УДК 004.4'22
DOI: 10.15827/0236-235X.127.389-397
Дата подачи статьи: 28.01.19
2019. Т. 32. № 3. С. 389–397

Проектирование интерпретатора языка QVT Operational Mappings для программного средства UML Refactoring в рамках модельно-ориентированного подхода

О.А. Дерюгина [1], *к.т.н., преподаватель, o.a.derugina@yandex.ru*
Е.В. Крючкова [1], *бакалавр*

[1] *МИРЭА – Российский технологический университет, г. Москва, 119454, Россия*

В работе рассмотрена концепция модельно-ориентированного подхода MDA для решения задач автоматизации разработки ПО. Подход предполагает разделение процесса разработки на три основных шага: разработка платформонезависимой модели PIM, создание платформозависимой модели PSM, разработка кода ПО. Подробно рассмотрены стандарты MDA: XMI (XML Metadata Interchange), унифицирующий обмен моделями между программными средствами, и QVT (Query/View/Transformation), описывающий языки запросов к моделям.

Цель работы – проектирование интерпретатора языка QVT Operational Mappings, одного из семейства языков QVT, для программного средства UML Refactoring. Программное средство UML Refactoring предназначено для анализа и трансформации UML-диаграмм классов, описывающих объектно-ориентированную архитектуру ПО.

В процессе анализа рассчитываются объектно-ориентированные метрики (Avg. DIT, Avg. NOC, Avg. CBO и др.), а также выполняется поиск трансформаций «Введение интерфейса», «Стратегия», «Фасад», снижающих значение целевой функции рефакторинга, выбранной пользователем. На основе информации о языке создания запросов к моделям QVTo для системы UML Refactoring спроектирован класс QVTInterpreter.java, который интерпретирует QVT-запрос к диаграмме классов, а затем преобразует его в последовательность трансформаций, таких как добавление класса, добавление атрибута к классу, добавление метода к классу, добавление интерфейса, добавление метода к интерфейсу, добавление пакета, добавление класса в пакет, добавление интерфейса в пакет, добавление пакета в пакет.

Для каждой трансформации спроектирован отдельный класс-наследник класса Refactoring.java, в ходе трансформации передаваемый на вход классу Transformator.java, который, в свою очередь, вызывает метод execute() каждой трансформации.

Ключевые слова: *MDA, модельно-ориентированный подход, проектирование программного обеспечения, архитектура программного обеспечения, UML Refactoring, UML, QVT.*

В рамках модельно-ориентированного подхода к разработке программных систем MDA, призванного повысить эффективность труда разработчиков, консорциумом OMG предложен стандарт QVT (Query/View/Transformation), описывающий языки задания запросов к UML-моделям, наиболее часто применяемым при проектировании объектно-ориентированной архитектуры сложных программных систем [1–3].

Методология MDA включает в себя набор стандартов для создания инструментальных средств модельно-ориентированной разработки: MOF – стандарт, описывающий основную мета-модель подхода MDA [4]; UML – стандарт, описывающий унифицированный язык моделирования, предназначенный для проектирования моделей сложных систем [5]; XMI – стандарт, описывающий XML-подобный формат передачи моделей между инструментальными средствами [6]; OCL – язык задания требований к моделям, на основе которых может производиться верификация [7]; QVT – стандарт, документирующий три языка написания запросов к моделям: QVTc, QVTr, QVTo [3].

Для проведения анализа UML-моделей (верификация, сравнение, оценка качества), их трансформации и доказательства свойств необходим способ формального описания UML-моделей.

В настоящее время разработаны различные подходы к описанию UML-диаграмм классов (табл. 1).

Одни исследователи стремятся к унифицированному подходу для описания UML-модели в целом [1, 8], другие предлагают частные решения для конкретных диаграмм [9–11].

Преимуществом унифицированного подхода является возможность обработки UML-модели целиком, а специфицированного – воз-

Таблица 1

Основные подходы к описанию UML-диаграмм классов

Table 1

The main approaches to the UML class diagram description

Подход	Достоинства	Недостатки
Графический [8]	Наглядность	Сложность машинной обработки
Основанный на специализированных языках моделирования [3, 9]	Стандартизированность. Наличие средств автоматизированной валидации	Для XMI-формата: вычислительная сложность поиска в документе – $O(n)$
Основанный на алгебре логики [10, 11]	Высокая степень формализации – меньшая вероятность разночтений в понимании UML-диаграммы. Возможность проверки свойств UML-модели	Сложность машинной обработки
Основанный на теории графов [12, 13]	Удобство машинной обработки	Наличие в диаграммах классов различных типов ребер и узлов, усложняющих обработку. Неопределенность, чем являются пакеты в терминах теории графов

