for the *working memory* itself. An `AbstractWorkingMemory` defines the basic methods for a concrete *working memory* implementation.

No usage of this pattern could be identified in JavaDON and OpenRules.

4.7 Strategy

"**Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Gamma et al. [9]"

An example for the Strategy pattern in Drools is the `DefaultAgenda` class having a reference to an implementation of `InternalWorkingMemory`. As already mentioned before, the implementation of the working memory can differ, depending on the inference algorithm in use (see figure 6).

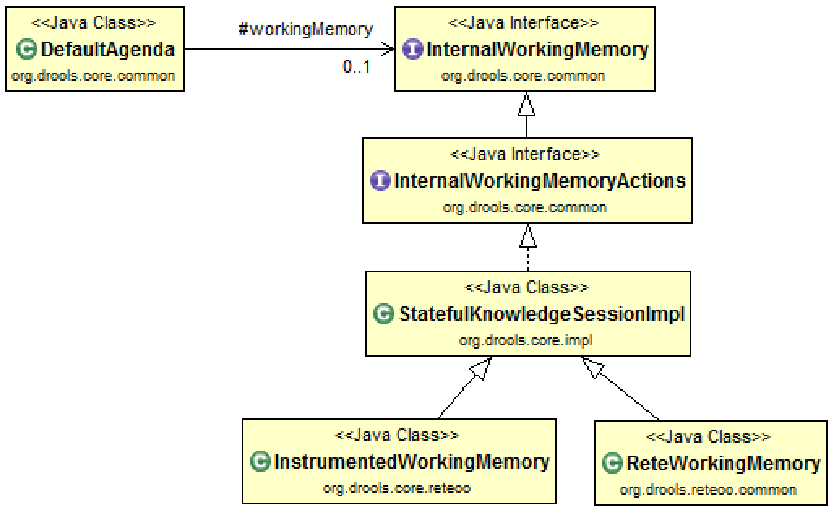


Fig. 6. An example for the STRATEGY pattern in Drools.

An interesting application of the STRATEGY pattern can be found in OpenRules. There, it is used to define a *rule*. The class `DecisionTable` uses instances of `IDecisionRow` to model the IF and THEN parts of the *rule*. Depending on the concrete implementation, `DecisionRow` can either be a condition, consequence, or an action (see figure 7).

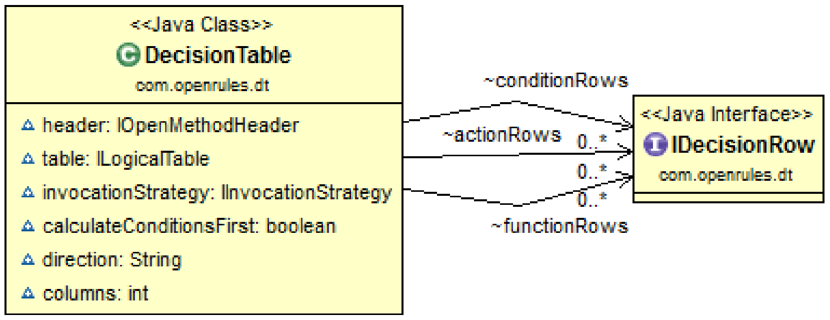


Fig. 7. An example for STRATEGY in OpenRules.

In order to allow different evaluation methods for `DecisionTable`, in OpenRules an `IInvocationStrategy` is defined (see figure 8).

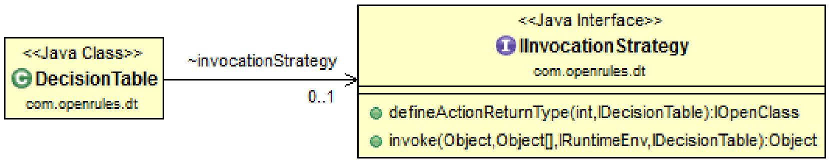


Fig. 8. An example for STRATEGY in OpenRules.

In JavaDON, the STRATEGY pattern can be found in `Formula`. These appear in rules and allow the integration of complex mathematical functions into rule evaluation. The same structure can also be found in `Relation` (see figure 9).

