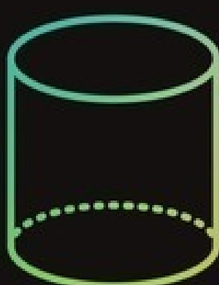
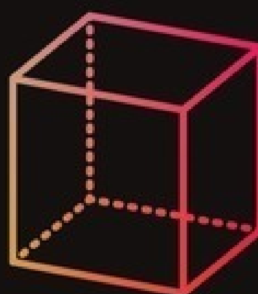
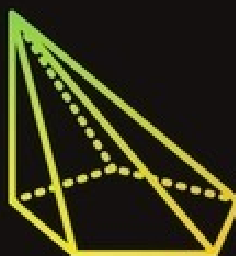
ПРОЕКТИРОВАНИЕ НА UML

Сборник задач

Хританков • Полежаев • Андрианов

Антон Хританков

Проектирование на UML. Сборник задач

«Издательские решения»

Хританков А. С.

Проектирование на UML. Сборник задач / А. С. Хританков — «Издательские решения»,

ISBN 978-5-44-857954-7

В данном сборнике представлены задачи по проектированию ПО с использованием унифицированного языка моделирования UML 2, принципов и паттернов проектирования. Сборник содержит более 120 задач с несколькими заданиями в каждой по разным разделам UML и проектирования ПО. Для каждого раздела приводятся основные понятия, для задач даны ответы и пояснения по решению.http://www.objectoriented.ru

ISBN 978-5-44-857954-7

© Хританков А. С.
© Издательские решения

Содержание

Проектирование на UML. 7
ОБ АВТОРАХ 8
ПРЕДИСЛОВИЕ КО ВТОРОМУ ИЗДАНИЮ 9
ПРЕДИСЛОВИЕ К ПЕРВОМУ ИЗДАНИЮ 10
ГЛАВА 1. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО МОДЕЛИРОВАНИЯ 11
§1. КЛАССЫ И ОБЪЕКТЫ 13
ОСНОВНЫЕ ПОНЯТИЯ 13
ЗАДАЧИ 15
§2. СЦЕНАРИИ И ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ 18
ОСНОВНЫЕ ПОНЯТИЯ 18
ЗАДАЧИ 19
§3. КООПЕРАЦИИ И ВЗАИМОДЕЙСТВИЯ КЛАССОВ 22
ОСНОВНЫЕ ПОНЯТИЯ 22
ЗАДАЧИ 24
ГЛАВА 2. МЕТОДЫ И ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 28
§4. РАСШИРЕННЫЕ КЛАССЫ И ОБЪЕКТЫ 30
ОСНОВНЫЕ ПОНЯТИЯ 30
ЗАДАЧИ 32
§5. АНАЛИЗ И ВЫДЕЛЕНИЕ КЛАССОВ 38
ОСНОВНЫЕ ПОНЯТИЯ 38
ЗАДАЧИ 38
§6. АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ, КОМПОНЕНТЫ 41
ОСНОВНЫЕ ПОНЯТИЯ 41
ЗАДАЧИ 43
§7. СХЕМЫ СОСТОЯНИЙ И КОНЕЧНЫЕ АВТОМАТЫ 47
ОСНОВНЫЕ ПОНЯТИЯ 47
ЗАДАЧИ 48
§8. ПРЕДСТАВЛЕНИЕ ДЕЯТЕЛЬНОСТИ И ПОТОКОВ РАБОТ 54
ОСНОВНЫЕ ПОНЯТИЯ 54
ЗАДАЧИ 56
§9. ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ 61
ОСНОВНЫЕ ПОНЯТИЯ 61
ЗАДАЧИ 62
§10. МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ В СТРУКТУРЕ КЛАССОВ 68
ЗАДАЧИ 68
ГЛАВА 3. УКАЗАНИЯ, ОТВЕТЫ И РЕШЕНИЯ ЗАДАЧ 73
§11. МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ 73
§12. ПРИМЕРЫ ЛАБОРАТОРНЫХ И КОНТРОЛЬНЫХ РАБОТ 83
§13. ОТВЕТЫ И РЕШЕНИЯ 87
13.1. КЛАССЫ И ОБЪЕКТЫ 87
13.2. СЦЕНАРИИ И ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ 88
13.3. КООПЕРАЦИИ И ВЗАИМОДЕЙСТВИЯ КЛАССОВ 90
13.4. РАСШИРЕННЫЕ КЛАССЫ И ОБЪЕКТЫ 93

13.5. АНАЛИЗ И ВЫДЕЛЕНИЕ КЛАССОВ 101
13.6. АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ, КОМПОНЕНТЫ 104
13.7. СХЕМЫ СОСТОЯНИЙ И КОНЕЧНЫЕ АВТОМАТЫ 111
13.8. ПРЕДСТАВЛЕНИЕ ДЕЯТЕЛЬНОСТИ И ПОТОКОВ РАБОТ 115
13.9. ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ 120
13.10. МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ В СТРУКТУРЕ КЛАССОВ 129
ЛИТЕРАТУРА 140

Проектирование на UML
Сборник задач

Антон Сергеевич Хританков
Валентин Александрович Полежаев
Андрей Иванович Андрианов

© Антон Сергеевич Хританков, 2017
© Валентин Александрович Полежаев, 2017
© Андрей Иванович Андрианов, 2017

ISBN 978-5-4485-7954-7
Создано в интеллектуальной издательской системе Ridero

Проектирование на UML. Сборник задач по проектированию программных систем

Рекомендовано ученым советом ФИВТ МФТИ
к использованию в учебном процессе факультета
при подготовке студентов по направлениям
010400 «Прикладные математика и информатика» и
010600 «Прикладные математика и физика»
Рецензенты:
д.ф.-м.н., профессор, Соколинский Л. Б.,
ведущий разработчик, Колпаков Е. А.

Аннотация

В данном сборнике представлены задачи по проектированию программных систем с использованием унифицированного языка моделирования UML2, принципов и паттернов проектирования. Сборник содержит более 120 задач с несколькими заданиями в каждой по разным разделам UML и проектирования ПО. Для каждого раздела приводятся основные понятия, для задач даны ответы и пояснения по решению. Приведены рекомендации по составлению проверочных работ с использованием задач сборника по темам проектирования.

Для слушателей курсов по объектно-ориентированному анализу и проектированию программного обеспечения, студентов технических и физико-математических специальностей, преподавателей высших учебных заведений, специалистов по программной инженерии.

Дополнительную информацию и материалы можно найти на сайте книги http://www.objectoriented.ru

При цитировании, используйте следующую информацию о книге.

Хританков А. С., Полежаев В. А., Андрианов А. И.

Проектирование на UML. Сборник задач по проектированию программных систем. 2-е. изд. – Екатеринбург.: Издательские решения, 2017. – 240 с.; ил.

ISBN 978-5-4485-7954-7

УДК 004.41+004.02+372.8

ББК 32.973.23—018

(С) Хританков А. С., 2017

(С) Полежаев В. А., 2017

(С) Андрианов А. И., 2017

ОБ АВТОРАХ

Хританков Антон Сергеевич, к.ф.-м.н.
доцент кафедры АТП, Московский физико-технический институт.

Защитил диссертацию в сфере высокопроизводительных вычислений (МФТИ / ИСА РАН). Опыт преподавания более восьми лет, научные интересы: архитектура программного обеспечения, автоматизированные и интеллектуальные методы разработки программ. Опыт работы в индустрии более 12 лет от разработчика ПО до архитектора и руководителя департамента разработки и исследований. Сертифицированный специалист по UML2 (OMG Certified UML Professional Advanced).

Email: anton.khritankov@objectoriented.ru

Полежаев Валентин Александрович
директор по разработке и анализу данных компании Интелиор.

Окончил ВМК МГУ, автор нескольких статей по теме машинного обучения и практике применения предметно-ориентированных методов проектирования. Участвовал в разработке более десяти информационных систем, из них более половины в качестве бизнес-аналитика и архитектора.

Email: valentin.polezhaev@objectoriented.ru

Андрианов Андрей Иванович
руководитель группы морфологии «Аби Продакшн» (ABBYY).

Магистр физ.-мат. наук (МФТИ), в разное время преподавал в МФТИ курсы «Алгоритмы и структуры данных», «Проектирование программных систем», «Машинное обучение». А также «Концепции языков программирования», «Промышленное программирование». Опыт разработки, проектирования архитектуры и управления проектами более 9 лет.

Email: andrey.andrianov@objectoriented.ru

ПРЕДИСЛОВИЕ КО ВТОРОМУ ИЗДАНИЮ

Проектирование – это процесс построения модели объекта, который предполагается разработать или создать. Модели в проектировании играют важную роль: модели используются для уточнения того, что нужно сделать, для прояснения способа реализации, для понимания реализуемости решения и его соответствия требованиям, для передачи знаний и распространения информации о проектируемом объекте, для планирования и ведения работ по реализации.

Во многих инженерных отраслях проектирование занимает важное место и часто регулируется государственными, промышленными стандартами или стандартами уровня предприятия. В сфере разработки программного обеспечение проектирование будущей программной системы происходит как в начале работ, так и по ходу реализации.

Важно понимать, что область знаний проектирования – это отдельная дисциплина, требующая особых навыков работы с моделями, применения методов и практик, которые обычно не используются при реализации создаваемого объекта.

На данный момент в индустрии разработки программного обеспечения сложилось несколько отраслей, каждая из которых использует несколько отличные от других методы проектирования. Среди этого разнообразия в книге уделено больше внимания проектированию прикладных программ, компонентов и приложений с помощью объектно-ориентированных методов и унифицированного языка моделирования UML2.

Овладение этими методами позволит в дальнейшем с легкостью освоить и другие сферы проектирования программных систем, и другие языки моделирования.

В данной книге собрано более сотни задач по проектированию. Авторы приложили все возможные усилия к тому, чтобы задачи помогли читателю понять смысл и освоить те или иные концепции, принципы и методы проектирования. Рассматриваемый перечень тем примерно соответствует программе курса по проектированию программных систем, читаемом авторами в Московском физико-техническом институте на протяжении уже более восьми лет и рекомендациям ACM/IEEE по составу учебных программ по проектированию программного обеспечения.

По сравнению с первым изданием книгу пополнили задачи, предлагавшиеся студентам на контрольных работах по проектированию программного обеспечения, вошел новый раздел по предметно-ориентированному проектированию, добавлены рекомендации по составлению проверочных и контрольных работ по отдельным темам проектирования, а также вошло множество задач разной сложности, которые авторы сочли интересными и важными для освоения дисциплины проектирования.

По сравнению с первым изданием в книге изменился состав авторов, тем не менее, важно отметить вклад Штукатурова А.Н, по согласованию с которым во второе издание вошли подготовленные им задачи 2.5, 3.7, 3.9, 4.4, 5.5, 5.6, 7.5, 8.9. Раздел §9 подготовлен Полежаевым В. А., задачи 6.2, 6.4, 2.3, 2.8, 3.10, 7.11, 7.12, 1.3, 1.4, 2.1, 7.1, 8.1, 3.2 предложены Андриановым А. И., остальные задачи, теоретическая справка и примеры решения задач составлены Хританковым А. С.

Апрель 2017

ПРЕДИСЛОВИЕ К ПЕРВОМУ ИЗДАНИЮ

Предлагаемая читателю книга является сборником задач по курсу проектирования программных систем, преподаваемому авторами в Московском физико-техническом институте.

Сборник включает задачи по проектированию и моделированию с помощью языка UML2. Каждая задача имеет условие, в котором описана заготовка модели, и несколько вопросов к ней. Часть вопросов направлена на уточнение и расширение заготовки модели. Другая часть проверяет понимание смысла построенной модели. Предполагается, что вопросы к задаче будут решаться по порядку.

Задачи сборника направлены на развитие навыков изложения проектных решений средствами UML и устроены таким образом, чтобы в наибольшей степени обеспечить единственность решения. В сложных случаях к задачам даны пояснения и указания по решению.

В сборник включены задачи по объектно-ориентированному моделированию предметной области, в том числе задачи на выделение классов, задачи по моделированию структур времени выполнения и размещения компонентов программных систем. Моделированию поведения систем в сборнике посвящены разделы описания взаимодействий, моделирования с помощью конечных автоматов и представления деятельности.

Разделы сборника снабжены пояснениями по основным понятиям. В конце сборника приведены ответы к задачам с пояснениями, приводятся примеры решения некоторых задач.

Сборник может быть использован при проведении семинаров по курсам объектно-ориентированного моделирования и программирования, проектирования программных систем, а также специализированных курсов по языку UML. Задачи сборника могут предлагаться в качестве примеров, демонстрирующих и поясняющих основные понятия и особенности языка, могут входить в задания для самостоятельной работы, проверочные и контрольные работы, использоваться для самостоятельного изучения методов проектирования и языка UML.

Задачи составлены авторами на основе собственного опыта преподавания, адаптированы из профессиональной практики, проведенных контрольных и проверочных работ, предлагаемых студентам проектов.

Июнь 2012

ГЛАВА 1. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО МОДЕЛИРОВАНИЯ

Краткая история UML. Унифицированный язык моделирования UML появился в результате объединения нескольких подходов к моделированию в середине 1990-х годов. В отличие от предыдущих попыток, в создании языка участвовали авторы этих подходов, а вследствие стандартизации через организацию OMG, участвовали также заинтересованные компании и исследовательские коллективы из разных отраслей.

Поэтому разные части UML отражают потребности в моделировании в разных отраслях и объединение их в одном языке и на основе одной базовой системы понятий позволяет говорить как раз об унифицированном языке моделирования.

В данной книге унифицированный язык моделирования выбран как основной для выражения проектировочных решений. При решении задач стоит ориентироваться на версию UML 2.4.1, которая была стандартизирована международной организацией по стандартизации ISO как ISO/IEC 19505—1:2012 и ISO/IEC 19505—2:2012.

Уровни использования UML. Выделяют несколько уровней владения и использования UML и моделей в целом при проектировании программных систем. На уровне эскиза модели используются для пояснения решений, неформального общения, документирования и не обладают полнотой, строгостью и могут быть несогласованными. На уровне спецификации модель используется как чертеж или план реализации, в соответствии с которым разрабатывается программная система.

Модели и диаграммы на этом уровне должны следовать нотации языка и быть **согласованными** (**well-formed**). Согласованность модели означает соответствие правилам использования языка UML2, определенным в его спецификации (метамодели) [4]. Например, если на диаграмме показан элемент модели, то в модели также должны быть определены все элементы, используемые показанным.

На исполняемом уровне модель представляет собой достаточное описание системы для ее воплощения автоматическими средствами. В этом случае исходный код системы может не сохраняться вовсе и быть промежуточным этапом получения работающей программной системы из исходных моделей.

В данном сборнике задач следует ориентироваться на использование UML на уровне спецификации. В то же время часть задач предполагает владение языком на исполняемом уровне.

Решение задач. Прежде чем приступить к решению задач стоит ознакомиться с рекомендуемой литературой для ознакомления с методами проектирования и нотацией. В помощь читателю в начале каждого раздела приводится краткая справка по используемым в задачах раздела понятиям и демонстрируется нотация языка.

Все задачи построены по единому принципу. В условии дается заготовка модели. Это может быть диаграмма или текстовое описание. В текстовом описании названия элементов модели приведены *курсивом* для облегчения их восприятия. Далее приводится несколько заданий или вопросов к условию. В качестве решения задания нужно указать по шагам ход рассуждения от условия или предыдущего задания к достижению условий, указанных в задании. Для ответа на вопрос следует привести рассуждение в обоснование полученного ответа и сам ответ. Задания и вопросы к задачам следует выполнять по порядку. Решение следующего задания может зависеть от решения предыдущего. Ответом на задание будет фрагмент диаграммы или нескольких диаграмм с представлением изменений, требуемых в данном задании.

При решении следует руководствоваться условием задачи, знаниями методов решения, нотацией и значением понятий языка моделирования. Часть задач составлена на основе реаль-

ных проектов разработки программного обеспечения, другую часть составляю учебные задачи. Такие задачи могут вызывать ассоциации с похожими ситуациями или реальными объектами. В этом случае следует придерживаться условия задачи. Если не указано иное, решение задач не предполагает каких-либо специальных знаний в специализированных областях. При необходимости дается сноска, где можно получить дополнительную информацию.

Задачи и задания повышенной сложности отмечены звездочкой (*), для некоторых задач приведено решение, в этом случае указана страница, на которой оно расположено (см. решение в §11). Перед тем, как приступить к решению задач рекомендуется ознакомиться с примерами решения и понять порядок ведения рассуждения и степень его детальности.

При составлении задач уделялось особое внимание тому, чтобы решение было единственным. При необходимости в заданиях к задачам даются указания по предполагаемому способу решения. Впрочем, вполне возможно, что читатель сможет предложить более удачные или лаконичные решения по некоторым задачам. Возможность существования лучшего решения следует учитывать и не требовать однозначного совпадения решения с ответами, приводимыми авторами сборника.

§1. КЛАССЫ И ОБЪЕКТЫ

ОСНОВНЫЕ ПОНЯТИЯ

Пространство имен (namespace) – это именованный элемент модели, который может содержать другие именованные элементы. Принадлежность пространству имен показывается отношением **включения в пространство имен (membership)**. **Полностью квалифицированное имя (fully-qualified name)** элемента в модели состоит из последовательности имен всех вложенных пространств имен, в которые включен элемент.

Классификатор (classifier) – это пространство имен в модели, указывает на общие некоторому множеству объектов черты. Черты классификатора могут быть поведенческими, структурными или соединительными.