Таблица 2

Основные подходы к трансформации UML-диаграмм классов

Table 2

The main approaches to the UML class diagram transformation

Подход	Достоинства	Недостатки
Детерминированный подход	Возможность производить вычисления более эффективно	Необходимость описывать трансформации на языке программирования
Специализированные языки трансформаций [3, 14, 15]	Возможность добавления пользователем трансформаций без использования алгоритмического языка программирования. Стандартизированный подход	Сложность понимания трансформаций
Графовые трансформации [11]	Наличие развитого математического аппарата	

можность выбора наиболее оптимального способа представления для каждой диаграммы в отдельности (например, для диаграмм состояний подходят конечные автоматы, сети Петри или темпоральная логика).

Существуют различные подходы к трансформации UML-диаграмм классов (табл. 2).

Рефакторинг – это реструктуризация системы, сохраняющая ее поведение.

Распространенным является подход, при котором в уже написанный работоспособный код программы вносятся изменения, направленные на повышение читаемости кода, его гибкости, облегчение возможностей сопровождения и т.д.

Существует множество работ, в которых излагаются общие принципы, рекомендации и правила по улучшению качества кода ПО: так, в [8, 14, 15] содержатся подробные описания паттернов (шаблонов) проектирования объектно-ориентированного ПО, рецепты рефакторинга (указания конкретных ситуаций и рекомендуемых для применения паттернов), принципы написания хорошо структурированного кода и проектирования объектно-ориентированной архитектуры.

Одной из сложностей автоматизации процесса рефакторинга является то, что часто предлагаемые авторами решения основаны на субъективных экспертных оценках, которые порой трудно обосновать математически.

Некоторые из методов, предложенных для рефакторинга кода, могут быть использованы и при рефакторинге архитектуры ПО.

В области рефакторинга UML-моделей ПО существуют три подхода: автоматический, автоматизированный и рефакторинг вручную (в данной работе он не рассматривается).

Автоматический рефакторинг

На вход алгоритма рефакторинга подаются UML-диаграмма *d*, целевая функция *f*(*d*), число итераций алгоритма *n*, невязка ε, множество трансформаций *T*.

На выходе алгоритм возвращает UML-диаграмму классов *d'*, значение целевой функции *f*(*d*) для которой локально или глобально минимальное/максимальное.

Автоматический рефакторинг UML-диаграмм классов связан с поисковой программной инженерией (SBSE – search based software engineering) [16]. В SBSE задачи программной инженерии формулируются как задачи оптимизации, которые затем решаются поисковыми алгоритмами (генетическим алгоритмом, симуляцией отжига, алгоритмами роевого интеллекта и т.п.). SBSE используется для решения задачи рефакторинга UML-диаграмм классов в работах [17–19].

Методы поисковой инженерии используют трансформацию UML-диаграмм при помощи эволюционных алгоритмов для таких задач, как улучшение повторного использования существующих архитектур ПО через паттерны проектирования [17, 19], построение иерархической декомпозиции для программной системы [20], проектирование структуры классов [21].

Недостатком подхода является то, что алгоритм не учитывает семантику трансформаций.

Автоматизированный рефакторинг

На вход алгоритма рефакторинга подаются UML-диаграмма *d*, целевая функция *f*(*d*), множество трансформаций *T*. На выходе алгоритм возвращает список рекомендованных к применению трансформаций *T'*, использование которых снижает/увеличивает значение целевой функции *f*(*d*).

Недостаток подхода в том, что применение одной из предложенных трансформаций может исключить возможность применения некоторых других, то есть конечный результат может быть неоптимальным с точки зрения целевой функции.

Принятие конечного решения осуществляется разработчиком, у которого, помимо самой диаграммы классов, есть информация о назначении каждого класса, интерфейса и т.д.

Такой подход связан с разработкой инструментальных средств автоматизированного рефакторинга, в которых ведущая роль отдается проектировщику системы. Одним из этих инструментальных средств является UML Refactoring [22–24], разработанное для анализа UML-диаграмм классов (расчета объектно-ориентированных метрик, поиска трансформаций, снижающих значение целевой функции рефакторинга) и трансформации.

В данной статье рассматривается задача проектирования интерпретатора языка запросов к UML-моделям QVT Operational Mappings (далее – QVTo), описанного в стандарте QVT для программного средства UML Refactoring. Этот язык написания запросов призван по аналогии с языком SQL упростить редактирование UML-моделей пользователем средств проектирования.

Язык QVTo позволяет пользователю системы проектирования редактировать систему путем написания запросов, что может быть эффективнее редактирования модели при помощи средств визуального проектирования в случае, если количество элементов системы велико.