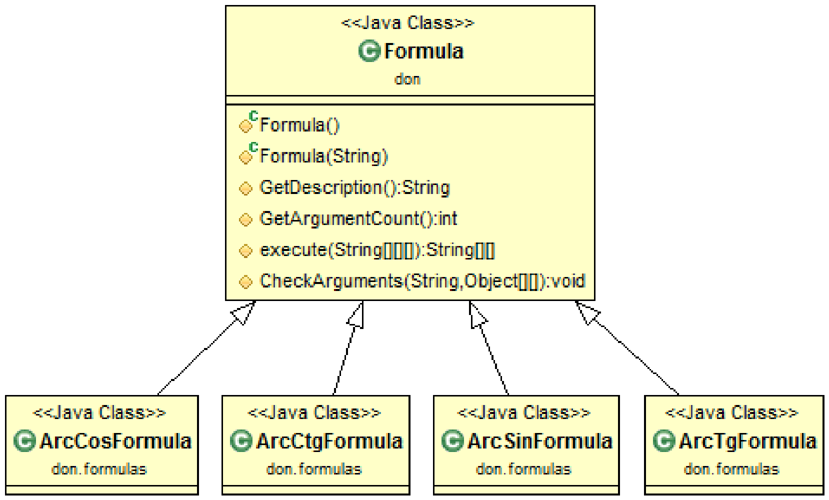


Fig. 9. An example of STRATEGY in JavaDON.

No relevant usage of the STRATEGY pattern could be found in NxBRE.

4.8 OBSERVER

"**Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically. Gamma et al. [9]"

One central component in Drools that uses the OBSERVER pattern is `WorkingMemoryLogger`, which is part of the *explanation module*. It is responsible for logging ("explaining") events that happen in the *working memory*. Therefore, `WorkingMemory` functions as *Subject* and `WorkingMemoryLogger` as *Observer*. Events that will be logged are, for example, manipulation of rules, or when matches in the Agenda happen (see figure 10).

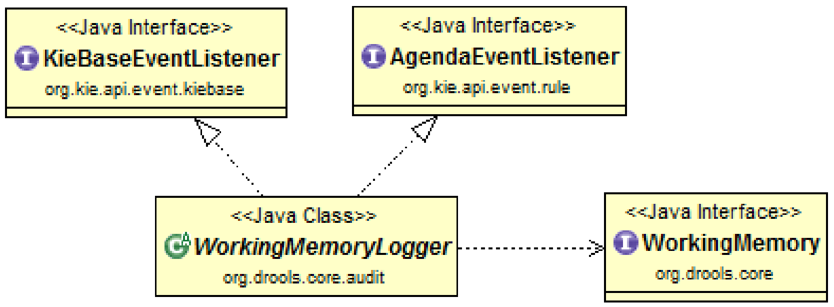


Fig. 10. Example for the OBSERVER pattern in Drools.

In NxBRE, the OBSERVER pattern is used for notifications when *facts* are added, modified, or removed from the *working memory*. This is done using the `NewFactEvent` delegate.

This pattern could not be found in JavaDON and OpenRules.

5 DISCUSSION

The main design patterns that were applied in knowledge processing systems are shown in table 2.

Table 2. An overview of the identified design patterns.

	Factory Method	Singleton	Observer	Adapter	Decorator	Template Method	Proxy	Strategy
Drools	✓	✓	✓	✓	✓	✓		✓
JavaDON								✓
OpenRules							✓	✓
NxBRE			✓	✓		✓		

Not many of the 23 Gang-of-Four design patterns are used in these systems, and there are also big differences in the implementation. This decreases the possibility to implement custom knowledge processing systems following the same high quality implementation standard. Also, the knowledge of successful software architectures is lost. Moreover, not many design patterns are used in object oriented knowledge processing system implementations to establish a high-quality implementation.

This list and the sparse matrix of used patterns may be a hint of a lack of awareness of qualitative software architecture in knowledge processing systems. The absence of design patterns alone does not indicate this (Khomh and Gueheneuce [10], or Zhang and Budgen [20]). The usage of design patterns is a way to improve the quality on the premise that they are used carefully.

Figure 11 shows an overview of identified design patterns and locations where they can be applied. The highlighted cells are patterns identified only manually.

After carefully studying the remaining Gang-of-Four design patterns and thinking about their possible usage in a knowledge processing system, the following could be identified to extend the list:

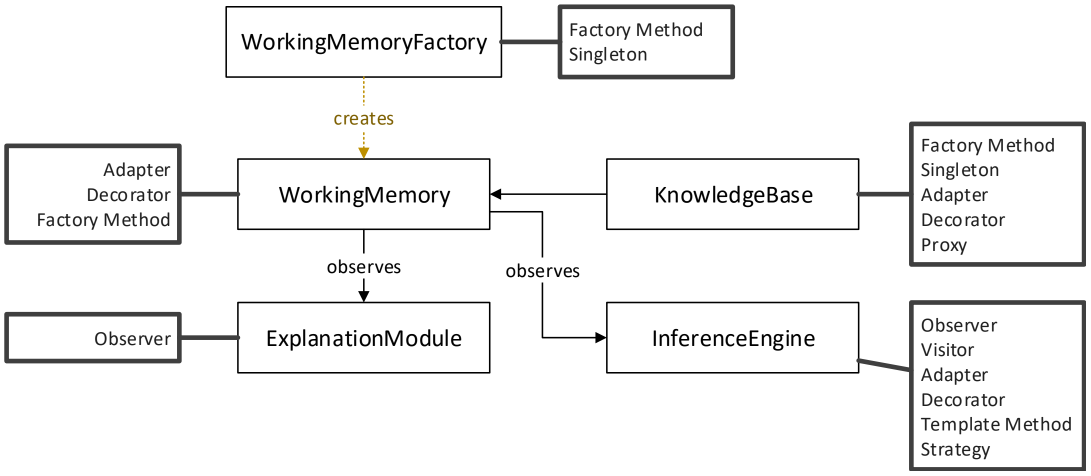


Fig. 11. Overview of design pattern usage.

5.1 Abstract Factory

"**Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes. Gamma et al. [9]"

Abstract Factory supports the creation of different inference engines with various rule/fact representations. This allows to create an extensible framework to integrate the ideal structure for every kind of knowledge.

5.2 Chain of Responsibility

"**Intent:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. Gamma et al. [9]"

Chain of Responsibility is a suitable pattern to avoid explicit triggering of rules in inference engines. Facts / Rules can pass on the call to other facts / rules.

5.3 Command

"**Intent:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. Gamma et al. [9]"

The Command pattern can be used for triggering of new rules, in contrast to Chain of Responsibility. Examples for the usage of the Command pattern are assertion or retraction to the working memory.

5.4 Iterator

"**Intent:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Gamma et al. [9]"

Iterator can be applied for iterating over collections of facts /rules. Therefore, hiding the concrete representation of the rules and facts storage.

5.5 Visitor

"**Intent:** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. Gamma et al. [9]"

Visitor can be used to represent different "commands" (functionality, not to confuse with the Command pattern) in the inference algorithm (e.g., checking if a fact exists in the knowledge base).

Figure 12 shows an overview containing the additional design patterns discussed in this section.

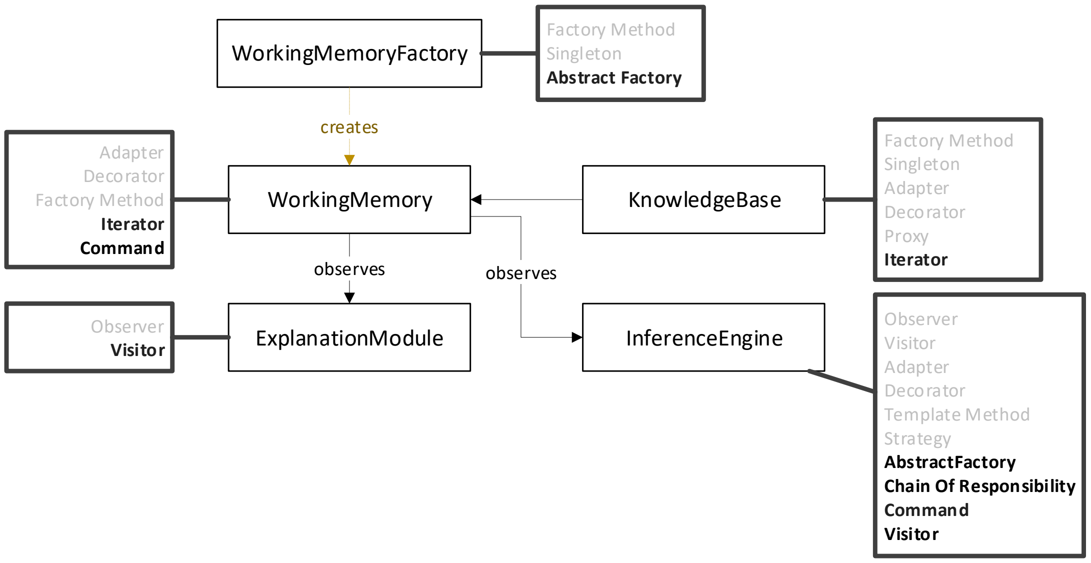


Fig. 12. Overview of design pattern usage including additional patterns.

6 SUMMARY AND FUTURE WORK

In this paper, the analysis of existing knowledge processing systems on usage of Gang-of-Four design patterns and where to apply them was presented. The identified catalogue of design patterns does not claim to be complete but provides a good overview of patterns that can be used in the implementation of custom knowledge processing systems. These extend the already existing pattern collection for knowledge processing system development and it was also shown, where and how they can be applied to develop custom knowledge processing systems. According to the hypothesis that a consistent architecture should lead to a similar set of applied design patterns, the analysis revealed a lack of quality awareness of knowledge processing software architectures.

In future work, additional systems can be analysed for usage of object-oriented design patterns. Also, additional components of a knowledge processing system (e.g., an extended explanation module) have to be analysed.