Класс (class) – это классификатор, который описывает некоторую концепцию моделируемой области. Черты класса могут быть различных видов, наиболее часто для описания функциональности класса используются операции (operation), а для описания хранимых данных или связей с другими классами **– свойства (property)**. Если типом свойства является примитивный тип или тип данных, свойства показывают как атрибуты, класса иначе как часть ассоциации.

Операция (operation) – черта поведения интерфейса, класса или типа данных. Операция задается именем, набором параметров, типом возвращаемого значения и его кратностью. Каждый параметр операции может иметь имя, тип, кратность. В программировании операции будет соответствовать сигнатура метода.

Обратите внимание, что определение операции в классе не влечет определение ее реализации в этом классе. Понятие метода в UML2 обозначает реализацию операции алгоритмом, который не описывается средствами UML или не уточняется в модели. В последнем случае, такую реализацию операции называют нечетким поведением (opaqueBehavior).

Интерфейсом (interface) называют особый вид классификатора, который определяет способ взаимодействия с экземпляром класса, реализующего интерфейс. Интерфейс обычно включает операции, но может включать и свойства. В последнем случае наличие указанных свойств является обязательным для реализующего интерфейс класса.

Экземпляр класса (instance) – это элемент модели с описанием, возможно неполным, объекта, которому в системе приписаны черты данного класса. Для того чтобы указать значения свойствам класса в экземпляре используют слоты.

Связью (link) называется экземпляр ассоциации, соединяющий экземпляры классов. В языке программирования однонаправленной связи соответствует типизированный указатель или ссылка.

Ассоциация (association) – это типизированное отношение между классами, которое указывает на логическую связь между ними. Ассоциация имеет два или более полюсов, по одному у каждого связанного класса. Название полюса обычно указывает на роль, которую класс играет в ассоциации.

Обобщение (generalization) является направленным отношением от более специализированного классификатора к более общему. Специализированный, или дочерний, классификатор наследует черты более общего, или родительского, классификатора. Отношение обобщения уточняется отдельно для каждого вида классификатора, в том числе для классов и интерфейсов.

Украшениями (adornments) называются свойства полюса ассоциации, уточняющие роль участвующего в ассоциации класса. С помощью украшений указываются направление навигации, вид композиции и другие свойства полюса.

Типом данных (data type) называется классификатор, экземпляры которого не обладают индивидуальностью и, при совпадении значений свойств, взаимозаменяемы. **Простыми (primitive)**, или примитивным типами данных, являются предопределенные типы: целое *Integer*, строка *String*, логический тип *Boolean*, числа с плавающей запятой *Real* и неограниченные натуральные числа *UnlimitedNatural*, которые используются для моделирования неопределенного количества элементов, например, экземпляров класса, участвующих в ассоциации.

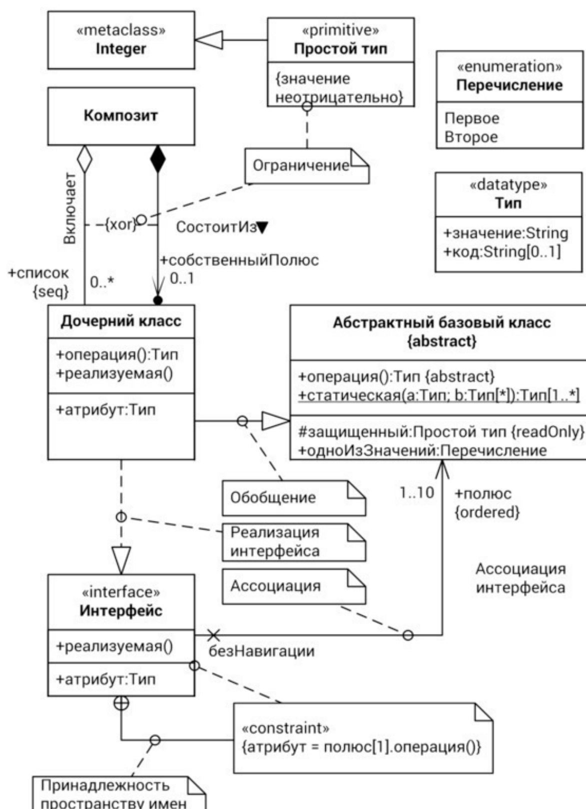


Рис. 1. Основная нотация диаграмм классов

Рис. 2. Нотация диаграмм экземпляров

Ограничением (constraint) называется логическое выражение об ограничиваемых элементах модели, вычисляемое в контексте какого-либо элемента. Если выражение ложно, то модель считается противоречивой (ill-formed).

Примеры нотации указанных выше элементов модели приведены на рис. 1 и рис. 2.

ЗАДАЧИ

1.1. Абстрактный класс *Account* имеет два дочерних класса: счет физического лица *PersonalAccount* и юридического *CompanyAccount*. При решении задачи используйте диаграммы классов.

а. Добавьте класс *Person* с общедоступным атрибутом *FullName* строкового типа и свяжите его с классом *PersonalAccount* ассоциацией *Owns* с полюсом *owner* у *Person* и навигируемым полюсом *account* у *PersonalAccount*.

б. Аналогично для счета юридического лица добавьте владельца *Company*, свяжите анонимной ассоциацией с *CompanyAccount* и укажите подходящие названия полюсов.

в. Добавьте класс адреса *Address* с атрибутами строкового типа *street*, *city* и целочисленным положительным *building*. Укажите с помощью новых анонимных ассоциаций, что *Person* может иметь адрес регистрации *registeredAt*, фактический адрес *actual*, в то время как компания связана с одним юридическим адресом *legalAddress* и может иметь почтовый адрес *postAddress*.

1.2. Интерфейс *Stack* определяет операции помещения в стек *push* с параметром *obj* типа *Element*, операцию получения элемента из стека *pop* с возвращаемым значением типа *Element*. При решении задачи используйте диаграммы классов.

а. Добавьте в интерфейс *Stack* операции очистки стека *reset*, которая не имеет параметров, статическую операцию создания нового стека *createNew* с возвращаемым значением типа *Stack*.

б. Покажите, что интерфейс *Stack* зависит от типа данных *Element*.

в. Добавьте класс *ListStack*, который реализует интерфейс *Stack*. Покажите реализуемые классом операции интерфейса.

г. Добавьте в класс *ListStack* частное структурное свойство *arr* типа *Element* с кратностью больше нуля, значения которого упорядочены и могут повторяться.

д. Добавьте частный целочисленный атрибут *increment* только для чтения и защищенную операцию изменения размера *resize* с целочисленным параметром *newSize*.

е. Покажите на диаграмме экземпляров экземпляр *stack* класса *ListStack*, свойство *arr* которого содержит элемент *first* типа *Element* первым и *second* того же типа вторым. Укажите, что атрибут *increment* экземпляра *stack* равен *10*.

1.3. В пространстве имен *Time* расположены перечисления *Month*, *DayOfWeek*, а также классы *Date* и *Period*. При решении задачи используйте диаграммы классов.

а. Укажите, что перечисление *Month* может принимать значения: *Jan*, *Feb*, *Mar*, *Apr*, *May*, *Jun*, *Jul*, *Aug*, *Sep*, *Oct*, *Nov*, *Dec*.

б. Укажите, что перечисление *DayOfWeek* может принимать значения: *Mon*, *Tue*, *Wed*, *Thu*, *Fri*, *Sat*, *Sun*.

в. Добавьте классу *Date* частные атрибуты *year*, *month*, *dayOfMonth* типа *Integer*, а также общедоступные операции:

– получения года *getYear* типа *Integer*; – получения месяца *getMonth* типа *Month*; – получения дня *getDayOfMonth* типа *Integer*; – получения дня недели *getDayOfWeek* типа *DayOfWeek*.

г. Добавьте классу *Date* общедоступную статическую операцию *now ()* типа *Date*.

д. Добавьте классу *Period* общедоступную статическую операцию *between*. У операции два аргумента: *from* и *to*. Оба аргумента имеют тип *Date*. Операция возвращает значение типа *Period*

е. Добавьте классу *Date* операцию *plus* с аргументом *delta* типа *Period*. Результат операции – значение типа *Date*.

1.4. Класс *MyWindow* уточняет абстрактный базовый класс *Window*. *MyWindow* состоит (композиция) из кнопки класса *Button* и надписи класса *Label*. Отобразите на диаграмме классов.

а. Класс *Label* имеет частный атрибут *text* типа *String* и общедоступную операцию *setText* с параметром *text* типа *String*.

б. Композиция между *MyWindow* и *Button* называется *HoldsButton*. Полюс со стороны кнопки имеет имя *okButton*, защищенную видимость, кратность *1*. Композиция между *MyWindow* и *Label* называется *HoldsLabel*. Украшения полюса со стороны *Label*: название *textLabel*, частная видимость, кратность *1*.

в. Для реакции на события кнопки реализован паттерн Слушатель (Listener) следующим образом. Класс *Button* предоставляет операцию *setClickListener* с единственным параметром *l* типа *IClickListener*. Интерфейс *IClickListener* содержит единственную операцию *onClick* без параметров.

г. Класс *MyWindow* реализует интерфейс *IClickListener* для реакции на нажатие кнопки. Отобразите на диаграмме, что между классом *Button* и *MyWindow* есть ассоциация с именем *NotifyListener* с направлением от кнопки к окну. Укажите, что полюс со стороны окна называется *listener*, имеет тип *IClickListener*, множественную кратность и частную видимость.

д. И *Label* и *Button* имеют строковый атрибут *text*, который можно менять с помощью метода *setText*. Вынесите общий атрибут и метод в абстрактный базовый класс *TextWidget*.

е. Отобразите на диаграмме объектов, как в процессе выполнения объекты связаны между собой: объект *window* класса *MyWindow* связан с кнопкой *button* класса *Button* и с надписью *label* класса *Label*.

1.5. (см. решение в §11) Интерфейс доступа к коллекции элементов *Collection* обобщает интерфейс работы со списками *List*. Абстрактный класс *BaseCollection* реализует интерфейс *Collection*, абстрактный класс *BaseList* является потомком *BaseCollection* и реализует интерфейс *List*, оставляя операции по хранению данных дочерним классам.

а. Используя наследование, добавьте в модель класс *ArrayList*, реализующий операции со списками с помощью массива.

б. Пусть интерфейс *List* содержит операцию *get* получения элемента списка по заданной позиции *k*. Укажите, в каких классах должна быть объявлена данная операция, чтобы модель была согласованной. Ответ поясните.

в. Пусть интерфейс *Collection* содержит операцию *add* добавления элемента *obj*. Укажите, в пространстве имен каких классов может присутствовать поведение, реализующее операцию *add*. Ответ поясните.

1.6. Класс *Collections* содержит общедоступную статическую операцию *addAll* с возвращаемым значением типа *boolean*. Первый параметр операции называется *coll* и имеет тип *Collection*, второй параметр называется *elements* и имеет тип *Object* и кратность больше нуля.

а. Добавьте в класс *Collections* статический атрибут *empty* типа *Collection,* предназначенный только для чтения.

б. Реализуйте в классе *Collections* операцию *addAll* с помощью нечеткого поведения (метода), используя операцию добавления элемента *insert (e: Object)* класса *Collection*. *Указание*. Алгоритм реализации можно показать как псевдокод в комментарии в формате *{method = {<language>} <method body>}*.

1.7. Узел дерева *Node* может иметь несколько дочерних *child* узлов того же класса *Node*.

а. Приведите пример бинарного дерева, состоящего из семи узлов *Node*.

б. Постройте модель дерева, в котором каждый узел имеет от двух до четырех дочерних узлов.

в. Разработайте модель дерева, узлы которого могут быть двух видов: узел *Red* и узел *Black*. *Указание.* Вид узла может изменяться, при этом следует считать, что поведение узла не изменяется при смене типа.

1.8. У абстрактного класса заказа *Reservation* имеется два потомка: одиночный *Single* и подписка *Subscription*. *Single* связан с одним билетом *Ticket* ассоциацией бронирован *reserved*, *Ticket* может быть связан той же ассоциацией не более чем с одним *Single*.

а. Свяжите подписку с билетами в количестве от трех до шести включительно. Билет не обязательно связан с подпиской.

б. Как с помощью ограничений указать, что билет не может быть одновременно связан и с подпиской, и с одиночным заказом?

в. Пусть одиночная подписка наследует свойства одиночного заказа и подписки. С каким максимальным количеством билетов она может быть связана? Ответ поясните.

1.9. Умный дачный домик *SmartHouse* состоит из четырех стен *Wall* и крыши *Roof*. Домик реагирует на штормовые предупреждения *stormWarning* и укрепляет крышу *harden*, закрывает окна *closeWindows* в стенах. Используемые стройматериалы *Material* характеризуются ценой *price* и удельным весом *unitWeight*.

а. Добавьте стройматериалы для постройки домика: красный и белый кирпич *Brick*, доски *Plank* из сосны и дуба.

б. Укажите, что кирпич является материалом *material* стен. Используя ассоциации, покажите, что каркас крыши *Frame* сделан из не более чем сорока досок и может быть одного из видов *FrameKind*: мансарда, плоский или треугольный.

в. Каркас можно покрыть стройматериалом черепица *Tiling*, отразите это в модели.

г. Допустим, изобретен универсальный стройматериал, заменяющий доски, кирпичи и черепицу. Постройте из него дачный домик. Сколько экземпляров материала понадобится? Ответ поясните.

§2. СЦЕНАРИИ И ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ

ОСНОВНЫЕ ПОНЯТИЯ

Актором (actor) называется классификатор, который моделирует пользователя или систему, внешнего по отношению к моделируемой системе или компоненту. Акторов, которые используют систему для достижения собственных целей, называют основными. Акторов, которых система использует для достижения целей других акторов, называют второстепенными.

Вариантом использования (use case) называют классификатор, который описывает совокупность сценариев взаимодействия акторов с системой или компонентом для достижения какой-либо цели, значимой для акторов. Варианты использования могут различаться по уровню цели, достижение которой они обеспечивают: высокоуровневые цели, пользовательские цели и отдельные функции системы.

Субъектом (subject) варианта использования называют систему или компонент, взаимодействие акторов с которым он описывает.

Ассоциация (association) актора с вариантом использования указывает на взаимодействие актора с субъектом в одном из сценариев данного варианта использования.

Отношение **расширения (extension)** между вариантами использования указывает, что при выполнении заданного в **точке расширения (extension point)** условия сценарий расширяемого варианта использования будет приостановлен, и взаимодействие будет продолжено в рамках расширяющего варианта использования.

Отношение **включения (inclusion)** указывает, что в процессе выполнения сценарии базового варианта использования вызывают выполнение сценариев включаемого варианта использования.

Как и для других классификаторов, для акторов и вариантов использования определено отношение **обобщения (generalization)**.

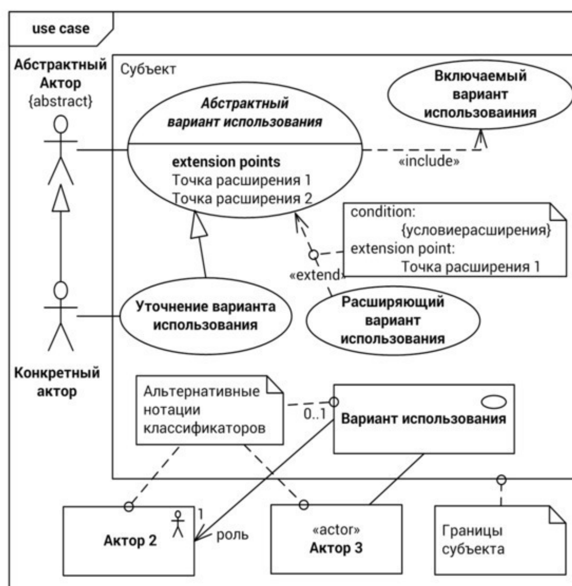


Рис. 3. Основная нотация диаграмм вариантов использования

ЗАДАЧИ

2.1. Актор *User* взаимодействует с системой *OnlineTranslator* в рамках абстрактного варианта использования *Translate*. Варианты использования *TranslateText* и *TranslateWebPage* уточняют *Translate*. Отобразите на диаграмме вариантов использования.

а. Вариант использования *TranslateWebPage* включает *«include»* вариант использования *SetURL*.

б. Вариант использования *SetLanguages* расширяет *«extend»* вариант использования *Translate* в точке расширения *specifyLaguages*. Условие расширения «язык не определен автоматически».

в. Добавьте в модель актора *ExperiencedUser*, уточняющего *User*. *ExperiencedUser* может взаимодействовать с системой в рамках варианта использования *ProposeTranslation*, который уточняет вариант использования *TranslateText*.

2.2. (см. решение в §11) Автор *Author* направляет статью *SendPaper* редактору журнала *Editor*. Редактор передает статью на рецензирование *Review* нескольким рецензентам *Reviewer*. Затем редактор возвращает отзывы рецензентов автору в том же варианте использования *SendPaper*.

а. Добавьте возможность автору вместе с корректором *ProofReader* подготовить статью к публикации *PrepareForPublishing*.

б. Доработайте модель, укажите, что подготовка статьи к публикации выполняется, только если она была одобрена редактором в варианте использования *SendPaper*.

2.3. Распознавателю текста *OCR* от модуля морфологии нужны возможность определить, принадлежит ли слово языку, и функция приведения слова к заданной форме, в частности, восстановления начальной формы. Также нужна функция получения грамматического значения конкретного слова.

а. Постройте модель модуля, выделите акторов, варианты использования и укажите отношения между ними.