Стандарт языка написания запросов к моделям QVT

Данный стандарт включает в себя три языка: QVT Core Language (QVTc), QVT Relations Language (QVTr) и QVT Operational Mappings Language (QVTo) [3].

Декларативные языки QVTc и QVTr способны описывать двунаправленные преобразования. Спецификация определяет их конкретный текстовый и абстрактный синтаксис. Кроме того, QVTr имеет графический синтаксис.

QVTo – императивный язык, описывающий только однонаправленные преобразования. Является расширением QVTr и QVTc. Основная идея QVTo заключается в том, что объектные шаблоны, указанные в отношениях, создаются с помощью императивных конструкций. Таким образом, декларативно заданные отношения реализуются императивно. Синтаксис оперативных отображений языка обеспечивает конструкции, часто встречающиеся в императивных языках (циклы, условия и др.).

Язык QVTc реализован в коммерческом дополнении к OptimalJ. Язык QVTr реализован в IKV++ medini QVT, Tata Consultancy ModelMorf, MOMENT-QVT и Eclipse M2M Relations2ATLVM. Язык QVTo реализован в SmartQVT и Eclipse M2M.

Трансформация на языке QVTo может быть описана следующим образом:

```
transformation MMaToMMb(in Ma: MMa,
out Mb: MMb);
```

Данная запись объявляет трансформацию MMaToMMb, на входе принимающую модель Ma, метамоделью которой является MMa, и отображает ее на модель Mb, метамоделью которой является MMb.

Для того чтобы объявить тип метамодели, требуется описать ее при помощи директивы modeltype:

```
modeltype ECORE uses 'http://www.
eclipse.org/emf/2002/Ecore';
```

Входной точкой трансформации является функция main().

Операция отображения описывается следующим образом:

```
mapping T1::T1toT2() : T2;
```

T1 – тип данных на входе отображения; T2 – тип данных, получаемых в результате отображения.

```
t.map T1toT2(); // пример вызова
операции отображения
```

БНФ для языка QVTo может быть описана следующим образом:

```
QVTo = "modeltype ECORE uses EMetaMo-
del;" "transformation" name_transformation
"(" in-out_model ");" "main() {" new_Object
! Map";"…Map "}" </ Mapping…Mapping />
new_Object = "object EObject {" Object
"};"
Object =  name_EObject ! eOb-
jects";"…eObjects "};" ! Map";"…Map
Map = in_model_name ".rootObjects()
[" EObject "]->" name_map "()"
Mapping =  "mapping" EObject "::
" name_map "():" EObject "{" Object "}"
eObjects = "eClassifiers += object
ecore::EClass {" inf_eObjects "}" !
"eOperations += object ecore::EOpera-
tion {" inf_eObjects "}"!
"eAttributes += object ecore::EAttrib-
ute {" inf_eObjects "}"
EObject = "EPackage" ! "EClass" !
"EOperation" ! "EAttribute"
in-out_model = </ "in" in_model_name ":
ECORE,"/> "out" out_model_name ": ECORE"
inf_eObjects = name_eObjects ! id_eOb-
jects ! type(return)_eObjects
name_transformation = c…c
name_EObject = "name := '" c…c "' ;"
name_map = c…c
in_model_name = c…c
out_model_name = c…c
name_eObjects = "name := '" c…c "' ;"
type(return)_eObjects = "type := '" c…c
"' ;"
id_eObjects = "id := '" ц…ц "' ;"
c = 6!ц
б = "A"!"B"!"C"!...!"Z"
ц = "0"!"1"!...!"9"
```

Интерпретатор языка QVT Operational Mappings программного средства UML Refactoring

Программное средство UML Refactoring предназначено для анализа UML-диаграмм классов с целью поиска рефакторингов, которые сделают архитектуру UML-диаграммы классов более гибкой, снизят сложность сопровождаемости проектируемой системы [22–24].

Программное средство рассчитывает для UML-диаграммы классов основные объектно-ориентированные метрики (Avg.DIT, Avg. NOC, Avg.CBO и т.д.). Затем на основе выбранного пользователем критерия качества и соответствующей ему целевой функции пользователю предлагается список трансформаций, которые улучшат показатели UML-диаграммы классов по выбранному критерию.

Пользователь может применить к диаграмме классов трансформации «Стратегия», «Фасад» и «Введение интерфейса».

Официальный сайт проекта: www.uml-refactoring.ru.

На рисунке 1 представлены основные функциональные блоки программного средства UML Refactoring.

Процесс работы с программной системой может быть описан следующим образом.

Пользователь экспортирует UML-диаграмму классов в формат XMI, затем в главном меню системы UML Refactoring выбирает команду Import и XMI-файл. Этот файл обрабатывается классом TranslatorFromXMI.java, и на его основе наполняется абстрактная структура данных UML Map [22].