Moreover, the patterns presented in this paper are on a high level, providing a good overview, but the work has to be extended for analysing patterns used in more concrete components (for example, inference algorithm).

7 ACKNOWLEDGEMENTS

The authors would like to thank their shepherd Nistala Venkata Padmalata for the really helpful feedback that led to a much-improved version of this paper.

REFERENCES

[1] Rajendra Akerkar and Priti Sajja. 2010. *Knowledge-based systems.* Jones & Bartlett Publishers.

[2] Fatimah Mohammed Alghamdi and M. Rizwan Jameel Qureshi. 2014. Impact of Design Patterns on Software Maintainability. 6 (2014), 41–46. Issue 10.

[3] Mike Beedle. 2017. JLisa: A Clips-like Rule engine accessible from Java with the full power of Common Lisp. http://jlisa.sourceforge.net/. (01 2017).

[4] Christoph Beierle and Gabriele Kern-Isberner. 2006. *Methoden wissensbasierter Systeme.* Springer.

[5] Mario L. Bernardi, Marta Cimitile, and Giuseppe Di Lucca. 2013. Design Pattern Finder : A Model-driven Graph-Matching Approach to Design Pattern Mining. (2013).

[6] David Dossot. 2016. NxBRE .NET Business Rule Engine. (11 2016). https://github.com/ddossot/NxBRE/wiki

[7] Drools. 2017. Hybrid Reasoning: Phreak algorithm. https://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html_single/index.html#PHREAK. (01 2017).

[8] Charles L Forgy. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence* 19, 1 (1982), 17–37.

[9] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1994. *Design Patterns. Elements of Reusable Object-Oriented Software.* Prentice Hall.

[10] Foutse Khomh and Yann-Gael Gueheneuce. 2008. Do Design Patterns Impact Software Quality Positively?. In *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 274–278.

[11] Stefan Nadschläger and Josef Küng. 2016. A Pattern Collection for Knowledge Processing System Architecture. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*.

[12] Stefan Nadschläger and Josef Küng. 2017. A Pattern Language for Knowledge Processing Systems. In *accepted for publication in proceedings of VikingPLoP 2017, Grube, Schleswig-Hohlstein.*

[13] Inc. OpenRules. 2017. OpenRules Business Rules - Time to Excel. http://openrules.com/. (01 2017).

[14] Bojan Tomić, Jelena Jovanović, and Vladan Devedžić. 2006. JavaDON: an open-source expert system shell. *Expert Systems with Applications* 31, 3 (2006), 595 – 606. http://www.sciencedirect.com/science/article/pii/S0957417405002721

[15] Alex Toussaint. 2003. Java Rule Engine API JSR-94. (2003).

[16] Nikolaos Tsantalis. 2017. Design Pattern Analysis Tool. https://users.encs.concordia.ca/ nikolaos/pattern_detection.html. (2017).

[17] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. 2006. Design pattern detection using similarity scoring. *IEEE transactions on software engineering* 32, 11 (2006), 896–909.

[18] Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong (Eds.). 2015. *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC '15.* ACM Press.

[19] Drools Website. 2016. Drools Website. (2016). https://www.drools.org/

[20] Cheng Zhang and David Budgen. 2012. What Do We Know about the Effectiveness of Software Design Patterns? 38 (2012), 1213–1231. Issue 5.

APPENDIX A - GANG-OF-FOUR PATTERNS

In this appendix an overview of the Gang-of-Four design patterns, used in the paper, is given.

Factory Method

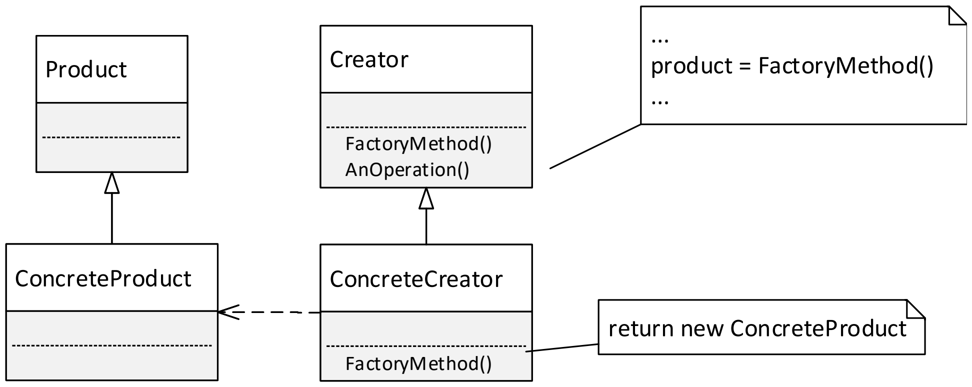


Fig. 13. The Factory Method as it is shown in Gamma et al. [9].