б. Добавьте функцию вывода слов, похожих на введенное, если его нет в словаре языка. Каким образом данная возможность системы связана с другими функциями?

в. Укажите в модели, что все перечисленные задачи подразумевают выполнение поиска слова (или его основы) в словаре.

г. Некоторые языки могут не поддерживаться системой. Перед выполнением любой функции модуля морфологии нужно проверить, поддержан ли язык. Отобразите это в модели.

2.4. Ответственное лицо *ResponsiblePerson* может прикрепить документ *AttachToIssue* к обсуждаемому вопросу, выступая в роли автора *author*, и к постановлению *AttachToResolution*, выступая в роли председателя *chairman*.

а. Покажите в модели, что прикрепление документа выполняется согласно общему сценарию прикрепления, реализуемому в частном случае прикрепления к вопросу или прикрепления к постановлению.

б. Добавьте в модель оператора *Operator*, который является ответственным лицом с возможностью удаления документов *DeleteDocument*.

в. Доработайте модель, укажите, что при прикреплении документа рассылается оповещение *SendAnnouncement*. Несколько операторов могут выступать в роли контролеров *controller*.

г. Каким образом можно указать, что прикрепление документа возможно только к вопросу или к постановлению? Ответ поясните.

д. (*) Покажите в модели, что ответственное лицо участвует в сценарии прикрепления в роли пользователя *user*, объединяющей роли автора и председателя. *Указание*. Используйте производные свойства. См. §4.

2.5. Пользователь *User* настраивает подключаемые модули аудиоплеера *AudioPlayer* в рамках варианта использования *ConfigurePlugins*.

а. Добавьте к варианту использования *ConfigurePlugins* возможность выбора определенного модуля для настройки *SelectPlugin* и возможность настройки конкретного модуля *ChangeSettings*.

б. Добавьте в модель возможность обновить подключаемые модули *UpdatePlugins* с внешнего сервера *PluginsServer*.

в. Помимо обычного пользователя в системах обычно есть привилегированный пользователь *SuperUser*, который имеет права на изменение конфигурации системы. В системе аудиоплеера такой пользователь может обновить модули *UpdatePluginsList*. Обновление включает в себя удаление *DeletePlugin*, установку *InstallPlugins* и просмотр списка доступных на сервере *CheckPluginsList*.

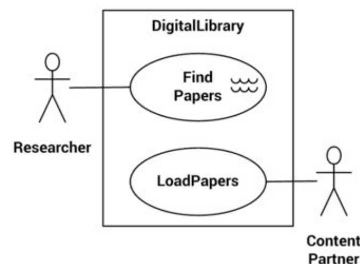


Рис. 4

2.6. Рассмотрим электронную библиотеку научных работ, представленную на рис. 4.

а. Поясните, каким образом используется электронная библиотека. Перечислите актеров и варианты использования.

б. Укажите, что аналитик *Analyst* принимает участие в индексировании статей, выполняемом в процессе их загрузки бизнес-партнером *ContentPartner*.

в. Предоставьте возможность исследователю *Researcher* использовать расширенный поиск *AdvancedSearch*, который позволяет указать другие параметры поиска в *FindPapers*.

г. Укажите, что все варианты использования преследуют цели уровня пользователя (user goal) системы. Указание. Уровни целей[1] не входят в стандарт UML2.

2.7. Клиент *Client* выполняет операции над своими счетами в банке *Bank*, используя банкомат *ATM* в рамках абстрактного варианта использования *PerformOperation*, который включает информирование об услугах в варианте использования *InformAboutServices*. Для выполнения операций *ATM* обращается к платежной системе *PaymentSystem*.

а. Перечислите основных и вспомогательных акторов системы *ATM*. Какие из них взаимодействуют с системой в варианте использования *PerformOperation*?

б. Отразите в модели вариантов использования, что клиенты могут только выполнять операции по получению наличных, в то время как клиенты *BankCustomers* банка, владеющего банкоматом, могут также оплачивать услуги из списка, предоставляемого банком *Bank*. При этом сценарии оплаты услуг и получения наличных отличаются между собой, но следуют общему сценарию выполнения операций.

в. Добавьте возможность получения наличных как в валюте счета, так и в другой валюте. При этом в обоих случаях банкомат запрашивает у клиента *Client* подтверждение на списание средств в валюте счета по курсу банка *Bank*.

[1] Коберн А. Современные методы описания функциональных требований к системам. – М.:Лори. – 2011. – 288 с.

2.8. Во время подготовки данных для морфологического модуля лингвист *Linguist* взаимодействует с системой подготовки данных *MorphoDPS* с целью изменения данных *ModifyData*. Кроме того, для проверки целостности модифицируемых данных лингвисты могут компилировать данные *Compile*. Компиляция также включает в себя экспорт данных *ExportData* в формат, понимаемый компилятором. Каждую ночь сервер сборки приложения *BuildServer* компилирует данные *Compile*.

а. Добавьте в систему программиста *Programmer*, которому доступны те же возможности, что и лингвисту. Кроме того, он может экспортировать данные *ExportData* для отладки подсистемы компиляции данных.

б. Укажите, что для повторного использования словаря, который хранится на сервере данных морфологии, модуль семантики *Semantics* может взаимодействовать с системой подготовки данных морфологии в варианте использования *ExportWordList*.

в. Добавьте функции изменения данных: добавление, изменение и удаление слова.

г. Добавьте в модель возможность при изменении данных в некоторых случаях проверять целостность данных перед сохранением в систему.

д. Будет ли проверяться целостность данных при удалении слова? Ответ поясните.

2.9. Инкассатор *Cashier* и заправщик *Loader* занимаются обслуживанием автомата с газировкой. В обязанности инкассатора входит сбор денег *CollectCash*, а заправщик загружает в автомат баллоны с водой *ChangeWater* и газом *ChangeGas*.

а. Выделите в модели общий сценарий обслуживания, который включает авторизацию в системе обслуживания автомата и завершение сессии обслуживания.

б. Укажите, что автомат также может быть заправлен сиропом.

в. В каком случае инкассатор может загрузить в автомат баллон с водой? Ответ поясните.

г. Отразите в модели, что инкассатор может наблюдать за автоматом через Интернет с помощью встроенной видеокамеры с включением по сигналу датчика присутствия здания. Решение поясните.

2.10. Первоначальная модель загрузки *CreateDocument* и проверки документов *ReviewDocument* преподавателем *Professor* учебного заведения приведена на рис. 5.

а. Добавьте преподавателю возможность создавать курсы, как по шаблону, так и повторяя курс прошлого года.

б. Покажите на диаграмме, что у преподавателя есть три возможности проверки документа: с помощью мастера *GuidedReview*, совместно со студентом *JointReview*, и простое ревью *BasicReview*. При этом студенты сами могут загружать документы *Upload* и регистрироваться *Enroll* на курс.

в. Добавьте в модель ассистента преподавателя *TA* так, чтобы он обладал всеми обозначенными выше возможностями преподавателя, но не мог создавать курсы. При этом студент может быть ассистентом, но не преподавателем.

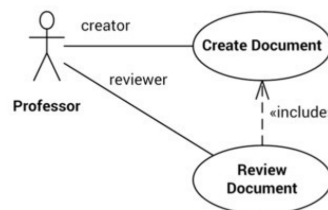


Рис. 5

§3. КООПЕРАЦИИ И ВЗАИМОДЕЙСТВИЯ КЛАССОВ

ОСНОВНЫЕ ПОНЯТИЯ

Структурированным классификатором (structured classifier) называется классификатор, который может включать соединители, связывающие содержащиеся в классификаторе свойства. Структурированные классификаторы определяют контекст для соединителей и позволяют описать связи между экземплярами, возникающие во время выполнения системы.

Соединители (connector) – это черты структурированного классификатора, имеющие тип и связывающие два или более свойств классификатора. В то время как связи (links) являются экземплярами ассоциаций, соединители ограничивают возможные связи между экземплярами в зависимости от контекстного классификатора, которому принадлежат эти экземпляры.

Частью (part) классификатора называется свойство, с которым классификатор связан отношением композиции.

Кооперация (collaboration) является структурированным классификатором, обладающим поведением, и определяет роли составляющих ее частей и взаимодействия между ними в контексте кооперации. Кооперации используются для определения взаимодействий, обеспечивающих достижение какой-либо цели или реализации функции системы.

Вхождение кооперации (collaboration use) связывает элементы модели с ролями, определенными в кооперации.

Ролями (role) в кооперации называют части кооперации, параметры поведения или локальные переменные в поведении кооперации.

Состоянием (state) экземпляра классификатора называют условие или ситуацию, во время которой его свойства удовлетворяют некоторому условию, он выполняет определенное собственное поведение или ожидает какого-либо события.

Поведение (behavior) классификатора описывает изменение состояния классификатора с течением времени в ответ на внешние события и внутренние вычисления. Поведение может быть исполняемым (executable) и производным (emergent). Исполняемое поведение является описанием процесса исполнения некоторого алгоритма экземпляром классификатора путем выполнения действий. Производное поведение возникает в результате взаимодействия нескольких экземпляров.

Суть **овеществления** (**reification)** заключается в представлении происходящего поведения в виде экземпляра класса поведения. Таким образом, выполнение поведения экземплярами классов отождествляется с созданием и уничтожением экземпляра класса этого поведения.

Событием (event) называется описание группы изменений, которые могут привести к модификации значений свойств экземпляров в модели или выполнению поведения.

Сигнал (signal) является специальным видом классификатора, который описывает асинхронные запросы, направляемые экземплярам классификаторов. Черта приема определенного типа сигнала (reception) указывает, что экземпляры принимающего активного класса обрабатывают направленные им сигналы данного типа.

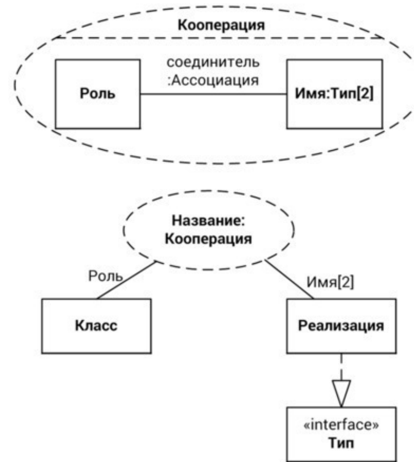


Рис. 6. Основная нотация коопераций

Траекторией (trace) называется частично упорядоченная последовательность возникновений событий (occurrence specification).

Взаимодействием (interaction) называется производное поведение участников, указывающее разрешенные и запрещенные траектории. Участникам сопоставлены **линии жизни (lifeline)**, на которых откладываются возникающие на траекториях события. Когда взаимодействие происходит в контексте динамического структурированного классификатора, линии жизни соответствуют ролям в этом классификаторе, локальным переменным данного взаимодействия или параметрам вызываемых операций и отправляемых сигналов. Если кратность участвующего во взаимодействии свойства, переменной или параметра больше единицы, то для соотнесения линии жизни с определенным значением из нескольких используются **селекторы** (**selector**).

Сообщения (message), передаваемые в процессе взаимодействия, могут быть нескольких сортов: синхронный и асинхронный вызов операции, асинхронная отправка сигнала, создание и уничтожение экземпляра, и ответные (reply) сообщения. Передача сообщения между линиями жизни отмечается возникновением событий отправки и получения сообщения. Если отправитель или получатель находится вне взаимодействия, вместо него подставляется шлюз (gate).

Фрагмент взаимодействия (interaction fragment) является частью взаимодействия и включает множества разрешенных и запрещенных подпоследовательностей возникновений событий для всех или некоторых линий жизни.

Операторы взаимодействия (interaction operator) используются для изменения траекторий комбинированного фрагмента взаимодействия, состоящего из нескольких фрагментов. Определены операторы альтернативного выбора (alt), цикла (loop), параллельного возникновения событий фрагментов (par), условного выполнения (opt) и другие.

Спецификация исполнения (execution specification), отложенная на линии жизни, указывает на выполнение экземпляром классификатора соответствующего данной линии некоторого исполняемого поведения.

Вхождение взаимодействия (interaction use) служит для повторного использования взаимодействий, вместо фрагмента подставляется содержимое указанного взаимодействия.

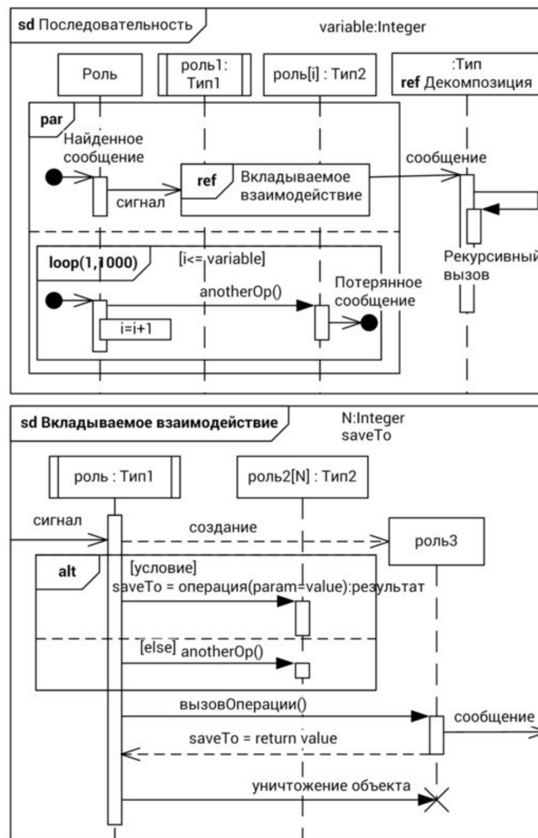


Рис. 7. Основная нотация диаграмм последовательности

ЗАДАЧИ

3.1. (см. решение в §11) Кооперация продажа *Sale* включает роли продавец *Salesman* и покупатель *Customer*.

а. Покажите, что продавец и покупатель могут взаимодействовать друг с другом.

б. Используя вхождения коопераций *Sale*, создайте модель кооперации продажи с посредником *BrokeredSale*, в которой покупатель взаимодействует с посредником *Broker*, а посредник как покупатель взаимодействует с продавцом.

3.2. Моделируется серверная часть веб-приложения интернет-магазина, построенного на основе паттерна Model-View-Controller (MVC). Взаимодействие между ролями *Model*, *View* и *Controller* отобразим на диаграмме последовательности.

а. Разместите на диаграмме роли *Controller*, *Model*, а также роль типа *ORM* с именем *db*. Синхронное найденное сообщение *postBuy (purchase)* приходит на линию жизни *Controller*. После этого *Controller* посылает *Model* синхронное сообщение *addPurchase (purchase)*. В ответном сообщении *Model* возвращает объект *purchaseDetails*.

б. Реализуем на диаграмме поведение *Model* в ответ на сообщение *addPurchase*. *Model* посылает синхронное сообщение *addPurchase (purchase)* линии жизни *db*. Затем открывается фрагмент *alt*. При условии *purchase.needDelivery Model* посылает сообщение *addDelivery(purchase.address)* линии жизни *db*. Фрагмент *alt* окончен. *Model* посылает *db* синхронное сообщение *saveChanges ()*.

в. В ответ на запрос покупки *Controller* должен сообщить пользователю, что заказ совершен успешно. После получения от *Model* ответного сообщения *Controller* создает новую линию жизни с ролью *View* сообщением *createConfirmationView*. Затем *Controller* посылает линии

жизни *View* сообщение *setPurchaseDetails (purchaseDetails)*. Затем *Controller* отправляет в ответ на входящий запрос ответное сообщение, содержащее *View*.

г. Отдел доставки нужно уведомить о том, что требуется доставить новый заказ. Добавим в *Model* (в опциональный фрагмент *alt*) посылку асинхронного сообщения *notifyNewDelivery*. Сообщение является потерянным.

д. Код интернет-магазина достаточно универсальный. Можно сделать на базе этого кода несколько сайтов для разных магазинов. Для этого нужно заполнить *View* при создании информацией о конкретном магазине. Добавим в контекст взаимодействия переменную *shopInfo* типа *ShopInformation*. После сообщения *setPurchaseDetails* начинается фрагмент использования взаимодействия *ref* с именем *FillShopInformation*. В этот фрагмент входят линии жизни *Controller* и *View*. В виде аргумента во взаимодействие передается переменная *shopInfo*.

3.3. (см. решение в §1) Автор *Author* направляет статью сообщением *manuscript* редактору *Editor* и ожидает от него подтверждения получения. Редактор отправляет сообщением *evaluate* статью рецензенту *Peer*. Рецензент отправляет сообщение редактору с оценкой статьи *review*. Редактор направляет сообщение автору с результатами *resolution* и рецензенту с благодарностью *thanks*.

а. Восстановите структурную модель взаимодействия в виде кооперации *ReviewManuscript*, укажите кратность роли рецензента так, чтобы статья направлялась на рецензию одному из пяти рецензентов.

б. Укажите, используя фрагменты, что статья направляется на рецензирование каким-либо трем из пяти рецензентов.

в. Используя фрагменты, покажите, что порядок отправки результатов рецензирования автору и благодарностей рецензентам не имеет значения.

3.4. Терапевт *Therapist* ведет прием посетителей *Person*, в ходе которого выписывает лекарства *Medicine*; посетители принимают лекарства.

а. Постройте логическую модель, включающую классы *Therapist*, *Medicine* и *Person*, и отношения между ними.

б. Используя кооперации, покажите, что на приеме терапевт выполняет обязанности врача *Doctor*, посетитель является пациентом *Patient*, лекарства выписываются в виде рецептов *Subscription*.