Кроме того, пользователь в начале работы может выбрать в меню системы команду Open и открыть файл .umlr, являющийся встроенным расширением файлов для UML Refactoring, основанным на JSON. Затем .umlr-файл обрабатывается классом TranslatorFromJSON.java и на его основе наполняется абстрактная структура данных UML Map.

После этого происходят расчет метрик и поиск трансформаций, снижающих значение целевой функции рефакторинга. Если пользователь выбрал одну из предложенных трансформаций, являющихся наследниками класса Refactoring, то класс Transformator.java вызывает его метод execute(). В результате происходит трансформация UML-диаграммы классов.

Для поддержки языка запросов к UML-диаграммам классов QVT был спроектирован класс QVTInterpreter.java, который интерпре-

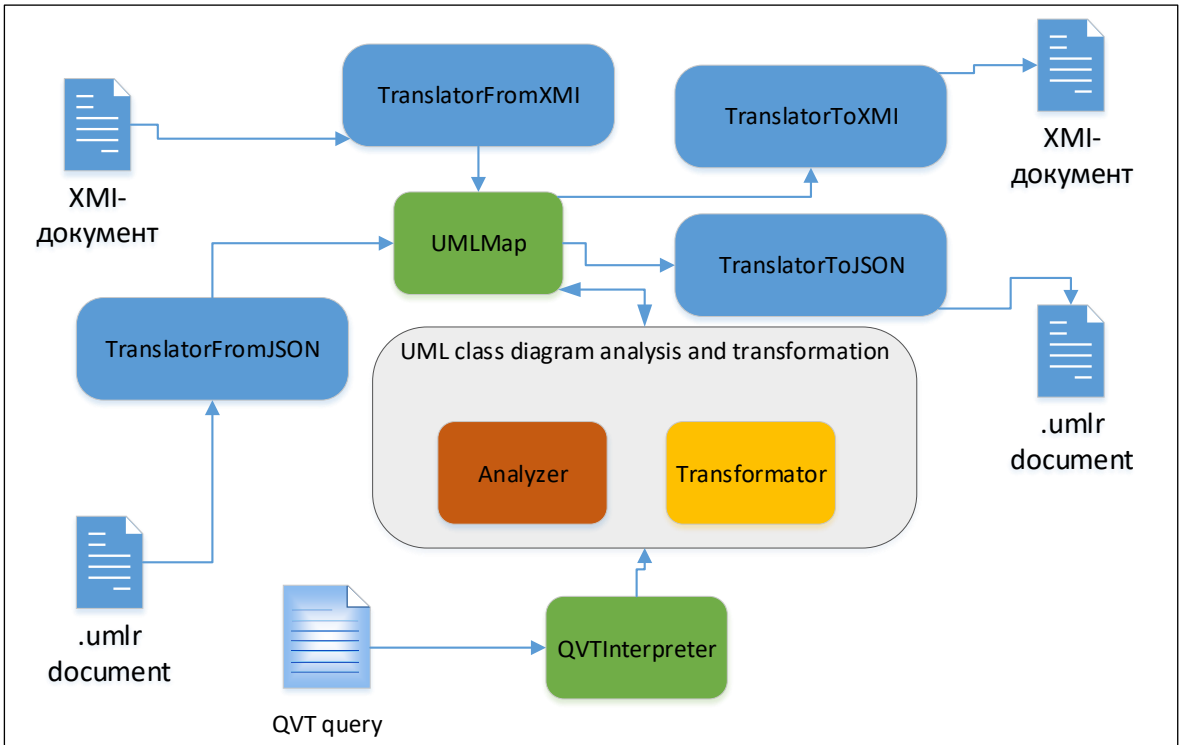

Рис. 1. Основные функциональные блоки программного средства UML Refactoring

Fig. 1. Main functional blocks of UML Refactoring

тирует QVT-запрос к диаграмме классов, а затем преобразует его в последовательность трансформаций, таких как добавление класса, добавление атрибута к классу, добавление метода к классу, добавление интерфейса, добавление метода к интерфейсу, добавление пакета, добавление класса в пакет, добавление интерфейса в пакет, добавление пакета в пакет.

Результат анализа инструментальным средством UML Refactoring диаграммы классов библиотеки агентно-реляционного отображения ArPlatform представлен на рисунке (см. http://www.swsys.ru/uploaded/image/2019-3/2019-3-dop/35.jpg).

QVT Interpreter разрабатывается с целью поддержки инструментальным средством UML Refactoring языка запросов QVTo. Он расширит возможности UML Refactoring, позволив пользователю не только выбирать предложенные программой трансформации, но и создавать свои сценарии для преобразования моделей.