Singleton

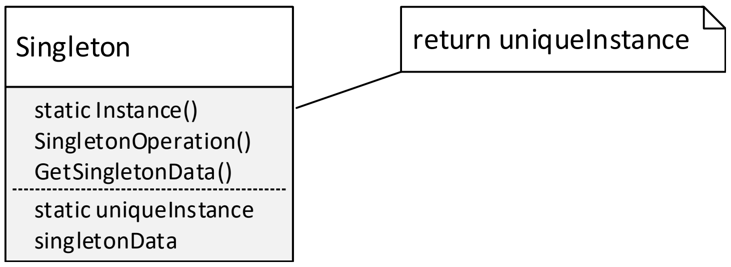


Fig. 14. The Singleton as it is shown in Gamma et al. [9].

Adapter

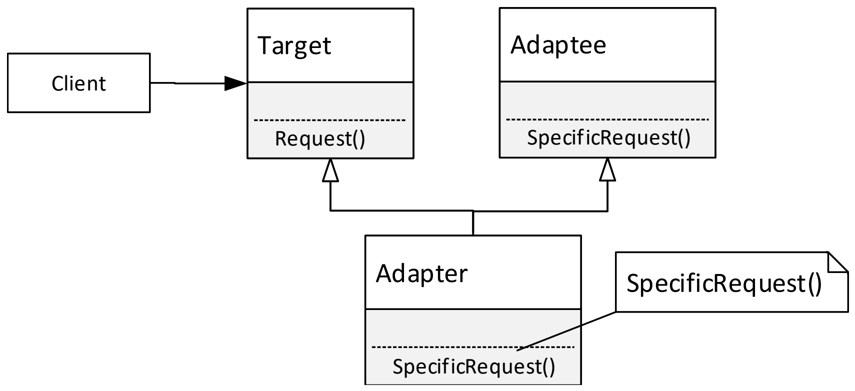


Fig. 15. The Adapter as it is shown in Gamma et al. [9].

Decorator

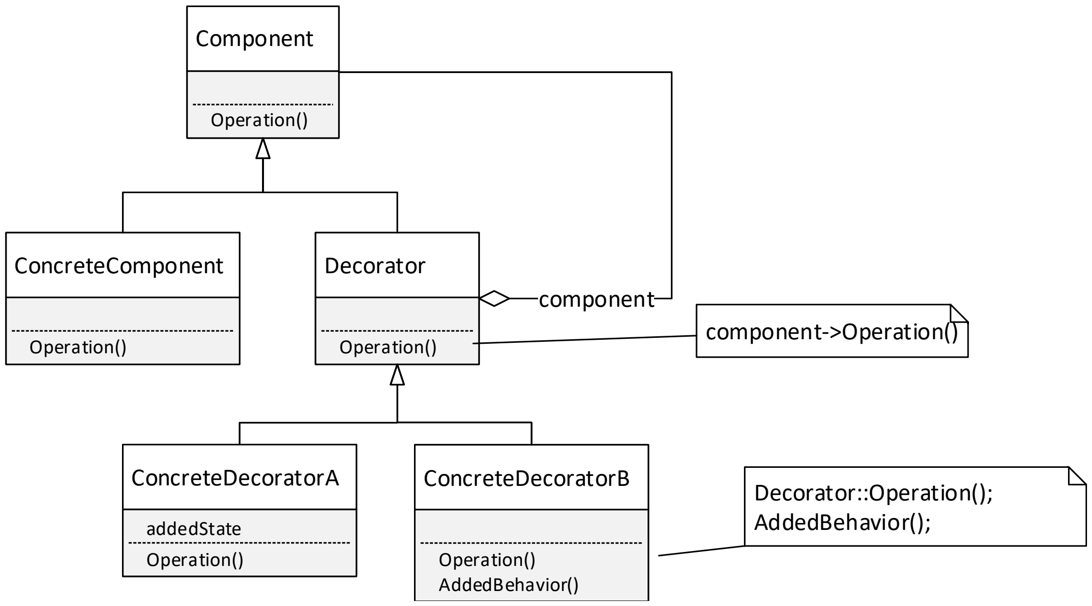


Fig. 16. The Decorator as it is shown in Gamma et a. [9].