3.5. Автомобиль *Car* состоит из двигателя класса *Engine*, пары передних *front* и задних *rear* колес класса *Wheel*.

а. Добавьте привод *drivetrain* так, чтобы автомобиль был переднеприводным.

б. Расширьте модель так, чтобы наряду с переднеприводными автомобилями, она описывала полноприводные автомобили как частный случай переднеприводных. Добавьте необходимые элементы, используйте двигатель *DoubleEngine* с двумя приводами типа *drivetrain*.

3.6. Пассажир *Person* заходит в лифт и нажимает кнопку *pressButton* лифта *Lift* с указанием целочисленного номера этажа *floor*. Лифт закрывает двери и начинает движение синхронным вызовом операции *startMoving*. После этого сообщает пассажиру номера проезжаемых лифтом этажей сообщением *floorReached* с указанием номера этажа. Затем лифт вызывает операцию *stopMoving* и останавливается. Пассажир нажимает кнопку *pressDoors* лифта для открытия дверей.

а. Как можно уточнить модель взаимодействия, если известно, что лифт обслуживает с первого по пятый этажи?

б. Уточните взаимодействие пассажира с лифтом. Укажите, что до нажатия кнопки этажа, пассажир обязан закрыть двери кнопкой *pressDoors*.

в. Используя фрагменты, покажите, что пассажир не может нажать кнопку открытия и закрытия дверей в процессе движения лифта.

г. (*) Укажите, что лифт проезжает один этаж за три секунды.

3.7. Менеджер подключаемых модулей *pluginsManager* класса *PluginsManager* получает сообщение *loadPlugins* – указание на необходимость загрузки доступных модулей. Он синхронно запрашивает у объекта *settings* класса *PluginManagerSettings* пути к директориям с модулями и получает от *settings* значение свойства *pluginsDirs*. После чего в цикле для каждой директории и каждой библиотеки **.dll* загружает модули вызовом собственной операции *loadPlugins*, передавая путь к библиотеке в параметрах.

а. Реализуйте операцию *loadPlugins* класса *PluginsManager*. Взаимодействие начинается с создания нового экземпляра *PluginsDll*, затем идет получение количества подключаемых модулей в библиотеке *getPluginsCount* и получение всех модулей через вызовы *getPlugin* с параметром – номером модуля. После этого происходит инициализация каждого полученного модуля *IPlugin* вызовом метода *initPlugin* класса *PluginsManager*.

б. Добавьте в модель описание действий по инициализации модуля. Метод *initPlugin* проверяет, обрабатывает ли модуль события графического интерфейса вызовом *isUIHandled*. Если обрабатывает, то регистрирует модуль в качестве слушателя событий *addListener* в классе *PlayerUIPresenter*.

3.8. Взаимодействие выбора этажа *SelectFloor* содержит линию жизни кабины, представленной экземпляром активного класса *Cabin*, линию жизни *floor* экземпляра класса кнопки этажа *FloorButton* с селектором *«1»*, и линию жизни класса *Algorithm*. Взаимодействие начинается с синхронного вызова кабиной операции нажатия кнопки *isPressed* на линии *floor*. Операция возвращает логическое значение «истина», если кнопка нажата. Затем экземпляр класса *Cabin* вызывает операцию *selectFloor* у алгоритма на линии в данном взаимодействии.

а. Используя оператор цикла, покажите, что проверяется нажатие кнопок всех этажей. Переменную цикла, содержащую номер кнопки этажа, объявите как атрибут взаимодействия.

б. Уточните взаимодействие, добавив вызовы операции указания алгоритму этажей, кнопки которых нажаты.

в. Пусть взаимодействие описывает поведение кооперации, линии жизни соответствуют ролям в этой кооперации. Каким образом необходимо изменить взаимодействие, если кооперация владеет ролью с линией жизни алгоритма?

г. (*) Приведите по одному примеру разрешенной и неразрешенной траекторий в данном взаимодействии.

3.9. Пользовательский интерфейс мультимедиа проигрывателя *PlayerView* сообщает *ev* контроллеру *UIPresenter* о начале проигрывания элемента списка воспроизведения *playEntry*. Контроллер: 1) рассылает это сообщение всем своим слушателям собственной операцией *raiseEvent*; 2) получает от управляющего компонента *Engine* тип контента по имени *name* выбранного элемента *item* события *ev* вызовом *getContentType*; 3) по типу контента получает медиапоток вызовами *getAudioStream* или *getVideoStream*; 4) переводит интерфейс в состояние проигрывания либо видео, либо аудио вызовом *setPlaybackMode* и указывает медиапоток операцией *setMediaStream*; 5) запускает воспроизведение операцией *play* контроллера.

а. Добавьте в модель возможность параллельной обработки сообщений в *raiseEvent* так, что операция обработки *handleEvent* вызывается асинхронно у каждого зарегистрированного слушателя типа *IUIListener*. Для отображения в модели используйте два анонимных слушателя.

б. Уточните взаимодействие при обработке события запуска проигрывания песни модулем *lyricsPlugin*, зарегистрированным в качестве слушателя событий *UIPresenter*. Модуль определяет тип контента элемента, передаваемый в сообщении о запуске. Для аудиоконтента создает запрос о тексте песни, вызывая собственный метод *makeRequest*. Затем модуль асинхронно вызывает операцию *dispatchRequest* класса *NetController*, передавая в параметрах запрос и себя как обработчика ответа. Получив ответ от сервера, *NetController* передает *processResponse* его обработчику. Модуль вызывает *displayPage* у *UIPresenter* и передает полученную HTML-страницу с текстом песни

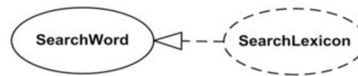


Рис. 8

3.10. (*) При решении основных задач, морфологический модуль ищет запрашиваемое слово в словаре языка. Таким образом, многие задачи в модуле зависят от реализации функции поиска слова в словаре. См. диаграмму на рис. 8. Для уменьшения объема, который занимает словарь, используется тип данных префиксное дерево[2]. Каждый узел дерева содержит ассоциативный массив буква – следующий узел. Таким образом, если слово есть в словаре, то в дереве есть путь, начинающийся в корне и проходящий по вершинам, соответствующим буквам слова.

а. Постройте модель кооперации поиска слова в словаре, используя роли класса *Dictionary* и узла дерева *Node*. Для моделирования префиксного дерева используйте квалифицированные соединители и булевый атрибут *leaf*, указывающий на конечный узел слова.

б. Добавьте в модель поведение кооперации. Актор парсер строки *StringParser* вызывает операцию *hasWord* класса *Dictionary* с параметром *word* типа *String*. Метод *hasWord*, реализующий операцию *hasWord*, получает корневой узел с помощью операции *getRoot* класса *Dictionary*. Получив экземпляр класса *Node*, метод в цикле для каждой следующей буквы слова *word* вызывает у этого экземпляра операцию *getNextLetter* с параметром *c* типа *char* – текущей буквы слова. Данная операция возвращает дочерний узел дерева. Когда буквы слова *word* закончились, нужно вернуть актору значение операции *isLeaf* последнего полученного узла *Node*.

в. Модифицируйте поведение. Если в какой-то момент вызов *getNextLetter* прерван по исключению *NoSuchLetter*, операция *hasWord* должна вернуть *false*.

3.11. Вариант использования просмотр каталога *SearchCatalog* реализован кооперацией *GetAllRecords*. Основной сценарий варианта использования начинается с получения контроллером приложения *AC* команды *showRecords*. *AC* отображает *show* в пользовательском интерфейсе *UI* сообщение «Идет запрос». *AC* параллельно отправляет источнику данных *DataSource* запрос *readRecords*. *DataSource* передает *AC* одну запись в параметре действия *acceptRecord*. Затем *AC* показывает запись в *UI*.

а. Укажите, что перед запросом записи, *AC* запрашивает *getListSize* количество записей у источника данных. Результат присваивается переменной *listSize*.

б. Измените модель так, чтобы источник данных передавал контроллеру *listSize* записей по одной, а контроллер отображал *show* в *UI* все полученные записи вместе.

в. Реализуйте альтернативный сценарий, когда источник данных не содержит записей.

г. Перечислите все необходимые соединители в кооперации *GetAllRecords*, укажите, какие сообщения по ним передаются. Ответ поясните.

д. (*) Какое минимальное количество экземпляров классов необходимо, чтобы выполнить описанное поведение? Ответ поясните.

[2] Префиксное дерево, «Trie», http://en.wikipedia.org/wiki/Trie.

ГЛАВА 2. МЕТОДЫ И ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Парадигмы проектирования. Возникновение проектирования программного обеспечения можно связать с появлением языков высокого уровня в 60-х и 70-х годах. Проектирование возникло как дисциплина, целью которой было управление сложностью программных систем и расширение возможностей разработчиков по созданию систем большего размера предсказуемого качества.

В развитии проектирования как дисциплины выделяют несколько этапов, в каждом из которых доминировала одна из парадигм проектирования. В 70-е и 80-е такой была структурная парадигма проектирования. Ее основу составляют нисходящие декомпозиционные методы, итеративно разделяющие систему на функциональные блоки, все более понятные и простые в реализации. Примерами таких методов могут служить метод постепенного уточнения (stepwise refinement) [6], метод структурного анализа и структурного проектирования SSA/SD [6], техника структурного анализа и проектирования SADT[3]. Среди восходящих методов структурного проектирования следует отметить метод структурного проектирования Джексона [6].

Преимущественные нотации моделей в этой парадигме – это структурная схема, схема потоков данных, диаграмма сущность-связь, диаграммы IDEF.

В 80-е и 90-е преобладающими стали методы объектно-ориентированного проектирования. Основой методов является выделение общего описания групп объектов и использование этого описания в качестве абстрактного типа данных. Различные эвристики выделения общего описания и процедуры создания модели нашли выражение в различных методах анализа и проектирования. В результате объединения Objectory, OMT и OOAD был создан унифицированный процесс разработки RUP и язык моделирования UML. Во многом благодаря обсуждению принципов проектирования на страницах таких журналов как C++ Report были сформулированы эвристики повышения изменяемости и сопровождаемости систем SOLID [7]. Следует также отметить методы проектирования, основанные на выделении и группировке обязанностей RDD и связанные с ними эвристики назначения обязанностей GRASP [8]. А также методы, построенные на прямом использовании модели предметной области для построения программной системы. Позднее они нашли отражение в книге Эванса по предметно-ориентированному проектированию [5].

Сейчас объектно-ориентированные методы применяются для разработки отдельных компонент сложных систем. Методы структурного проектирования нашли применение при разработке систем обработки данных, в проектировании архитектуры систем масштаба предприятия.

Систематизация и количественный подход. Существование разных подходов к проектированию в одних и тех же отраслях ведет к необходимости их сравнения и определения предпочтительного и указания границ применимости. Инициатива по выработке базовых методов и теории программной инженерии SEMAT ставит своей целью каталогизацию методов, выработке их описания на основе нескольких базовых понятий и, таким образом, создания основ накопления данных об использовании методов для их последующего сравнения.

Распространенный подход к накоплению знаний в области проектирования – составление каталога повторно применимых приемов решения часто встречающихся задач проекти-

[3] Дэвид А. Марка, Клемент Л. МакГоуэн. Методология структурного анализа и проектирования: [Пер. с англ.] / Предисл. Д. Т. Росса. – М.: Фирма «Мета Технология». – 1993. – 240 с.; ил.

рования, которое могут быть адаптированы для разных случаев – паттернов проектирования. В конце прошлого века были составлены первые каталоги паттернов. Наиболее известным является набор из двадцати двух паттернов «банды четырех» GoF [3]. В сфере высокоуровневого (архитектурного) проектирования аналогом паттернов выступают архитектурные стили [9], комбинации которых, исходя из требований к системе, составляют первоначальное архитектурное описание.

Результатом процесса проектирования являются отраженные в модели решения по реализации программной системы. Для прогнозирования нефункциональных характеристик системы, в том числе сопровождаемости, переносимости и других, вводят показатели проектировочного решения (метрики дизайна). Показатели используют для количественного описания размера системы, сложности решения, выявления недостатков и потенциально проблемных мест в системе. В данной книге рассматриваются показатели сложности проектировочного решения Чидамбера-Кемерера [10].

Применение методов. До автоматического проектирования программных систем пока еще далеко, в проектировании остается существенная доля искусства. В том числе вследствие самой сути задач проектирования, относящихся к так называемым плохо поставленным задачам (wicked problems), условия которых неполны, противоречивы, меняются со временем и в процессе решения. Тем не менее, владение базовыми методами позволяет не отвлекаться на обдумывание задач, решение которых уже известно, и, таким образом, повысить качества результата и сократить время его создания.

Рассматриваемые в сборнике методы проектирования отражают современное состояние практики разработки и проектирования на уровне компонентов и приложений. Их изучение позволит как вступить в специальность проектирования, восполнить отдельные пробелы, так и по-новому взглянуть на уже известную область.

Принципы проектирования классов и интерфейсов SOLID. Аббревиатура SOLID расшифровывается по первым буквам сокращений названий принципов проектирования классов.

Single responsibility principle (SRP), принцип ограничения обязанностей, говорит, что модуль или класс должен иметь только один набор функционально сходных обязанностей.

Open-closed principle (OCP), принцип открытости-закрытости, указывает, что класс или модуль должен быть расширяем (открыт для расширения) без внесения в него изменений (закрыт для изменения).

Liskov substitution principle (LSP), принцип подстановки (описан в статье Барбары Лисков, отсюда название), указывает правило построения иерархии типов так, что любой подтип или дочерний класс подставим вместо базового типа или класса соответственно.

Interface segregation principle (ISP), принцип разделения интерфейса, говорит, что для обозначения разных ролей, которые играет класс в разных взаимодействиях, следует использовать разные интерфейсы.

Dependency inversion principle (DIP), принцип обращения зависимостей, указывает на корректное применение принципа сокрытия информации в объектно-ориентированном подходе к проектированию, когда зависимости направлены от реализации класса к выделенным абстракциям: описаниям типов данных или интерфейса

§4. РАСШИРЕННЫЕ КЛАССЫ И ОБЪЕКТЫ

ОСНОВНЫЕ ПОНЯТИЯ

Квалификатор (qualifier) используется для разделения всех связей ассоциации на подмножества согласно уникальным ключам. Обычно в бинарной ассоциации «один-ко-многим» по ключу выделяют связи «один-к-одному».

Класс ассоциации (association class) используют, когда логическое отношение между объектами обладает сложной структурой или поведением. Класс ассоциации является одновременно и ассоциацией, и классом, поэтому может иметь свойства и операции.

Напомним, что операция – это поведенческая черта класса, которая определяется именем, набором параметров, их типами и кратностями, типом и кратностью возвращаемого значения. В дополнение к этому, можно задать **ограничения** на реализацию операции: **предусловие (precondition)**, **постусловие (postcondition)**, **ограничение возвращаемого значение (body condition)**.

Если операция не изменяет значения свойств класса, то к операции добавляется украшение **запрос** (*query*).

Каждый параметр операции может иметь имя, тип, множественность. Параметру можно задать **направление параметра** (*in*, *out*, *inout*, *return*), значение по-умолчанию, ограничение на множественный параметр: **упорядоченность** значений *(ordered*), **уникальность** значений (*unique*).

Производные свойства класса (derived property) вычисляются на основе других свойств или результатов вызова операций класса или других классов модели. Способ вычисления значений свойства указывается вместе с определением самого свойства, при этом вычисление значений не должно иметь сторонних эффектов и должно быть идемпотентным (повторные вычисления дают тот же результат при неизменности остальной модели). Часто используются производные свойства, значения которых составлены из объединения **union** множеств значений свойств, выделяющих подмножества **subsets** в базовых свойствах.

Шаблонные классы (template class) по сути не являются полноценными классами, а только их заготовками, определенными с точностью до значений параметров шаблона. Шаблонным может быть не только класс, но и другие элементы модели. Операция присвоения значения параметру называется **связыванием (bind)**. По смыслу, связывание класса с шаблонным классом с приданием значения параметру аналогично созданию класса из шаблона с указанным значением параметра и наследованием от этого класса.

Переопределение (derived) позволяет заменить определение черты классификатора в рамках его **контекста переопределения (redefinition context)**, составляющего совокупность всех классов, обобщающих данный. При переопределении можно заменить, например, сигнатуру операции. После переопределения при обращении к новой операции в классе следует использовать новую сигнатуру. При этом наследованная переопределенная операция также доступна для использования.

Множество обобщения (generalization set) позволяет логически группировать отношения обобщения. Можно указать, что в данном множестве обобщений приведены все возможные уточнения базового класса, или что совмещение непосредственно уточняющих классов в одном экземпляре или подклассе не допускается.

Супертип (powertype) используется совместно с множеством обобщений. Супертип это классификатор, экземплярами которого являются классы-элементы множества обобщений.

Пакеты (package) позволяют группировать элементы модели под общим именем. Для обращения к элементу пакета необходимо использовать квалифицированное имя, состоящее из имени пакета и имени элемента.

Между пакетами определены отношения **доступа** *«access»*, которое для элементов пакета делает доступным элементы указанного пакета без необходимости указания квалифицированного имени, и отношение **импорта** *«import»*, которое аналогично доступу, но делает элементы также доступными при последующем импорте или доступе из другого пакета.

Отношение **объединения пакетов (merge)** *«merge»* позволяет объединить в одном элементе определения этого элемента в других пакетах.