На рисунке 2 показано взаимодействие классов системы UML Refactoring с новым классом QVTInterpreter.java. Для каждой трансформации спроектирован отдельный класс-наследник класса Refactoring.java. Он в ходе трансформации передается на вход классу Transformator.java, который, в свою очередь, вызывает метод execute() каждой трансформации. Задача класса QVTInterpreter.java состоит в анализе поступившего на языке QVT Operational Mappings запроса и преобразовании его в цепочку команд-наследников класса Refactoring.java, которые по очереди будут поданы на вход методу refactor() класса Transformator.

Требования, предъявляемые к разработке QVT Interpreter, заключаются в поддержке следующих операций: создание пакетов с классами, имеющими атрибуты и методы, изменение существующих пакетов, удаление модели.

Окно ввода запросов на языке QVTo в системе UML Refactoring представлено на рисунке (см. http://www.swsys.ru/uploaded/image/2019-3/2019-3-dop/36.jpg).

Заключение

В данной работе рассмотрена концепция модельно-ориентированного подхода MDA. Подробно описаны стандарты MDA – XMI и QVT.

На основе информации о языке создания запросов к моделям QVTo для поддержки данного языка запросов к UML-диаграммам клас-

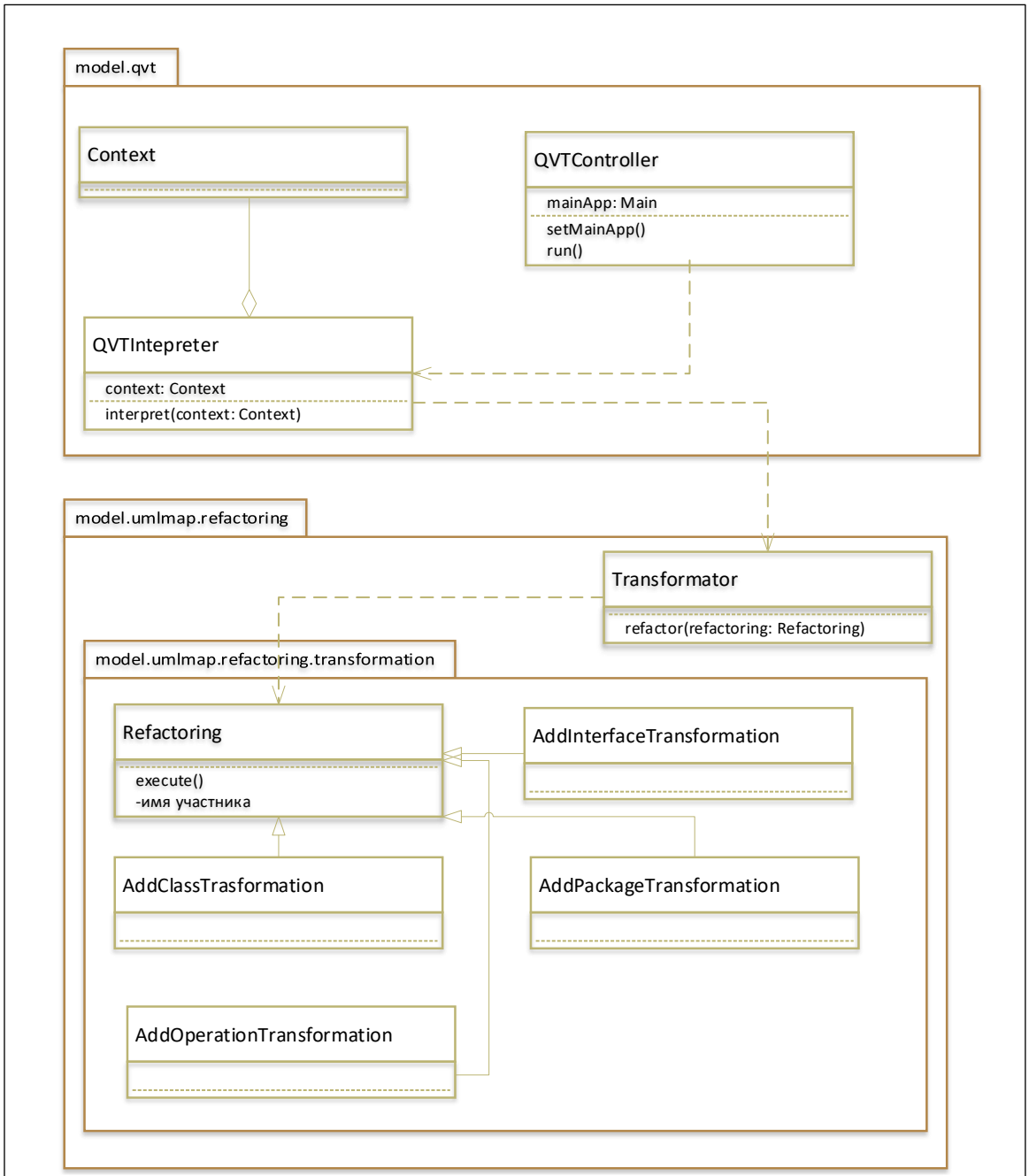

Рис. 2. Взаимодействие базовых классов UML Refactoring и класса QVTInterpreter.java