Proxy

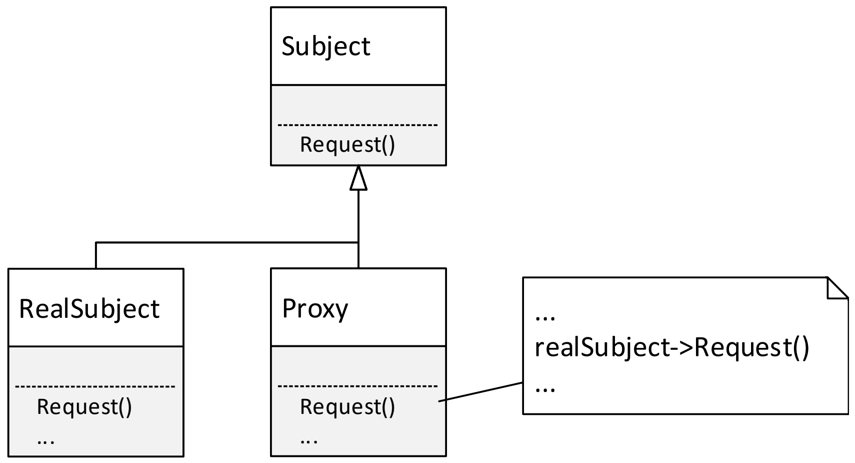


Fig. 17. The Proxy as it is shown in Gamma et al. [9].

Template Method

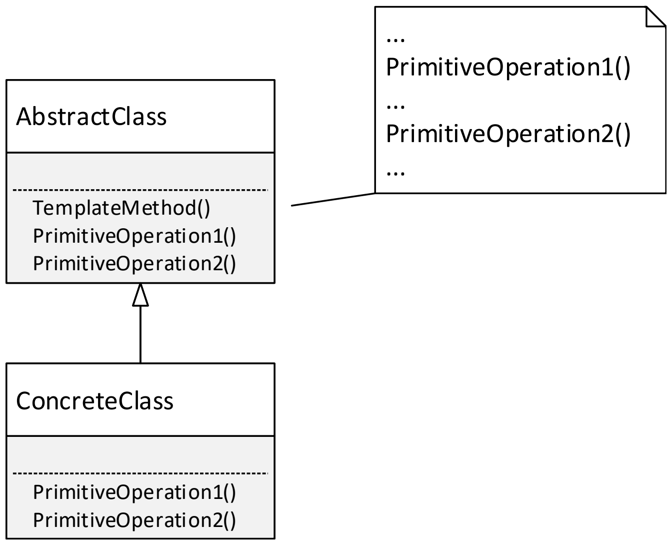


Fig. 18. The Template Method as it is shown in Gamma et al. [9].

Strategy

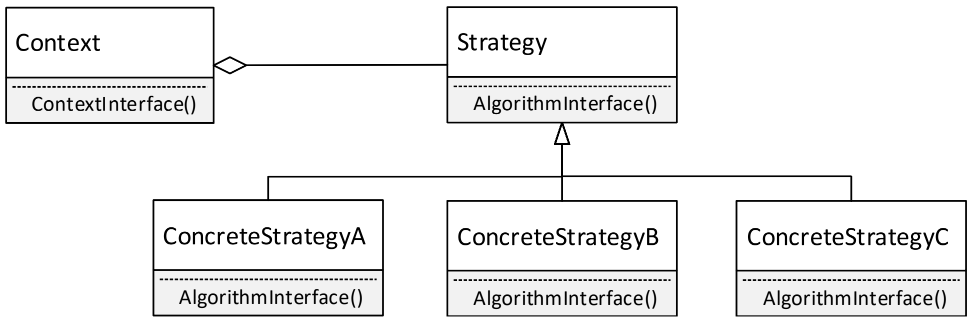


Fig. 19. The Strategy as it is shown in Gamma et al. [9].

Observer

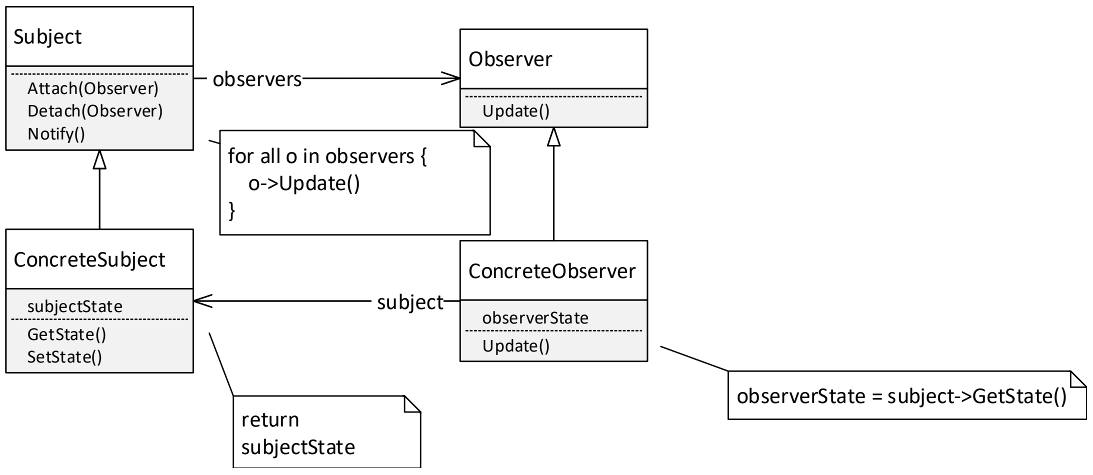


Fig. 20. The Observer as it is shown in Gamma et al. [9].