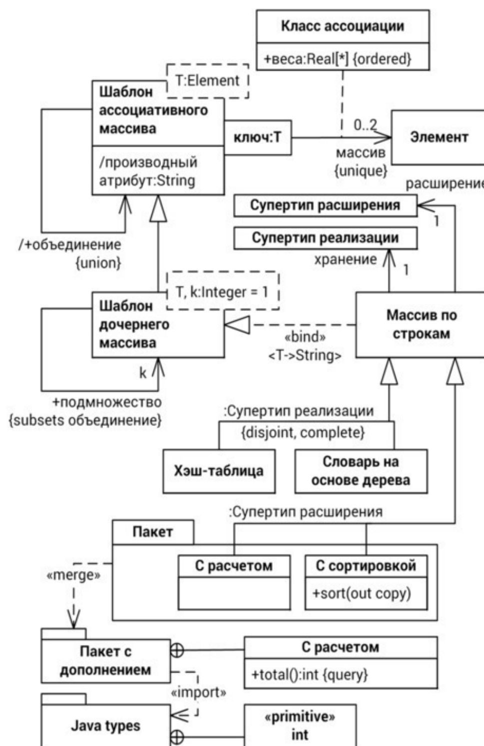


Рис. 9. Расширенная нотация диаграмм классов

Сигналом (signal) называют особый вид классификатора, экземпляром которого является сообшение, передаваемое асинхронно отправителем получателю или группе получателей. Для того, чтобы обрабатывать сигналы, получатель должен быть **активным классом (active class)**, объявлять черту поведения **– получение сигнала (reception)**, с которой может быть связан метод, либо определять собственное поведение, которое обрабатывает поступающие сигналы.

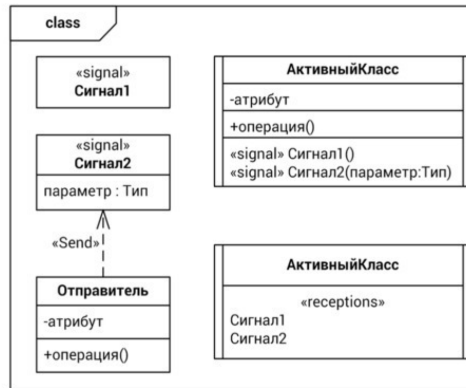


Рис. 10. Нотация сигналов

ЗАДАЧИ

4.1. На рис. 11 представлены шаблонные интерфейсы *Map* и *Entry*. Интерфейс *Map* позволяет по ключу типа *K* получить значение типа *V*. Интерфейс *Entry* представляет собой пару значений.

а. Измените модель так, чтобы шаблон *Entry* использовал параметры шаблона *Map*.

б. Определите интерфейс *Map_StringInteger*, который указывает *String* типом ключа и *Integer* типом значения в шаблоне *Map*.

в. Сколько операций содержит интерфейс *Map_StringInteger*? Ответ поясните.

4.2. Диск *Disk* содержит несколько папок *Folder*, которые могут содержать файлы *File* и папки. Произведения *Composition* хранятся на дисках в виде файлов.

а. Используя классы ассоциаций, постройте модель хранения произведений на дисках.

б. Дополните модель, укажите, что произведение может быть картинкой *Picture*, либо музыкой *Music*, либо фильмом *Movie*.

в. Может ли произведение храниться на одном диске в разных файлах? Ответ поясните.

г. (*) Сравните способы реализации в модели хранения произведения в нескольких файлах на одном диске. Приведите примеры на диаграмме экземпляров.

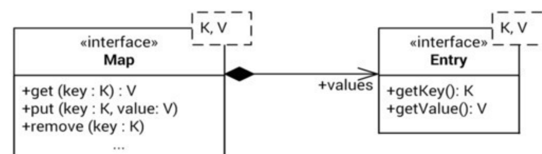


Рис. 11

4.3. На заседании *Meeting* обсуждается *discuss* не менее одного вопроса *Issue*. Вопрос может быть посвящен обсуждению артефакта *Artifact*. В каждом вопросе должно быть указано текстовое название, числовой код и имя автора.

а. Добавьте в модель вопрос по постановлению, отдельный вопрос и сложный вопрос.

б. Укажите, что постановление *Resolution* связано с вопросом по постановлению, как тема *topic*, и с несколькими артефактами *documents*.

в. К сложному вопросу примените паттерн Composite так, чтобы сложный вопрос включал несколько других вопросов.

г. Приведите пример, когда на заседании обсуждается сложный вопрос, включающий вопрос по постановлению *claim* и отдельный вопрос, а также связанные с постановлением документы *mail* и *agenda*.

4.4. Интерфейс работы с ассоциативным массивом *Map* в своем пространстве имен содержит интерфейс работы с элементом массива *Entry*. При этом реализации интерфейса *Map* включают несколько реализаций интерфейса *Entry*.

а. Добавьте в модель класс *HashMap*, реализующий ассоциативный массив с помощью хэш-таблицы, и класс *HashEntry*, реализующий интерфейс работы с элементом массива.

б. Пусть в классе *HashMap* определена частная операция изменения размера *resize*. При каких условиях данная операция будет доступна классу *HashEntry*?

в. Используя шаблоны, параметризуйте интерфейсы ассоциативного массива и его элемента, укажите, что тип ключа и тип значения в них совпадают.

4.5. Класс *CPluginsSettings* хранит пути к директориям, в которых могут находиться подключаемые модули. Класс управления подключаемыми модулями *CPluginManager* связан с *CPluginsSettings*. Модули входят в состав динамической библиотеки, содержащей доступные извне функцию получения количества подключаемых модулей *GetPluginsCount* и функцию *GetPlugin* с параметром типа *Integer*, которая возвращает экземпляр модуля *IPlugin*.

а. Работа с динамической библиотекой состоит из трех шагов: загрузка библиотеки, получение функции по ее имени, вызов функции. Добавьте в модель класс динамической библиотеки *CDynamicLibrary*, с операцией получения указателя на библиотечную функцию по ее имени *GetFunction*.

б. Одними из самых популярных считаются модули, добавляющие в плеер возможность построения визуального ряда для композиций и отображения текста песен. Добавьте в модель соответствующие модули *CVisualizationPlugin* и *CSongLyricsPlugin*.

в. Известно, что библиотека подключаемых модулей предоставляет неизменный набор функций, поэтому можно применить паттерн Wrapper к API библиотеки, реализуемый с помощью класса *CPluginDll*. Реализуйте и сравните два подхода: через уточнение класса *CDynamicLibrary*, и с помощью агрегации.

4.6. В системе из задачи 4.5 реализована модель обработки сообщений с помощью паттерна Observer: событие представлено классом *Event*. источник событий представлен абстрактным классом *EventSource*, обработчик событий – интерфейсом *IEventListener*. Класс *EventSource* определяет общедоступную операцию добавления обработчиков и защищенную операцию возбуждения событий, которая вызывается в дочерних классах и приводит к рассылке события всем обработчикам.

а. Добавьте в модель источники событий управляющего компонента *EngineSource* и представления *UISource*.

б. Укажите, что добавленные источники создают события, специфичные этим источникам, и поэтому принимают только специализированные обработчики *IEngineListener* и *IUIListener* соответственно.

в. Каждый модуль (см. задачу 4.5) может обрабатывать события управления и представления. Добавьте класс *PluginEventHandlers*, содержащий *IEngineListener* и *IUIListener*. При этом нужен доступ к обработчикам событий в конкретном экземпляре модуля.

4.7. В классе *Object* определена операция сравнения *equals* с параметром *obj* типа *Object* и определен метод (behavior), реализующий данную операцию. Дочерний класс *Rectangle* перекрывает и реализует операцию *equals*. Другой дочерний класс *Clickable* класса *Object* также перекрывает и реализует операцию *equals*.

а. Пусть экземпляр класса *Rectangle* присвоен переменной *var* класса *Object*. Какая из реализаций будет выполнена при вызове операции *equals* у переменной *var* с параметром *var*? Ответ поясните.

б. (*) Добавьте в модель класс *Button* дочерним к *Rectangle* и *Clickable*. Используя механизм переопределения (redefinition), добавьте в класс *Button* операцию *equals*, реализация кото-

рой использует наследованные операции сравнения в зависимости от типа параметра *obj* времени выполнения. Приведите реализацию операции на подходящем языке программирования.

4.8. Массив *Table* состоит из нескольких элементов *Element*.

а. Укажите, что элементы упорядочены и могут повторяться.

б. Отразите в модели, что имея экземпляр массива, можно перейти к его элементам, но от экземпляра элемента нельзя перейти к массиву.

в. Пусть массив проиндексирован таким образом, что индексом элемента также является элемент. Используя квалификаторы, отразите данное свойство в модели.

4.9. Преподаватель *Teacher* ведет *teaches* несколько курсов *CourseOffering*.

а. Используя агрегацию, покажите, что курс состоит из одной лекции *Lecture* и нескольких семинаров *Practice*.

б. Укажите, что преподаватель ведет семинары как ассистент *assistant* и читает лекции как лектор *lecturer*.

в. (*) Измените свойство *teaches* так, чтобы оно всегда указывало только на курсы, по которым преподаватель читает лекции или ведет семинары.

4.10. Каждый экземпляр абстрактного класса контроллер *Controller* связан по ассоциации *Sensor* с несколькими датчиками поезда *TrainSensor*. В ассоциации контроллер играет роль управляющего *controller*. Датчик поезда участвует в ассоциации как датчик *sensor* с частной видимостью.

а. Используя квалификаторы, укажите, что каждому значению индекса *index* типа *String* соответствует не более одного датчика в ассоциации *Sensor*.

б. Измените класс контроллера, укажите, что класс принимает сигналы приближения поезда *TrainSpotted* и отдаления поезда *TrainLeft*, имеет общедоступную операцию выполнения команд *execute* с параметром команда *cmd* типа данных *Command* и возвращает значение типа данных *Result*.

в. Определите класс цифрового контроллера *DigitalController*, уточняющий класс контроллера. В классе цифрового контроллера определена операция *executeDigital*, которая переопределяет операцию выполнения команд контроллера и возвращает цифровой результат *DigitalResult*.

г. Используя экземпляры классов, приведите пример контроллера с двумя датчиками.

4.11. Игрок *Player* заключает контракт *Contract* с командой *Team*. Команда может заключить до двадцати контрактов с разными игроками. Контракт заключается на определенный срок *period* с компенсацией *salary* (класс ассоциации). В контракте игрок указан работником *worker*, команда – нанимателем *employer*.

а. Добавьте в контракт пункты *Item*, каждый из которых содержит текст *statement*. Укажите, что менеджер *Manager* управляет *manages* контрактами.

б. Укажите, что игрок может быть нападающим *Forward*, защитником *Guard* или центровым *Center*. Игрок не может иметь несколько специализаций.

в. Игроки реагируют на команды тренера *Coach*. Во время игры, тренер может отправлять им указания: нападать *attack*, перейти к обороне *guard*, играть совместно *join* с другим игроком *peer*. Работа тренера состоит во взаимодействии и обучении *Train* команды.

г. Измените модель таким образом, чтобы игроки могли изменять специализацию: нападающий, защитник или центровой. Решение поясните.

4.12. (*) В файловой системе диска *Disk* данные сохраняются в кластерах *Cluster*, информация о связанных цепочках кластеров записана в таблице размещения файлов *FAT*. Таблица содержит записи *FAT Record*, номер записи в таблице соответствует номеру *cl_no* соответствующего кластера при этом номер кластера *cl_no* на диске. Содержимое записи указывает либо на следующий кластер, либо на конец цепочки *EOF*. Кластер директории *Folder* содержит *entries*

упорядоченный набор отметок *Entry* о файлах и директориях. Каждая из записей указывает имя *name* и начальный кластер *first*.

а. Постройте модель файловой системы, используя квалифицированные ассоциации. *Указание*. Следующий кластер в цепочке вычислять с помощью производных атрибутов.

б. Добавьте кластер данных *File*, содержащий данные файлов. Операция *getData* позволяет получить данные файла в виде массива с элементами типа *Byte* с диапазоном значений от 0 до 255.

в. Добавьте резервные кластеры *Reserved*, укажите, что отметка об испорченности *Bad*, пустоте *Empty* кластера хранится в *FAT Record*.

г. Приведите пример диска с таблицей размещения файлов на пять кластеров с одной директорией *folder* и двумя файлами *A* и *B*. Файл с именем *B* размещен в кластере с номером *4*, файл *A* в кластерах *2* и *3*. Таблица размещается в нулевом кластере, корневая директория в первом.

4.13. Процессом рецензирования *Review* управляет *Manages* пользователь *User* в роли редактора *editor*. Процесс включает разговор *Conversation* с пользователями автором *author* и разговоры с пользователям-рецензентами *peers*. Разговор состоит *msgs* из сообщений *Message*. Каждое сообщение связано с одним пользователем – отправителем *sender* сообщения, и несколькими получателями *recipients*. К сообщению может быть прикреплен документ *Document* в виде вложения *attachment*.

а. Укажите, что множество участников *participants* разговора включает всех получателей и отправителей сообщений.

б. Пусть процесс рецензирования связан с рукописью *manuscript*, которая является документом. Уточните модель, укажите, что рукопись получена как вложение в одном из сообщений разговора с автором.

в. Перечислите все экземпляры классов, которые необходимы для создания экземпляра процесса рецензирования.

4.14. Кабина лифта *Cabin* содержит кнопки *Button* и датчики *Sensor*. В классе *Button* имеется операция определения нажатия *isPressed*, которая возвращает логическое значение. Кнопки бывают двух видов: служебные кнопки *UtilityButton* и кнопки этажей *FloorButton*. В классе *FloorButton* определена операция сброса *reset*. Известно, что класс *Cabin* реализует собственное поведение, а класс *Sensor* реализует паттерн Template Method.

а. Укажите, что кабина содержит девять упорядоченных по номеру разных кнопок этажей.

б. Используя только имеющиеся классы, укажите, что в кабине имеется служебная кнопка вызова диспетчера. Кроме этого, в кабине может быть либо служебная кнопка остановки, либо служебная кнопка открытия дверей.

в. Покажите, что датчики бывают только трех видов: датчик задымления, датчик закрытия дверей и датчик перегрузки лифта. Датчики существенно отличаются, поэтому в одном классе не могут быть совмещены реализации всех видов датчиков.

г. Какое минимальное количество экземпляров необходимо, чтобы представить кабину с датчиками задымления и закрытия дверей?

4.15. Рассмотрим модель, которая описывает графы вычислений. Действия *Action* и ребра *Edge* являются элементами *Element* модели. Ребро указывает направление вычисления и может иметь начальное действие *from* и конечное *to*.

а. Добавьте в модель симметричные действия сложения и умножения и одноместные действия обращения и получения противоположного элемента. Эти действия принимают и возвращают значения по ребрам *Edge*. Класс *Value* уточняет *Edge* и имеет атрибут *val*, но не имеет источника *from*, откуда получить значение.

б. Постройте модель реализации отображения графа вычислений в графическом пользовательском интерфейсе, где действия показаны прямоугольниками, а ребра – стрелками. Следуйте принципу SRP.

в. Приведите пример графа вычислений выражения 4 * (a + b). Введите необходимые типы значений.

4.16. В графическом редакторе использованы фигуры: отрезок *Line* между начальной и конечной точками *Point* и дуга *Arc* второго порядка, построенная по упорядоченному набору из трех точек. Фигуры позволяют задать свое положение точками и умеют рисовать *paint* себя на холсте *Canvas*.

а. Определите в подходящем существующем или новом классе статические операции создания отрезков и дуг по передаваемым точкам, и операцию создания точек по координатам на плоскости. Примените паттерн Factory Method.

б. Добавьте в модель фигуру ломаная *Polyline*, состояние которой задается набором соединенных последовательно отрезков.

в. Предложите альтернативную реализацию ломаной *Polyline*, состояние которой определяется последовательностью точек. Как в этом случае воспользоваться алгоритмом рисования отрезков? Сравните решение с пунктом б.

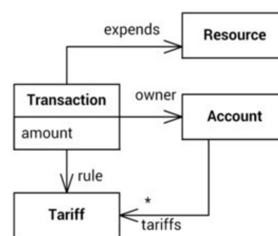


Рис. 12

г. Для замкнутых ломаных без самопересечений *Polygon* определите операцию заливки *fill* внутренней области одним из трех цветов *RGB*: красным, зеленым или синим. Примените паттерн Decorator. В каком классе следует реализовать проверку свойств замкнутости и отсутствия самопересечений ломаных?

4.17. Заготовка модели биллинга приведена на рис. 12. Экземпляры транзакций *Transaction* формируются, когда пользователь с учетной записью *Account* совершает действие по расходованию ресурса *Resource* согласно тарифу *Tariff*.

а. Покажите на диаграмме, что учетные записи пользователей бывают временные *Temporary* и постоянные *Permanent* с одной стороны, и корпоративные *Corporate* и индивидуальные *Personal* с другой. Для временной записи заданы время старта и завершения. Используйте множества обобщений и супертипы *Type* и *Time*.

б. Используя паттерн Decorator, добавьте динамическую учетную запись *DynAccount*, которая может менять тип и время действия по определенным ранее вариантам.

в. Покажите на диаграмме состояние системы после совершения двух транзакций по ресурсам *r1* и *r2* постоянным корпоративным пользователем с *DynAccount* по тарифу *basicTariff*. В каждой транзакции расходуется две единицы ресурса.

4.18. В Domain-Driven Design при моделировании предметной области используются особые виды типов данных и классов: объект-значение *Value Object*, сущность *Entity*, корень агрегата *Root*. В дополнение к определению класса UML используемые в DDD виды классов несут дополнительную информацию, которая не может быть представлена в стандартной метамодели UML. На рис. 13 показана заготовка профиля.