Fig. 2. Interaction of the basic UML Refactoring classes with the QVTInterpreter.java class

сов разработан класс QVTInterpreter.java, который интерпретирует QVT-запрос к диаграмме классов, а затем преобразует его в последовательность трансформаций, таких как добавление класса, добавление атрибута к классу, добавление метода к классу, добавление интерфейса, добавление метода к интерфейсу, добавление пакета, добавление класса в пакет, добавление интерфейса в пакет, добавление пакета в пакет.

Для каждой трансформации разработан отдельный класс-наследник класса Refactoring. java, передаваемый в ходе трансформации на вход классу Transformator.java, который, в свою очередь, вызывает метод execute() каждой трансформации.

Разработанный интерпретатор позволяет при помощи QVT-запросов редактировать диаграммы классов, импортированные из XMI-файлов, а также создавать новые, что значи-

тельно расширило функциональность системы UML Refactoring.

В дальнейшем на основе разработанного интерпретатора планируется предоставить пользователю возможность пополнения базы трансформаций системы UML Refactoring, описывая новые трансформации при помощи QVT-скриптов. Таким образом, для расширения списка трансформаций не потребуется доступ к исходному коду системы.

Литература

1. OMG Model Driven Architecture. URL: http://www.omg.org/mda (дата обращения: 26.01.2019).
2. Кузнецов М. MDA – новая концепция интеграции приложений. 2003. URL: https://www.osp.ru/os/2003/09/183391 (дата обращения: 26.01.2019).
3. MOF Query/View/Transformation Specification. Version 1.3. URL: http://www.omg.org/spec/QVT/1.3/ (дата обращения: 26.01.2019).
4. OMG Meta Object Facility Specification 2.5.1. URL: http://www.omg.org/spec/MOF/2.5.1/ (дата обращения: 26.01.2019).
5. OMG Unified Modelling Language UML. Version 2.5. URL: http://www.omg.org/spec/UML/2.5 (дата обращения: 26.01.2019).
6. XML Metadata Interchange (XMI) Specification. Version 2.4.2. URL: http://www.omg.org/spec/XMI/2.4.2 (дата обращения: 26.01.2019).
7. Object Constraint Language Specification. Version 2.4. URL: http://www.omg.org/spec/OCL/2.4 (дата обращения: 26.01.2019).
8. Booch G., Rumbaugh J., Jacobson I. The Unified Modeling Language User Guide. Addison-Wesley Publ., 2005, 512 p.
9. Evans A., France R., Lano K., Rumpe B. Developing the UML as a formal modelling notation. Proc. UML'98 LNCS, 1998, vol. 1618, pp. 336–348.
10. Efrizoni L., Wan-Kadir W.M.N., Mohamad R. Formalization of uml class diagram using description logics. Proc Intern. Sympos. IEEE ITSim, 2010, vol. 3, pp. 1168–1173. DOI: 10.1109/ITSIM.2010.5561621.
11. Rahmoune Y., Chaoui A., Kerkouche E. A framework for modeling and analysis UML activity diagram using graph transformation. Proc. Comp. Sc., 2015, vol. 56, pp. 612–617. DOI: 10.1016/j.procs.2015.07.261.
12. Beckert B., Keller U., Schmitt P.H. Translating the object constraint language into first-order predicate logic. Proc. VERIFY Workshop FLoC, 2002, pp. 113–123.
13. Labbani O. A UML and colored petri nets integrated modeling and analysis approach using graph transformation. J. of Object Technology, 2010, vol. 9, no. 4, pp. 25–43.
14. Kerievsky J. Refactoring to Patterns. Boston, Addison-Wesley, 2004, 430 p. DOI: 10.1007/978-3-540-27777-4_54.
15. Gamma E., Richard H., Ralph J., Vlissides J. Design patterns: Abstraction and reuse of object-oriented design. Proc. ECOOP'93 Springer, Berlin Heidelberg, 1993, LNCS 707, pp. 406–431. DOI: 10.1007/3-540-47910-4_21.
16. Harman M., Mansouri S.A., Zhang Y. Search-based software engineering: Trends, techniques and applications. ACM CSUR, 2012, vol. 45, no. 1, p. 11. DOI: 10.1145/2379776.2379787.
17. Amoui M., Mirarab S., Ansari S., Lucas C. A genetic algorithm approach to design evolution using design pattern transformation. J. Comput Inform. Tech., 2006, vol. 1, no. 2, pp. 235–244.
18. Vathsavayi S., Koskimies K., Sievi-Korte O., Kundi H. Tool support for software architecture design with genetic algorithms. Proc. ICSEA, IEEE, 2010, pp. 359–366. DOI: 10.1109/ICSEA.2010.61.
19. O'Keeffe M., Cinnéide M.O. Search-based software maintenance. CSMR. Proc. 10th Europ. Conf. IEEE, 2006, pp. 249–260.
20. Lutz R. Evolving good hierarchical decompositions of complex systems. JSA, 2001, vol. 47, no. 7, pp. 613–634. DOI: 10.1016/S1383-7621(01)00019-4.
21. Bowman M., Briand L.C., Labiche Y. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE Transactions on Software Engineering, 2010, vol. 36, no. 6, pp. 817–837. DOI: 10.1109/TSE.2010.70.
22. Дерюгина О.А. Программно-математические средства рефакторинга с учетом заданных критериев качества // Cloud of Science. 2018. Т. 5. № 1. С. 86–138.
23. Дерюгина О.А. Семантика и семантически эквивалентные трансформации UML-диаграмм классов // Тр. МФТИ. 2015. Т. 7. № 2. С. 146–155.
24. Дерюгина О.А., Никульчев Е.В. Инструментальное средство автоматизированного рефакторинга UML-диаграмм классов по заданным критериям качества // Кибернетика и программирование. 2017. № 1. С. 107–118. DOI: 10.7256/2306-4196.2017.1.21934.