Abstract Factory

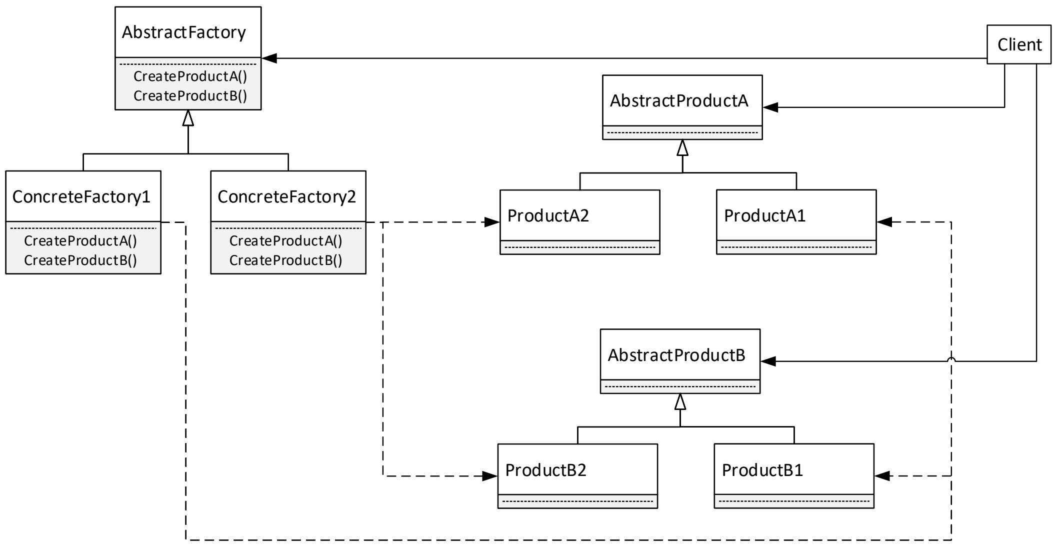


Fig. 21. The Abstract Factory as it is shown in Gamma et al. [9].

Chain of Responsibility

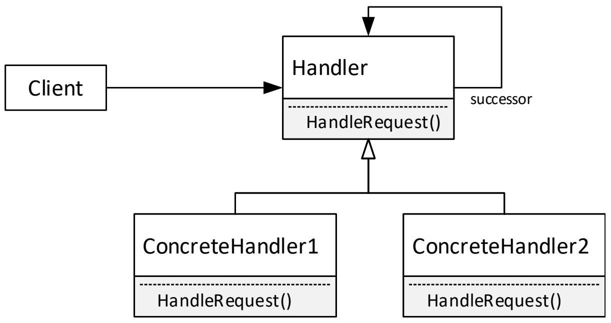


Fig. 22. The Chain of Responsibility as it is shown in Gamma et al. [9].

Command

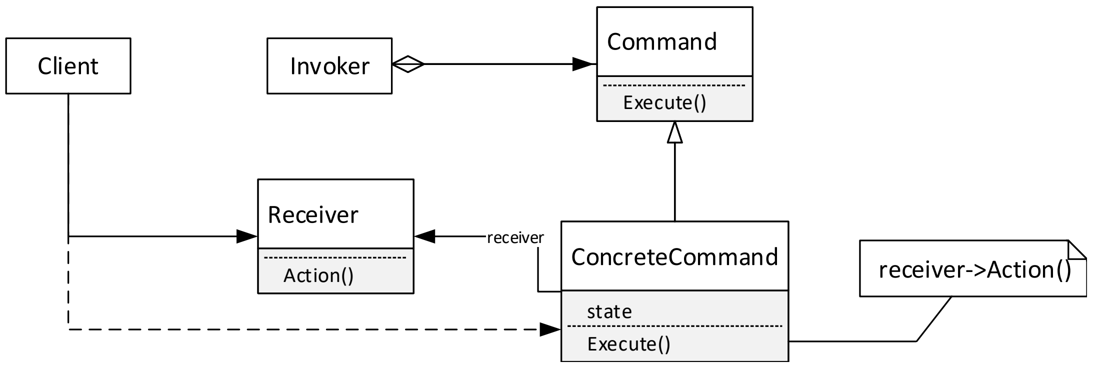


Fig. 23. The Command as it is shown in Gamma et al. [9].