а. Используя уже имеющиеся стереотипы, добавьте в заготовку профиля виды классов *Entity* и *Root*. Приведите решение на диаграмме профилей.

б. Агрегат состоит из корня, всех сущностей и объектов-значений, достижимых через свойства из корня без учета их видимости. Покажите это в модели, введя подходящий базовый класс и ассоциации между стереотипами.

в. Покажите, что некоторые пакеты могут быть модулями *Module*. Модуль определяет операцию *encloses* проверки принадлежности элементов непосредственно модулю или пакету, вложенному в него.

г. Используя ограничения, покажите, что корень агрегата может возвращать только объекты-значения или корни агрегатов, и что все общедоступные свойства или параметры операций возвращаемых типов объекты-значения, либо корни агрегатов.

д. Добавьте стереотипы сервиса предметной области *Domain Service* и репозитория *Repository* так, чтобы для метакласса *Interface* можно было указать только один из них. Покажите, что сервисы и репозитории можно передавать между агрегатами и интерфейсами сервисов.

е. Известно, что репозитории, сервисы предметной области и классы, их реализующие, не могут иметь свойства типа объект-значение или сущность. В то же время классы предметной области фабрики *Factory* могут. Отразите это в модели.

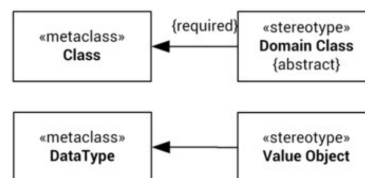


Рис. 13

§5. АНАЛИЗ И ВЫДЕЛЕНИЕ КЛАССОВ

ОСНОВНЫЕ ПОНЯТИЯ

Абстрактный тип данных (abstract data type) определяет совокупность объектов, которые полностью описываются одним набором действий над этими объектами. Таким образом, абстрактный тип данных полностью определяется составом и условиями выполнения этих действий. Классическими примерами абстрактных типов данных являются стек, очередь, список.

Обязанности (responsibilities) выделяют как составные элементы решения проблемы и назначают объектам или классам, ответственным за их реализацию в системе. Обязанности объекта или класса бывают трех видов: обязанность знать и владеть определенной информацией, обязанность выполнять определенные действия и предоставлять интерфейс для их вызова, обязанность организовывать взаимодействие других объектов.

Метод Аббота (Abbot method) [10] заключается в изучении текстовых описаний проблемы или постановки задачи с целью выделения классов-кандидатов. Метод определяет набор правил извлечения существительных из текста, объединения синонимов и исключения неподходящих существительных: числительных, не имеющих отношения к изучаемой проблеме понятий и других.

Метод именных групп (noun phrase) [10] является модификацией оригинального метода Аббота, изначально сформулированного для извлечения сущностей в структурном проектировании. Метод именных групп также основан на выделении существительных, но оставляет на усмотрение аналитика критерии отнесения класса-кандидата к релевантным поставленной задаче, нерелевантным или нечетким классам.

Метод карточек класс-контракт-коллеги (CRC) [10] направлен на анализ взаимодействия объектов для реализации некоторой функции системы. При рассмотрении сценариев взаимодействий участники формулируют обязанности предполагаемых объектов. Метод применяется итеративно, последовательно рассматривая ключевые функции системы. Результатом метода являются описания обязанностей объектов и их взаимодействий, обычно представляемые в виде карточек формата 90 на 140 мм.

Метод шаблонных классов [10] заключается в поиске в системе классов, обычно выделяемых в сходных по назначению системах. Для этого применяются перечни категорий классов, таких как внешние сущности, вещи, события, организационные единицы, места или структуры классов из других категорий.

Для выделения релевантных классов из списка классов-кандидатов также применяют разнообразные эвристики и **контрольные списки (checklisting)**. Контрольный список включает несколько вопросов относительно класса-кандидата и критерий исключения классов-кандидатов. Например, класс должен описывать множество объектов, содержать несколько атрибутов и операций, применимых к каждому объекту, быть важным для описания проблемы или предметной области.

ЗАДАЧИ

5.1. Диск состоит из пронумерованных кластеров. На диске есть именованные папки, в которые вложены папки или именованные файлы. Список файлов и папок в папке хранится в одном кластере диска, данные файлов хранятся в нескольких кластерах.

а. Выделите классы и определите отношения между ними, используя абстрактные типы данных (ADT) и метод Аббота.

б. Добавьте операции и атрибуты к имеющимся классам для создания, удаления папок и файлов, записи и чтения буфера данных с определенной позиции в файле.

5.2. Больной посещает доктора, чтобы получить рецепт на лекарства для своей болезни.

а. Применив метод Аббота, выделите классы и постройте модель предметной области для системы учета посещений больными докторов для поликлиники.

б. Применив метод Аббота, выделите классы и постройте модель предметной области для программы-ежедневника приема лекарств для посетителей.

Рис. 14

5.3. В межгосударственном стандарте по оценке качества программных средств ГОСТ 28195—89 качество характеризуется набором факторов. Краткое описание: на каждом из этапов разработки программного средства фактор описывается набором критериев. Каждый критерий измеряется с помощью нескольких метрик, различающихся для этапов разработки. Метрики могут использоваться разным критериями.

а. Выделите классы, применив метод именных групп к краткому описанию из условия задачи.

б. Уточните отношения между классами, учитывая, что на разных этапах разработки факторы описываются разными наборами критериев, которые измеряются различными метриками.

в. На рис. 14 представлена схема критериев и метрик качества для фазы анализа. Покажите ее представление в модели на диаграмме экземпляров.

г. Используя схему, представленную на рис. 15, постройте критерии и метрики надежности для этапа реализации программного средства.

5.4. Аудиоплееры состоят из менеджера подключаемых модулей, пользовательского интерфейса, который обрабатывает пользовательский ввод, управляющего компонента, реализующего основную функциональность, и библиотеки мультимедиа. Загружаемые модули взаимодействуют с компонентами плеера.

а. Выделите классы и определите отношения между ними, используя абстрактные типы данных (ADT) и метод Аббота.

б. Для загрузки, включения и выключения подключаемых модулей добавьте операции и атрибуты к выделенным классам. Полагайте, что пользовательский интерфейс обрабатывает команды от пользователя асинхронно.

в. Уточните описание аудиоплеера, чтобы в нем присутствовал менеджер соединений, и в системе в целом был сервер с базой доступных модулей. Менеджер модулей может, используя менеджер соединений, подключаться к серверу с целью проверки обновлений установленных модулей.

Рис. 15

5.5. Среди основных функций системы управления клиентами (CRM) выделяют управление карточками клиентов *ManageClients*, обработку заявки, проведение сделки и подготовку отчета. Менеджер по продажам использует эти функции в своей работе, кроме подготовки отчета, которую выполняет руководитель отдела. Клиенты участвуют в обработке заявки и проведении сделки. В карточке клиента *ClientCard* указываются его реквизиты, поданные заявки *Note* и проведенные сделки *Deal*. У заявки обязательно указывается дата и время, текст сообщения, может быть указан текст ответа. Менеджеру по продажам назначается несколько карточек клиентов и один руководитель отдела.

а. Выделите акторов и варианты использования, покажите на диаграмме. Решение поясните.

б. Выделите оставшиеся классы-кандидаты, используя метод именных групп (noun phrase).

в. Постройте модель предметной области, укажите отношения между классами, хранимые классами данные и операции классов. Решение поясните.

§6. АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ, КОМПОНЕНТЫ

ОСНОВНЫЕ ПОНЯТИЯ

Структурированным классификатором (structured classifier) называется классификатор, который может включать соединители, связывающие содержащиеся в классификаторе свойства. Структурированные классификаторы определяют контекст для соединителей и позволяют описать связи между экземплярами, возникающие во время выполнения системы.

Портом (port) называется часть структурированного классификатора, через которую осуществляется взаимодействие с другими внешними классификаторами. Порт может предоставлять и требовать наличия нескольких интерфейсов. Взаимодействие, осуществляемое через **поведенческий** (behavioral) порт, не делегируется частям классификатора, а обрабатывается им самим.

Делегирующий соединитель (delegating connector) связывает порт с одной из частей классификатора, если данная часть реализует предоставляемый портом интерфейс или нуждается в требуемом интерфейсе.

Сборочный соединитель (assembly connector) связывает части классификатора или разных классификаторов между собой и указывает возможные пути взаимодействия частей.

Компонент (component) является структурированным классификатором, обладающим поведением. Компонент – это модуль системы, который скрывает свою внутреннюю структуру, взаимодействует с другими компонентами через определенные интерфейсы, и материализация (manifestation) которого в системе может быть заменена при сохранении окружения (environment). Таким образом, поведение компонента полностью определяется набором предоставляемых и требуемых интерфейсов. Компоненты используются для определения повторно используемых частей системы и для описания модульной структуры системы.

Артефакт (artifact) представляет в модели файл, документ или другой объект, который используется или создается в процессе разработки программы или в результате ее установки и использования. Артефакт в модели является классификатором и может вступать в ассоциации с другими артефактами.

Артефакты **материализуют (manifestation)** другие элементы модели, таких как компоненты.

Узел (node) описывает вычислительный ресурс, на котором могут быть развернуты для выполнения артефакты. Узлы взаимодействуют через **подключения (communication path)**. В метамодели узел является специальным видом класса, а каналы связи – ассоциациями между узлами. Дочерние классы узла уточняют вид ресурса: вычислительные элементы оборудования моделируются **устройствами (device)**, а программные платформы **– средами исполнения (execution environment)**.

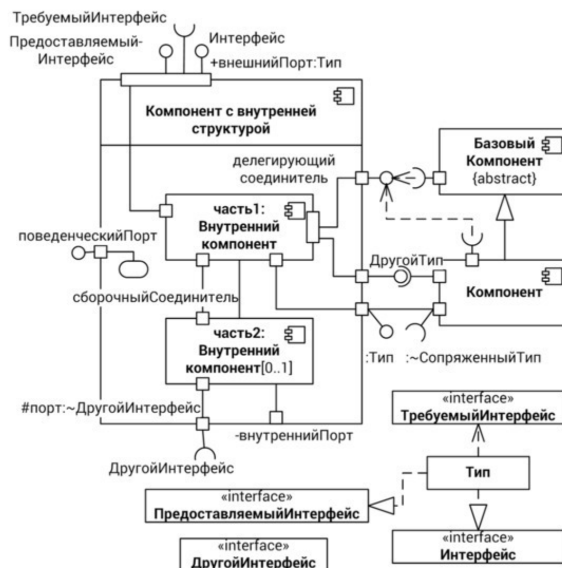


Рис. 16. Нотация диаграмм компонентов

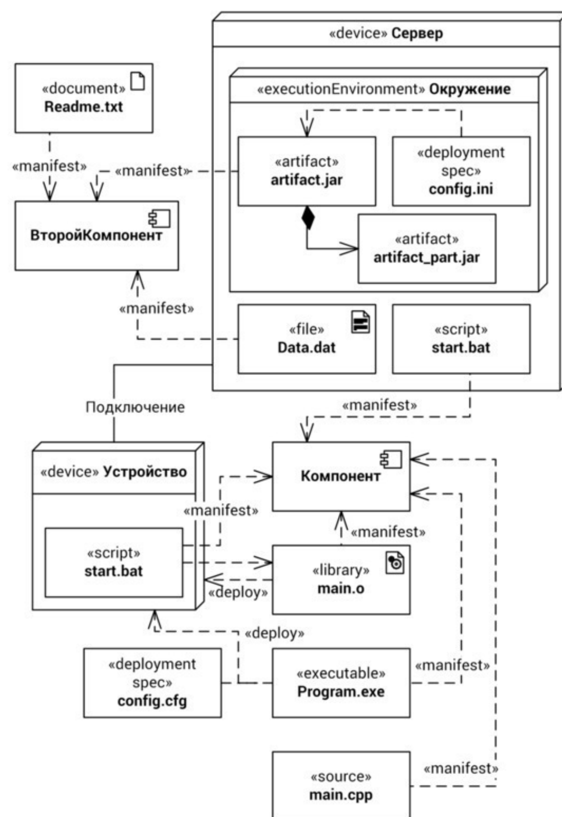


Рис. 17. Нотация диаграмм размещения

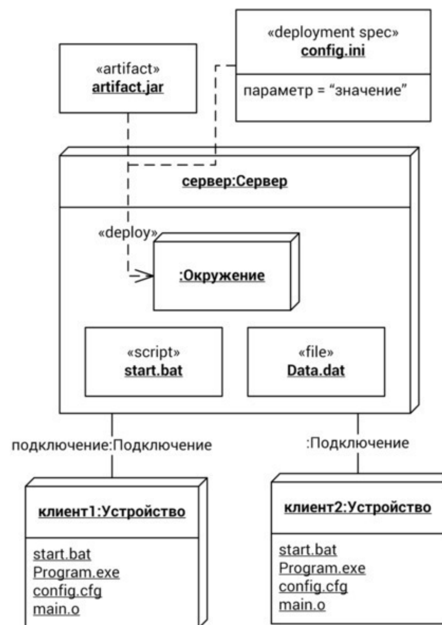


Рис. 18. Нотация диаграмм размещения на уровне экземпляров

ЗАДАЧИ

6.1. Структура поисковой системы представлена на рис. 19.

а. Покажите, что индекс *Index* может быть только двух видов – простой *SimpleIndex* и распределенный *DistrIndex*.

б. Распределенный индекс через очередь сообщений *MsgBus* взаимодействует с узлами индекса *IdxNode* по протоколу *AMQP*. Высоконадежная версия очереди сообщений состоит из нескольких реплик, соединенных через *InLink*.

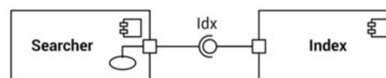


Рис. 19

в. Материализуйте и разместите компоненты системы согласно многозвенной архитектуре, в которой распределенная очередь сообщений располагается на выделенном сервере.

6.2. Файл *Morphology_.dll* материализует компонент *MorphoEngine*, который предоставляет интерфейс *IMorphology*. Компоненту *MorphoEngine* для работы необходим компонент *RootObjects* и файлы словарей. Файлы словарей имеют названия *<ISO_639—1_код_языка>.lng*. Например, *«ru_.lng»*, *«en_.lng»*. Компонент *RootObjects* материализован в библиотеке *RootObject_.dll*.

а. Отобразите в модели артефакты и отношения между ними, необходимые для запуска морфологического модуля для работы с французским и немецким языками.

б. Укажите, что для локализации сообщений пользователю компонент *MorphoEngine* использует интерфейс *IMorphoLocalize*. Этот интерфейс уже реализован для русского и английского языков компонентами *MorphoLocalizeRu* и *MorphoLocalizeEn*, материализованными библиотекой *MorphoLocalize. dll*. Добавьте в модель зависимость от компонента русской локализации.

в. (*) Пометьте, что для корректной работы морфологическому модулю нужна библиотека *RootObjects_.dll* версии *4.0.1.157*. *Указание.* Введите подходящий профиль.

6.3. Лифт *Elevator* состоит из кабины класса *Cage*, пульта управления класса *ControlUnit* и нескольких панелей вызова с этажа класса *FloorControls*. Соединитель между пультом управления и кабиной имеет тип *cageWire*, между пультом и панелями – *floorWire*. При этом пульт подсоединен к каждой панели индивидуально.

а. Добавьте в модель двигатель класса *Engine* как составную часть лифта. Двигатель связан с кабиной кабелем *cable* и с пультом схемой управления *controls*.

б. Доработайте модель так, чтобы взаимодействие лифта с внешними классами происходило только через интерфейс кнопок кабины *CageControls*, управления лифтом *Operations* и интерфейсы вызова с этажей *FloorButtons*. Команды, принимаемые через интерфейсы, направляются на соответствующие части лифта.

в. Укажите, что для работы лифту требуется подключение к электрической сети *Power*.

г. Перечислите имена и типы всех элементов пространства имен класса *Elevator*.

д. Чему соответствуют порты класса *Elevator* в программном коде реализации класса *Elevator*?

6.4. Подсистема подготовки данных модуля морфологии *MorphologyDPS* состоит из базы данных *Database*, клиента для модификации данных *DataClient*, компонента экспорта *Export* и компилятора данных *Compiler*.

а. База данных предоставляет интерфейс изменения данных *IMorphologyData* и интерфейс экспорта данных *IDataExport*. Клиент требует для работы интерфейс изменения данных, в то время как компонент экспорта требует интерфейс экспорта данных. Компилятор не требует внешних интерфейсов, но неявно зависит от базы данных. Укажите в модели, как компоненты связаны между собой в подсистеме.

б. Разместите базу данных на сервере *MorphoDB*, а остальные компоненты на компьютере лингвиста *LinguistWorkPlace*.

в. Уточните внутреннюю структуру компилятора следующим образом. Компилятор использует интерфейс *IMorphology* компонента *MorphoModel*. Сам компилятор состоит из парсера *Parser*, обработчика сообщений об ошибках *Handler* и сборщика модели *Builder*. Компоненты, реализующие парсер и сборку моделей, сообщают об ошибках через интерфейс *IErrorHandler* компонента *Handler* в составе компилятора. Сборщик модели компилятора требует внешний интерфейс *IMorphology*.

6.5. Приложение класса *Application* содержит подключаемые модули. Подключаемый модуль класса *Bean* является либо процессным модулем *ProcessorBean*, либо алгоритмическим модулем *ComputeBean*. Процессный модуль связан *ComputeLink* с подключаемыми модулями для выполнения расчетов.