Software & Systems
DOI: 10.15827/0236-235X.127.389-397

Received 28.01.19
2019, vol. 32, no. 3, pp. 389–397

Design of the QVT Operational Mappings interpreter for UML Refactoring in terms of the model driven architecture approach

O.A. Deryugina [1], *Ph.D. (Engineering), Lecturer, o.a.derugina@yandex.ru*
E.V. Kryuchkova [1], *Bachelor of Science*

[1] *MIREA – Russian Technological University, Moscow, 119454, Russian Federation*

Abstract. The paper discusses the MDA (Model Driven Architecture) approach, which has been introduced by the OMG consortium and is aimed at the automation of the software development process. MDA proposes the following steps of the software development: design of the Platform Independent Model (PIM), design of the Platform Specific Model (PSM), development of the Code Model.

The paper provides an overview of the MDA standards: XMI (XML Metadata Interchange), which unifiers model and metamodel interchange between software products; QVT (Query/View/Transformation), which describes model query languages.

The paper is aimed at the design of the QVT Operational Mappings language Interpreter for the UML Refactoring tool. The UML Refactoring tool provides the UML class diagram analysis and transformation. Typically, UML class diagrams are used to describe the software object-oriented architecture. UML Refactoring tool provides object-oriented metrics calculation (Avg, DIT, Avg. NOC, Avg. CBO, etc.) and searching for the transformations (Interface Insertion, Façade, Strategy) minimizing the refactoring fitness function value, which has been chosen by a user.

Based on the information about the QVTo language, the Interpreter class has been designed for the UML Refactoring tool. This class translates QVT commands to the sequence of the transformations of the UML class diagram including add class transformation, add attribute to class, add method to class, add interface, add attribute to interface, add method to interface, add package, add class to package, add interface to package, add package to package. For each transformation, there is a newly designed class to extend Refactoring.java class. This class is an input for the Transformator.java class, which calls method execute() of the Refactoring.java class.

Keywords: UML, XMI, software design, software architecture, MDA, refactoring, UML class diagram refactoring, UML Refactoring, UML, QVT.