Iterator

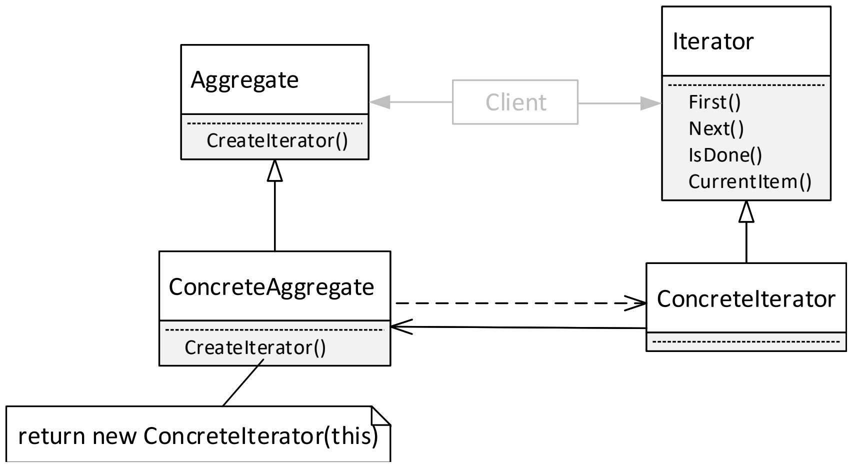


Fig. 24. The Iterator as it is shown in Gamma et al. [9].

Visitor

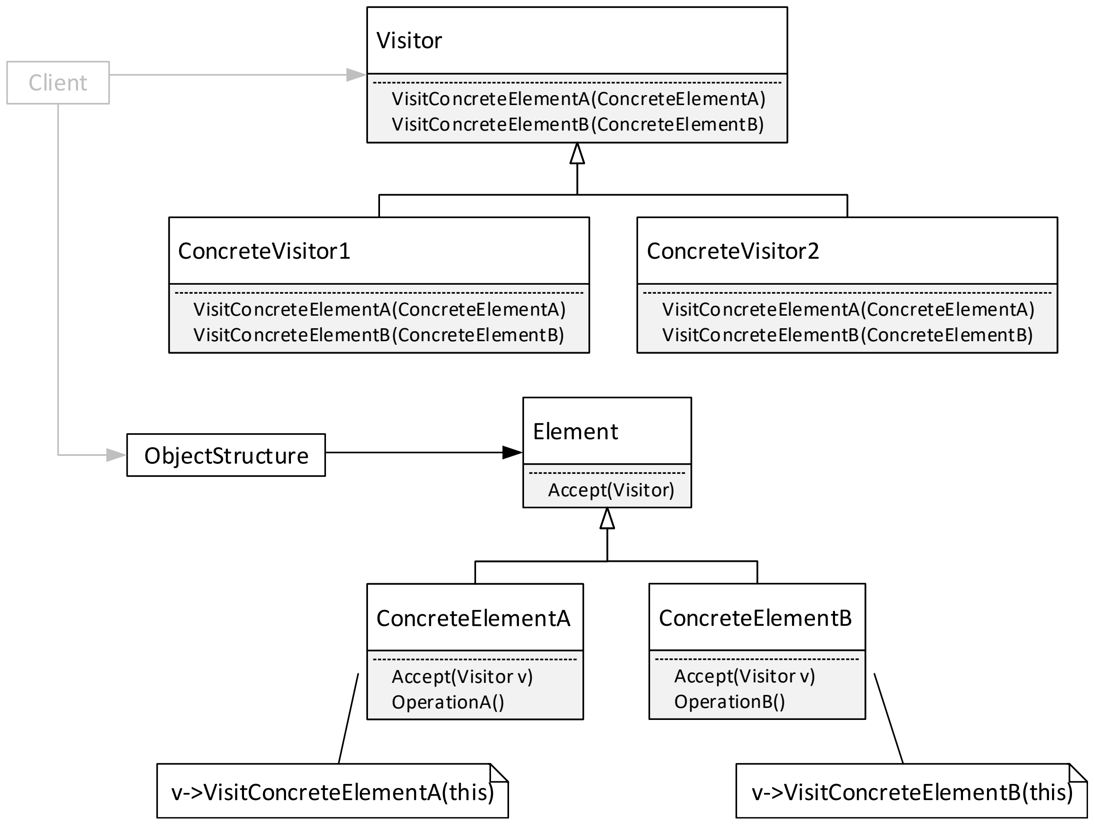

Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
Client
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Fig. 25. The Visitor as it is shown in Gamma et al. [9].

Received February 2017; revised ; accepted