а. Используя представление внутренней структуры, укажите, что специализация *MainApp* приложения *Application* включает один процессный модуль и два связанных с ним алгоритмических модуля.

б. Доработайте модель, укажите, что приложение *MainApp* включает два связанных процессных модуля, один из которых является основным *main*.

в. Покажите, что основной процессный модуль приложения *MainApp* реализует интерфейс управления вычислениями *Computation*, предоставляемый приложением через порт веб-сервисов *ComputationEndpoint*.

г. Укажите, что приложение *MainApp* предоставляет и реализует интерфейс конфигурации *Configuration* через порт *ConfigurationEndpoint*.

д. Используя соединители сборки, покажите, что основной процессный модуль приложения *MainApp* может обращаться через интерфейс *Computation* к приложению *SecondApp*.

е. Перечислите все черты приложения *MainApp*.

6.6. Интерфейсом *Frontend* сервиса *MonolithService* пользуются веб-сайт *Website* и *API*. Интерфейс предоставляет методы работы с кабинетом пользователя и с файлами пользователя. Реализация сервиса использует кэш *Cache* и контекст *EntityContext* доступа к данным.

а. Проведите рефакторинг интерфейса сервиса, если известно, что веб-сайт пользуется и кабинетом, и файлами, а *API* только файлами пользователя.

б. Разделите сервис *MonolithService* на модули, выделив подходящие абстракции на основе принципа сокрытия информации Information Hiding. Обеспечьте доступ модулей к кэшу и контексту доступа к данным. Решение поясните.

в. Необходимо обеспечить обратную совместимость с прошлыми версиями клиентов, предоставив интерфейс *Frontend* целиком. Воспользуйтесь паттерном Adapter для его реализации. Предполагайте наличие legacy-методов, которые отсутствуют в новой реализации.

6.7. Для системы обработки текстов выбран архитектурный стиль Pipes-and-Filters. Одна из функций системы – векторизация текста – включает чтение текста из файла, очистку текста от служебных символов и знаков пунктуации, разделение на слова, нормализацию и вычисление частоты вхождения в текст каждого слова и запись результатов в файл. Размещение системы предполагается на серверах *Srv1* и *Srv2* с ОС Linux, соединенных вычислительной сетью.

а. Покажите на диаграмме классов структуру системы, если известно, что фильтры бывают только обработчиками *Processor*, источниками *Source* или результатами *Result*.

б. Воплотите компоненты для функции векторизации текста в исполняемых файлах и разместите их на сервере *Srv1*.

в. Добавьте в компоненты системы возможность сбора статистики через интерфейс *Status* агентами сбора статистики *Monitor*. Разместите компоненты вместе с агентами и управляющим компонентом статистики на двух серверах *Srv1* и *Srv2* так, чтобы нормализация текста проводилась отдельно на сервере *Srv2*, где также установлен управляющий компонент статистики.

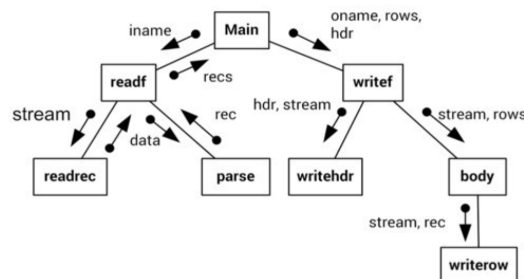


Рис. 20

6.8. На структурной диаграмме[4] (рис. 20) показан фрагмент структуры программы построения отчета. Программа читает записи данных *recs* из файла с именем *iname*, рассчитывает таблицу рядов показателей *rows* и записывает их в файл с именем *oname*. Имена файлов передаются программе в параметрах. Заголовок отчета *hdr* содержит названия отчета и показателей.

а. Перечислите модули, показанные на представленной диаграмме. Добавьте на диаграмму недостающие один или несколько модулей построения отчета, используя критерий сокрытия информации Information Hiding. Решение поясните.

[4] Структурная диаграмма или structure chart (https://en.wikipedia.org /wiki/Structure_chart) используется для описания структуры модулей программ, построенных в архитектурном стиле call-and-return [11].

б. Преобразуйте программу к объектно-ориентированному стилю, используя обращение зависимостей (DIP) и технику спецификации модулей (module specification)[5]. Покажите структуру программы на диаграмме.

в. Преобразуйте программу в сервис с веб-интерфейсом *WebUI* и программным интерфейсом *RESTService*. Каким образом следует реализовать в сервисе построение отчета по базе данных? Построение другого отчета?

6.9. Удаленный отладчик для встраиваемых программ *TargetDebugger* позволяет собирать треки выполнения программы на устройстве, передавать их для сопоставления с UML моделью программы на рабочей станции разработчика. Отладчик состоит из программы-монитора на устройстве *DeviceMonitor* и модуля подключения по сети *RemoteLib*.

а. Интерфейс *Monitor* монитора позволяет указать *setBreakpoint* точку останова по адресу в памяти *addr*, остановить *pause* и продолжить *resume* выполнение программы. Покажите на диаграмме классов интерфейс монитора, и его реализации для разных платформ *ARM7*, *PowerPC* и *Cortex-M3*.

б. Покажите структуру модуля подключения по сети, используя бинарный протокол *BinaryTcp* и текстовый протокол *Telnet*. Воспользуйтесь паттерном *Proxy* и представьте решение на диаграмме компонентов. Решение поясните.

в. Выберите артефакты и разместите модули отладчика на рабочем месте разработчика и устройстве на *ARM7*, предполагая связь по *Telnet*. В каком архитектурном стиле реализован отладчик?

[5] Восходящий (bottom-up) метод проектирования, основанный на выделении модулей путем объединения существующих модулей. См. статью D.L. Parnas. A Paradigm for Software Module Specification with Examples. 1971.

§7. СХЕМЫ СОСТОЯНИЙ И КОНЕЧНЫЕ АВТОМАТЫ

ОСНОВНЫЕ ПОНЯТИЯ

Состояние (state) конечного автомата, связанного с классификатором, моделирует ситуацию в жизненном цикле классификатора, когда выполняется некоторое, обычно неявное, условие. Например, свойства классификатора принимают значения из определенного множества, или условия при которых прием некоторого сигнала или вызов операции приводит к исполнению одного и того же поведения. В процессе выполнения конечного автомата, состояния могут быть активными и неактивными.

Композитным состоянием (composite state) называют состояние, у которого есть вложенные состояния. Если композитное состояние активно, то активно только одно из его вложенных состояний.

Ортогональное состояние (orthogonal state) включает два или более ортогональных региона, каждый из которых содержит вложенные состояния. Если ортогональное состояние активно, то в каждом ортогональном регионе активно только одно вложенное состояние.

Схема состояний (statechart) описывает недетерминированный иерархический конечный автомат, может включать композитные и ортогональные состояния. Конечный автомат в схеме состояний выполняется в контексте экземпляра некоторого классификатора, поведение которого он описывает. Схемы состояний часто используются для описания жизненного цикла объектов, возможных сценариев исполнения вариантов использования и бизнес-процессов.

Триггером (trigger) называют описание события, при возникновении которого будет выполнено указанное действие, осуществлен переход между состояниями. Если вместе с триггером указано сторожевое условие, то переход будет осуществлен, только если истинно сторожевое условие. Определены следующие виды событий: приема сигнала, вызова операции, событие времени, событие изменения. Примечательно событие получения неопределенного сообщения (AnyReceiveEvent), которое возникает, если для полученного сообщения не определен триггер.

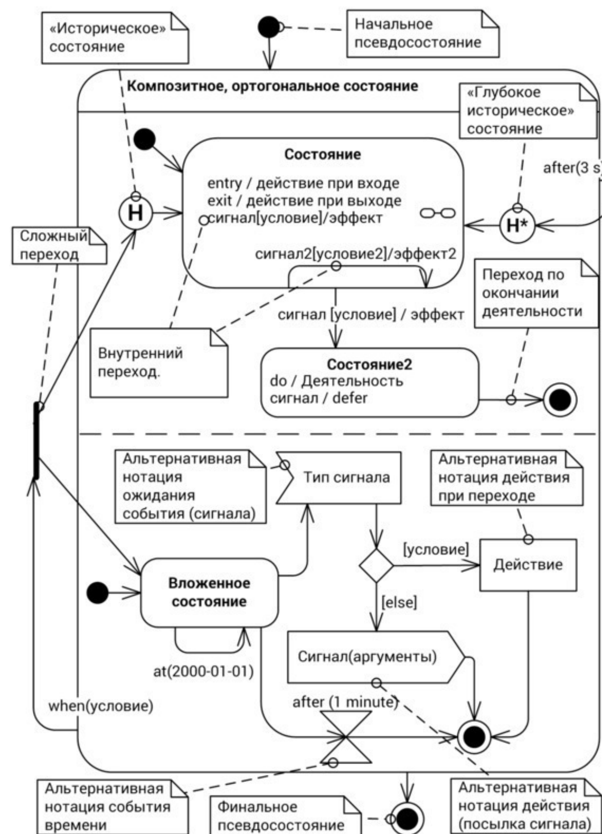


Рис. 21. Основная нотация диаграмм схем состояний

Переход (transition) указывает, каким образом конечный автомат реагирует на возникновение события. Простой переход соединяет два узла в схеме состояний автомата. Сложный или комплексный переход соединяет несколько ортогональных исходных или конечных состояний. Внутренний переход обеспечивает выполнение эффекта без смены состояния. Если из активного исходного состояния осуществляется переход в другое конечное состояние, то конечное состояние становится активным.

Эффектом (effect) при переходе называют действие, выполняемое при осуществлении перехода. Действие выполняется в контексте схемы состояний, при этом активным является состояние, в которое вложены исходное и конечное состояние перехода, если такое имеется.

Схема состояний может включать различные **псевдосостояния (pseudostate)**: начальное (initial), историческое (history state) и полное возвратное (deep history), разделения и слияния (для комплексных переходов), выбора (choice), точки входа (entry) и выхода (exit), терминальное (terminal) и соединительное (junction). Псевдосостояния обычно используются для объединения простых переходов в более сложные конструкции.

При достижении **конечного состояния (final state)**, состояние или регион, его содержащие, считаются завершенными. Если достигается конечное состояние в схеме состояний, то конечный автомат считается завершенным, и контекстный экземпляр классификатора уничтожается (terminates).

ЗАДАЧИ

7.1. Моделируется мобильное приложение – электронный словарь, имеющее несколько окон. Схему перехода между окнами будем моделировать с помощью диаграммы схем состояний.

а. Добавьте на диаграмму состояния *Список слов*, *Перевод*, *Словоформы слова*. Переход между *Список слов* и *Перевод* по событию *Выбрано слово*, обратный переход по событию *Назад*. Переход между *Перевод* и *Словоформы слова* по событию *Словоформы*, обратный переход по событию *Назад*.

б. В некоторых карточках перевода есть ссылки на другие карточки. При переходе по ссылке мы попадаем в то же состояние *Перевод*, но для другого слова. Добавьте переход в себя из состояния *Перевод*, активируемый событием *Переход по ссылке*.

в. Возврат после перехода по ссылке по событию *Назад* должен возвращать пользователя в предыдущую карточку. Для этого добавим к переходу по ссылке действие *Добавить в стек*. Добавим переход в себя из состояния *Перевод* по событию *Назад* со сторожевым условием *[стек не пуст]* и действием *Убрать из стека*. В переход из состояния *Перевод* в состояние *Список слов* по событию назад нужно добавить сторожевое условие *[else]*.

г. Перед запуском приложения нужно загрузить словари. На время загрузки словарей пользователю будет показан экран приветствия. Добавим ортогональное состояние *Инициализация*. В нем выделим два региона. В первом регионе добавим начальное и конечное псевдосостояния. Между ними добавим состояние *Загрузка* словарей с выполняемой при нахождении в состоянии деятельностью *Загрузить словари*. Ожидается переход в конечное состояние по завершении деятельности. Во втором регионе мы добавим переход из начального состояния в конечное по событию времени *after (3 s)*. Это необходимо, чтобы при быстрой загрузке пользователь успел прочитать содержимое экрана приветствия. Из ортогонального состояния *Инициализация* переход по завершению в *Список слов*.

д. Уточним поведение в окне списка слов. *Список слов* становится составным состоянием. Дальнейшие состояния являются вложенными в *Список слов*. Из начального псевдосостояния происходит переход в состояние *История запросов*. По событию *Ввод* из *Истории запросов* происходит переход в псевдосостояние выбора. Далее при условии *поле ввода пустое* происходит переход (возврат) в состояние *История запросов*. Иначе происходит переход во вложенное составное состояние *Поиск слова*. По событию *Ввод* происходит переход из *Поиск слова* в описанное выше псевдосостояние выбора. В составном состоянии *Поиск слова* непосредственно процедура поиска откладывается с помощью события времени *after (1 s)*. После этого события происходит переход из начального псевдосостояния в состояние *Поиск в словаре*. По завершению происходит переход из *Поиск в словаре* в *Отображение списка*.

7.2. (см. решение в §11) Светофор *TrafficLights* после создания переходит в состояние выключен *Offline*. При включении *On*, светофор переходит во вложенное состояние «зеленый» *Green* состояния включен *Online*. По истечении 50 секунд, светофор переходит в состояние «желтый» *Yellow* в *Online*. Затем, через 3 секунды – в состояние «красный» *Red*. По истечении 50 секунд светофор возвращается в состояние «зеленый».

а. Добавьте возможность выключить *Off* включенный светофор.

б. Доработайте модель, укажите, что интервалы *t* между переключениями сигналов светофора настраиваются вызовом операций *setGreen*, *setRed* и *setYellow* в выключенном состоянии.

в. Измените порядок включения светофора, используя сторожевые условия, укажите по аналогии с предыдущим пунктом, что начальный сигнал светофора при включении *initialGreen*, *initialYellow* или *initialRed* настраивается *setInitial* в выключенном состоянии.

7.3. После создания статья *Paper* является черновиком *Draft*. После отправки *sent* статья рассматривается программным комитетом конференции *OnReview*, при этом выполняется рецензирование статьи *reviewPaper*. При согласии комитета *approved*, статья принимается на конференцию *Accepted*. Если комитет не принял статью *declined*, статья становится черновиком *Draft*.

а. Уточните изменения состояний статьи при рецензировании. После получения статья проверяется *Checking*. Если обнаружены недостатки, статья направляется на доработку *correct* в состояние *OnCorrection*. Если недостатков нет, рецензирование статьи завершается. При повторной отправке статьи *sent* она проверяется повторно *Checked*.

б. Укажите, что если статья не была отправлена с исправлениями недостатков в течение 10 дней, рецензирование прекращается и статья становится черновиком *Draft*.

7.4. Самолет *Aircraft* изначально находится *InHangar* в ангаре. При выходе на рейс *flight* самолет переходит на посадку *Boarding*. В полете *InAir* самолет выполняет деятельность по управлению полетом *flightControl*. В начале полета самолет убирает шасси *pullGearUp*, и выпускает в конце *pullGearDown*.

а. Устраните недостаток в модели – покажите, что при получении разрешения на взлет самолет переходит в состояние взлета *TakeOff* и после отрыва – в состояние полета. При получении разрешения на посадку *landingPermit* самолет переходит в *Landing* и, по прибытии, переходит на посадку.

б. В соответствии с требованиями аэропорта, самолет должен отправлять сигнал об освобождении полосы *freeRunway* после взлета и перед прибытием, и поддерживать постоянный радиоконтакт *radioComm* при нахождении на полосе.

в. Уточните состояние полета, укажите, что в долгом полете через час после взлета пассажирам предлагается обед *Dinner*, который длится один час.

г. Отразите в модели, что в случае возникновения неисправности в полете *malfunction*, самолет будет поврежден *Damaged*. Покажите, что самолет может совершить посадку только в исправном состоянии *Normal*.

д. Что произойдет, если самолет получит разрешение на посадку во время обеда? Ответ поясните.

7.5. Контроллер мультимедиа-проигрывателя *PlayerController* изначально находится в нерабочем состоянии *NotInitialized*. При запуске проигрывателя происходит переход в состояние загрузки *Loading*, при входе в которое инициализируются управляющий компонент *initEngine* и интерфейс проигрывателя *initUI*. В данном состоянии загружаются доступные модули *loadPlugins*. По завершению загрузки контроллер переходит в состояние готовности *Ready*.

а. Основное назначение проигрывателя – воспроизводить аудио и видео. Добавьте в состоянии готовности режимы воспроизведения, остановки и паузы, а также события, инициирующие переходы между этими режимами.

б. Хорошие мультимедиа-проигрыватели параллельно воспроизведению автоматически обновляют медиатеку: осуществляют поиск нового контента *Searching* и загружают информацию о нем из сети *Downloading*. Добавьте в модель данную функциональность.

7.6. Изначально настольная лампа *Lamp* выключена *Off*. При переключении выключателя *turnOn* лампа работает *Light*. Когда лампа работает, при переключении *turnOff* она выключается *Off*.

а. Доработайте модель состояний. Укажите, что если лампа падает со стола *fell* или сгорает в процессе работы *burnDown*, то она выходит из строя *Damaged*.

б. Укажите, что неисправные лампы *Damaged* утилизируются *dispose* с уничтожением лампы.