References

1. *OMG Model Driven Architecture*. Available at: http://www.omg.org/mda (accessed January 26, 2019).
2. Kuznetsov M. *MDA – New Application Integration Concept*. 2003. Available at: https://www.osp.ru/os/2003/09/183391 (accessed January 26, 2019).
3. *MOF Query/View/Transformation Specification. Version 1.3*. Available at: http://www.omg.org/spec/QVT/1.3/ (accessed January 26, 2019).
4. *OMG Meta Object Facility Specification 2.5.1*. Available at: http://www.omg.org/spec/MOF/2.5.1/ (accessed January 26, 2019).
5. *OMG Unified Modelling Language UML. Version 2.5*. Available at: http://www.omg.org/spec/UML/2.5 (accessed January 26, 2019).
6. *XML Metadata Interchange (XMI) Specification. Version 2.4.2*. Available at: http://www.omg.org/spec/XMI/2.4.2 (accessed January 26, 2019).
7. *Object Constraint Language Specification. Version 2.4*. Available at: http://www.omg.org/spec/OCL/2.4 (accessed January 26, 2019).
8. Booch G., Rumbaugh J., Jacobson I. *The Unified Modeling Language User Guide*. Addison-Wesley Prof. Publ., 2005, 512 p.
9. Evans A., France R., Lano K., Rumpe B. Developing the UML as a formal modelling notation. *UML'98 LNCS*. 1998, vol. 1618, pp. 336–348.
10. Efrizoni L., Wan-Kadir W.M.N., Mohamad R. Formalization of UML class diagram using description logics. *IEEE Proc. 2010 Intern. Symp. in Information Technology (ITSim)*. 2010, vol. 3, pp. 1168–1173. DOI: 10.1109/ITSIM.2010.5561621.
11. Rahmoune Y., Chaoui A., Kerkouche E. A framework for modeling and analysis UML Activity diagram using graph transformation. *Procedia Computer Science*. 2015, vol. 56, pp. 612–617. DOI: 10.1016/j.procs.2015.07.261.

12. Beckert B., Keller U., Schmitt P. H. Translating the Object constraint language into first-order predicate logic. *Proc. VERIFY Workshop at Federated Logic Conf. (FLoC)*. 2002, pp. 113–123.

13. Labbani O. A UML and colored petri nets integrated modeling and analysis approach using graph transformation. *J. of Object Technology*. 2010, vol. 9, no. 4, pp. 25–43.

14. Kerievsky J. Refactoring to Patterns. Boston, Addison-Wesley Publ., 2004, 430 p. DOI: 10.1007/978-3-540-27777-4_54.

15. Gamma E., Richard H., Ralph J., John Vl. Design patterns: Abstraction and reuse of object-oriented design. *Proc. ECOOP'93*. Springer, Berlin Heidelberg Publ., 1993, LNCS 707, pp. 406–431. DOI: 10.1007/3-540-47910-4_21.

16. Harman M., Mansouri S.A., Zhang Y. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*. 2012, vol. 45, no. 1, pp. 11. DOI: 10.1145/2379776.2379787.

17. Amoui M., Mirarab S., Ansari S. and Lucas C. A genetic algorithm approach to design evolution using design pattern transformation. *Intern. J. of Information Technology and Intelligent Computing*. 2006, vol. 1, no. 2, pp. 235–245.

18. Vathsavayi S., Koskimies K., Sievi-Korte O., Kundi H. Tool support for software architecture design with genetic algorithms. *Proc. 5th Intern. Conf. ICSEA, IEEE*. 2010, pp. 359–366. DOI: 10.1109/ICSEA.2010.61.

19. O'Keeffe M., Cinnéide M.O. Search-based software maintenance. CSMR 2006. *Proc. 10th European Conf. on Software Maintenance and Reengineering, IEEE*. 2006, pp. 249–260.

20. Lutz R. Evolving good hierarchical decompositions of complex systems. *J. of Systems Architecture*. 2001, vol. 47, no. 7, pp. 613–634. DOI: 10.1016/S1383-7621(01)00019-4.

21. Bowman M., Briand L.C., Labiche Y. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Trans. on Software Engineering*. 2010, vol. 36, no. 6, pp. 817–837. DOI: 10.1109/TSE.2010.70.

22. Deryugina O.A. Mathematical software for refactoring with given quality criteria. *Cloud of Science*. 2018, vol. 5, no. 1, pp. 86–138 (in Russ.).

23. Deryugina O.A. Semantics and semantically equivalent transformations of UML class diagrams. *Proc. of MIPT*. 2015, vol. 7, no. 2, pp. 146–155 (in Russ.).

24. Deryugina O.A., Nikulchev E.V. A tool for automated refactoring of UML class diagrams according to specified quality criteria. *Cybernetics and Programming*. 2017, no. 1, pp. 107–118. DOI: 10.7256/2306-4196.2017.1.21934 (in Russ.).

Для цитирования

Дерюгина О.А., Крючкова Е.В. Проектирование интерпретатора языка QVT Operational Mappings для программного средства UML Refactoring в рамках модельно-ориентированного подхода // Программные продукты и системы. 2019. Т. 32. № 3. С. 389–397. DOI: 10.15827/0236-235X.127.389-397.

For citation

Deryugina O.A., Kryuchkova E.V. Design of the QVT Operational Mappings interpreter for UML Refactoring in terms of the model driven architecture approach. *Software & Systems*. 2019, vol. 32, no. 3, pp. 389–397 (in Russ.). DOI: 10.15827/0236-235X.127.389-397.