7.7. Телефонный аппарат *Phone* после включения становится свободен *Available*. Если поднять трубку *takeHandset*, то аппарат станет активным *Active*. В активном состоянии пока не нажата цифра *digit* или не прошло 15 секунд аппарат проигрывает гудок *playTone*. После набранной цифры аппарат ожидает набора следующей цифры *Dial*, если не возникла ошибка *error* и номер еще не полный *continue*. Если при наборе цифры номер завершен *complete*, то предпринимается попытка соединения *connect* с переходом в состояние занято *Busy*, если вызы-

ваемый номер занят *busy*, или вызов *Ringing*, если свободен *free*. Когда абонент возьмет трубку, аппарат начинает разговор с передачей голоса *voiceComm*.

а. Укажите, что аппарат переходит в состояние таймаут *Timeout* по истечении 15 секунд при проигрывании гудка до набора номера и при ожидании ответа.

б. Реализуйте переход в состояние *Available* по возвращении трубки на место *returnHandset*. Как влияет текущее состояние телефона на выполнимость этого перехода?

в. Укажите, что аппарат проигрывает короткие гудки *playBusy*, когда вызываемый номер занят, и длинные *playRing*, когда идет вызов свободного номера.

г. Покажите в модели, что после завершения разговора абонентом *hangUp* данный аппарат переходит в состояние *Busy*.

д. Пусть состояние вызываемого номера аппарату сообщает АТС в результате попытки соединения, отразите это в модели так, чтобы выполнялись требования run-to-completion. Ответ поясните.

7.8. При вызове операции пассивного открытия *listen* модуль протокола TCP переходит из *Closed* в *Listen*. При получении сообщения *SYN* будет выполнен переход в *SYN_Rcvd* с отправкой *SYN+ACK*, и далее в *Established* при получении *ACK*. При вызове операции *close* модуль переходит в *ActiveClose*, отправляя *FIN*, и через 2 секунды возвращается в *Closed*. Если же в *Established* было получено сообщение *FIN*, то модуль переходит в *PassiveClose* и по получении *ACK* переходит в *Closed*.

а. Покажите поведение модуля TCP на схеме состояний. Перечислите состояния, переходы, укажите для переходов триггеры, виды событий (получение сигнала, вызов операции и др.) и эффекты при переходе.

б. Добавьте активное открытие соединения *ActiveOpen* через отправку *SYN* при вызове операции *connect* в *Closed*, и переходе в *Established* по получении *SYN+ACK*.

в. Уточните, что в *PassiveClose* модуль до получения *ACK* ожидает однократного вызова *close*, отправляя *FIN*. При том же вызове *close*, модуль переходит в *ActiveClose* из *SYN_Rcvd* с отправкой *FIN*, в *Closed* из *ActiveOpen*.

г. Преобразуйте схему состояний модуля протокола TCP в конечный автомат Мура.

7.9. При выполнении проекта *Run* руководитель проекта (РП) осуществляет отслеживание текущего статуса *Monitor*. Если выявляется проблема *problem*, то РП устраняет ее *ManageProblem*, и возвращается *succeed* к отслеживанию. Если РП не удается решить проблему *fail*, то происходит завершение проекта *Shutdown* с выполнением остановки *stop*, по завершении которой поведение руководителя проекта завершается.

а. Добавьте этап работы РП по подготовке *Initiate* проекта. На этом этапе при необходимости *estim* производится оценка *preparePlan* для проектов с бюджетом *budget* более 100K. Для остальных проектов готовится *sketch* набросок. По утверждении *approve* проекта РП переходит к управлению им *Run*.

б. Покажите на диаграмме, что при управлении проектом РП взаимодействует с другими *Communicate*. Если поступает запрос на собрание m*eetingRequest*, то РП его назначает *schedule*. По завершении *done* собрания *RunMeeting*, РП возвращается к взаимодействию.

в. Добавьте возможность РП приостановить *suspend* работы в любой момент. Если работы приостановлены более года, то РП завершает проект *Shutdown*. При возобновлении *resume*, выполнение проекта продолжается с того момента, когда он был приостановлен.

г. (*) Разделите роли руководителя проекта и старшего руководителя проекта. Старший РП так же в любой момент управления проектом принимает запросы на изменения *AcceptCR*, при получении *accept* которого проводит оценку *Estimate*, отвергает *reject* или принимает *approve* с возвращением к приему. *Указание*. Добавьте дочерний класс и используйте расширение схем состояний.

7.10. Согласно спецификации платформы Java в части управления памятью, при создании объект находится в состоянии *Created*, при входе в которое выделяется память *alloc*, и вызываются инициализаторы и конструкторы *init*. Когда локальной переменной метода присваивается *assigned* объект, он становится используемым *InUse*. Если переменная вышла из области достижимости *outOfScope*, объект становится недоступным *Unreachable*. При обнаружении сборщиком мусора, если в классе объекта определен метод *finalize ()*, и он не был вызван, то объект помещается в очередь на сборку в состоянии *Collected*, иначе объект становится очищенным *Finalized*. После того, как память объекта возвращена *deallocated*, жизненный цикл объекта завершен.

а. Укажите, что если переменная используемого объекта выходит из блока, в котором она была определена, и других ссылок на объект нет, то объект становится невидимым *Invisible*. После завершения выполнения метода, в котором определена переменная, объект становится недоступным.

б. Покажите, что в очереди на сборку выполняется деятельность *waitFinalize*, которая завершается после завершения выполнения метода *finalize ()*. По ее завершении, если объект воскрешен *resurrected*, то есть, обнаружены ссылки на объект, то объект становится используемым, иначе переходит в состояние *Finalized*.

в. Используя псевдосостояние выбора явно, покажите, что если в классе объекта не определен метод *finalize ()*, то недоступный объект при обнаружении сразу переходит в состояние *Finalized*.

г. Используя композитные состояния, покажите, что объект может воскреснуть, пока он невидим, недоступен, ожидает сборки или очищен.

д. Сколько раз объект может выходить из состояния недоступности? Ответ поясните.

7.11. Изначально кофеварка *CoffeeMachine* находится в состоянии выключена *Off*. При нажатии на кнопку включения *powerButton* кофеварка переходит в композитное состояние включена *On*. В этом состоянии она сначала производит самопроверку *SelfTest*. После успешной проверки (условие *testSuccess*) происходит переход в композитное состояние готовности к работе *Ready*. В данном состоянии кофеварка находится в режиме ожидания *Idle* до тех пор, пока пользователь не нажмет кнопку приготовления кофе. При получении от кнопки сигнала *makeButtonPressed* кофеварка сначала мелет зерна *Grind*, затем разогревает пар и обрабатывает паром *Steam* полученный молотый кофе. После чего кофеварка наливает заваренный кофе *Pour* и возвращается в режим ожидания.

а. Добавьте действия включения и выключения индикатора *indicatorOn*, *indicatorOff* при включении, выключении кофеварки.

б. Нагревать пар непосредственно перед приготовлением кофе – очень длительное занятие. Пользователь хочет получить свою чашку кофе как можно быстрее. Реализуйте с помощью ортогональных состояний возможность поддерживать температуру воды в кофеварке в течение всего состояния готовности, независимо от процесса приготовления кофе.

в. Добавьте кнопку отмены приготовления кофе, по нажатию которой процесс приготовления прерывается, кофеварка переходит в состояние готовности.

г. При помоле кофе, зерен может не хватить. Нужно возвращаться в режим ожидания, показывая пользователю соответствующее сообщение *displayNotEnoughCoffee* ().

д. Постройте минимальную достаточную структурную модель кофеварки.

7.12. Ознакомьтесь с моделью кофеварки, описанной в условии задачи 7.11. По заказу производителя необходимо внести доработки в базовую модель.

а. Кофеварка должна иметь съемный бак для воды. Если бак снят, необходимо перейти из состояния *Ready* в состояние *WaterTankRemoved* с сообщением пользователю *displayWaterTankRemoved ()*. Добавьте возможность продолжать прерванный снятием бака процесс приготовления кофе, когда бак для воды был установлен на место.

б. В случае неуспешной проверки *SelfTest* кофеварка должна переходить в такое состояние, что ни нажатие на кнопку приготовления кофе, ни другие действия пользователя не будут ни на что влиять. При этом нужно отобразить сообщение *displayErrorMessage ()*.

в. Постройте минимальную достаточную структурную модель доработанной кофеварки.

7.13. Микроволновая печь *Microwave* после создания переходит в состояние выключена *Off*. Если нажать кнопку включения *turnOn*, то печь переходит в состояние приготовления *On*. Перед началом приготовления, значение атрибута таймер *timer* класса *Microwave* устанавливается равным нулю, а значение атрибута время приготовления *cookingTime* принимается равным десяти. По истечении секунды в состоянии приготовления значение таймера увеличивается на единицу. Если при этом значение таймера превысило время приготовления, печь выключается.

а. Добавьте возможность перевести микроволновую печь по нажатию кнопки *turnOff* из состояния приготовления в выключенное состояние. Какое значение будет иметь атрибут *cookingTime*, если выключить, а затем включить печь? Ответ поясните.

б. Реализуйте функцию добавления 10 секунд ко времени приготовления при повторном нажатии кнопки *turnOn* во включенном состоянии.

в. Выделите вложенные состояния: работает *Working* и не работает *NotWorking* в состоянии приготовления, покажите, что печь переходит из *Working* в *NotWorking* при открытии дверцы *open* и обратно при закрытии *close*. При этом время приготовления отсчитывается только в состоянии *Working*, в которое печь попадает по умолчанию при переходе в состояние приготовления.

г. Используя ортогональные регионы в рабочем состоянии, отобразите в модели состояние дверцы печи: открыта *DoorOpen* и закрыта *DoorClosed*, и переходы между ними при получении сигналов *opened* и *closed*. Используя комплексные переходы, укажите, что печь при включении *turnOn* переходит в *Working*, если дверца закрыта, и в *NotWorking*, если открыта.

7.14. (*) Лифт *Elevator* изначально находится в состоянии *Offline*. При получении сигнала *on* он переходит в рабочее состояние *Online*, выполняя действие инициализации и очистки очереди вызовов *cleanup ()* при входе в состояние. Внутри *Online* лифт переходит в композитное состояние *Closed*, в котором двери закрыты. При получении сигнала *floor* с целым параметром *pos*, если этаж назначения *dest* совпадает с текущим этажом *pos*, *Elevator* переходит в *NotClosed*, также вложенное в *Online*. Лифт закрывает двери, переходя из *NotClosed* в *Closed* по истечении 10 секунд.

а. Определите класс очереди *Queue* с операциями добавления и извлечения из очереди целочисленного значения. Постройте структурную модель лифта с очередью вызовов по нажатию кнопок из кабины и очередью вызовов с этажей.

б. Уточните модель, указав каким образом лифт может получать команды вызовов с этажей (сигнал *call*) и нажатия кнопок (сигнал *button*) в состоянии *Online*. Команды не должны теряться, и при получении заносятся в очередь.

в. Детализируйте состояние *Closed*, укажите, что при входе в это состояние лифт проверяет список вызовов и ожидает получения команд, если список пуст. При получении команды в состоянии ожидания или наличии команд в списке лифт приходит в движение, ускоряется и замедляется перед нужным этажом.

г. Реализуйте возможность выключения лифта (сигнал *off*) с переходом в *Offline*. Решение обоснуйте.

д. Дайте полное описание структурной модели системы. Перечислите классы, операции, сигналы и приемы сигналов, а также все реализуемые классами методы (behaviors).

е. Приведите по одному примеру принимаемой и отвергаемой конечным автоматом, определяющим поведение лифта, последовательностей событий.

§8. ПРЕДСТАВЛЕНИЕ ДЕЯТЕЛЬНОСТИ И ПОТОКОВ РАБОТ

ОСНОВНЫЕ ПОНЯТИЯ

Действия (action) являются неделимыми единицами поведения. В модели действий (action model) определено несколько видов действий, основные из которых: вызов поведения и операции, прием и отправка сигнала, прием события времени, нечеткое действие (opaque action).

Деятельность (activity) используется для указания порядка исполнения элементов поведения с помощью зависимостей по управлению и по данным. Деятельность представляет собой направленный граф, в узлах которого расположены действия, группы действий, управляющие и объектные узлы деятельности. Узлы графа деятельности могут быть связаны ребрами: потоками управления, объектными потоками и дугами обработки исключений (exception edge).

Объектным потоком (object flow) называют поток между узлами деятельности, вдоль которого передаются метки с ссылками на объекты. Вдоль **потока управления (control flow)** передаются метки управления. Выполнение действия возможно, только если для всех его входных контактов какого-либо набора доступны метки со значениями. При наличии хотя бы одного входящего потока управления, по нему также доступна метка управления.

Контакты (pins) обозначают места присоединения потоков к исполняемым узлам деятельности (действиям или структурным узлам). Контакты объектных потоков соответствуют параметрам поведения, вызываемого действием. Контакты позволяют указать тип параметра, минимальное количество меток для вызова действия и способ передачи поступающих меток действию. **Наборы контактов** (**parameter set)** объединяют несколько контактов и обозначают необходимые для вызова действия наборы входных параметров или выходных параметров.

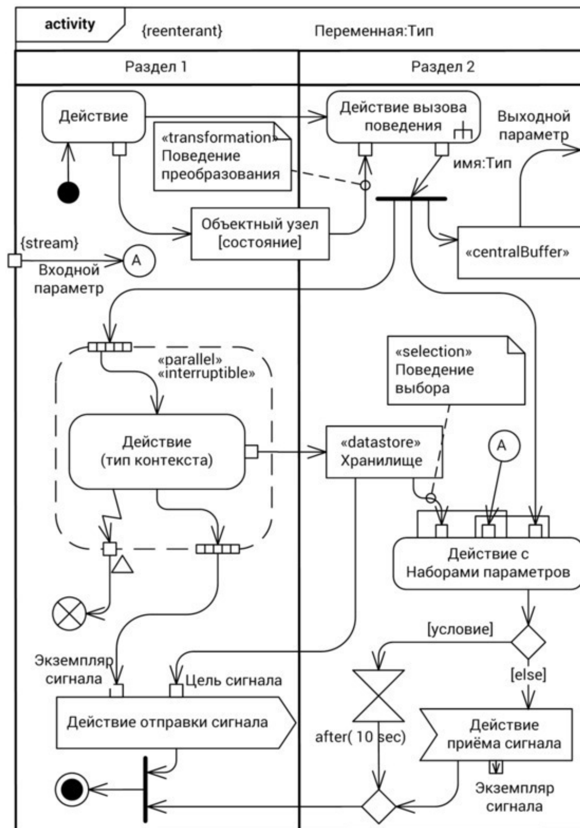


Рис. 22. Основная нотация диаграмм деятельности

Управляющие узлы (control nodes) деятельности указывают, при каких условиях вдоль объектных или управляющих потоков могут передаваться метки. Определены следующие виды управляющих узлов: разделения (fork) и объединения (join), ветвления (choice) и слияния (merge), начальный, конечный (flow final) и завершающий узел (activity final).

Объектные узлы (object nodes) обозначают экземпляры классификаторов, передаваемых действиям в деятельности. Объектные узлы используются при описании объектных потоков и позволяют указать дополнительные ограничения на передаваемые экземпляры и условия их распределения между исходящими потоками.

Центральный буфер (central buffer) позволяет перераспределить экземпляры между разными объектными потоками. **Хранилище данных (datastore)** является особым видом центрального буфера, который сохраняет все ссылки на экземпляры из поступивших меток. Хранилище по умолчанию предоставляет копию последней ссылки в каждый из исходящих объектных потоков. При этом может быть определен способ извлечения информации из всех экземпляров, на которые ранее поступили ссылки.

Раздел деятельности (partition) позволяет объединять в группы действия, обладающие какой-либо общей характеристикой. Например, разделы используются для уточнения контекста, в котором выполняются действия деятельности.

Структурным узлом деятельности (structured activity node) называют узел, объединяющий часть узлов и ребер деятельности, при этом подчиненные узлы могут принадлежать только одному структурному узлу деятельности. Структурный узел может иметь контакты для входных и выходных параметров, используемых подчиненными узлами.

ЗАДАЧИ

8.1. Рассмотрим алгоритм сборки программы из исходных файлов.

а. Деятельность *Сборка проекта* имеет входной параметр типа *Проект* и выходной параметр типа *Исполняемый модуль*. Входной параметр передается на узел действия *Анализ проекта*. Выходной контакт *Анализа проекта* имеет тип *Исходный файл* и множественную кратность. Исходные файлы передаются на узел действия *Компиляция*. Выходной контакт узла *Компиляция* имеет тип *Объектный файл* и передается на узел действия *Компоновка*. Выходной контакт узла *Компоновка* передается в выходной параметр деятельности.

б. Добавьте опциональное действие *Оптимизация*, которое производится над *Объектным файлом* при условии *[оптимизировать]*. Действие возвращает *Объектный файл* и передает его в действие *Компоновка*.

в. Уточним узел *Компиляция*. Этот узел представляет собой область разложения группы входных контактов, которая производит их распаковку с типом распаковки *«parallel»*. Объектный узел типа *Исходный файл* передается узлу действия *Лексический Анализ*. Результат передается узлу *Синтаксический анализ*, в свою очередь результат *Синтаксического Анализа* передается узлу *Генерация кода*. Результат *Генерации кода* имеет тип *Объектный файл* и является выходным контактом структурной деятельности.

г. В процессе *Лексического Анализа* заполняется таблица символов. Добавьте выходной контакт типа *Идентификатор* к узлу *Лексического Анализа*. Передайте объект с этого контакта в объектный узел с вида *«datastore»* и именем *Таблица идентификаторов*.

д. Идентификаторы из таблицы идентификаторов используются при генерации кода. Добавьте входной контакт для действия *Генерация кода* с типом *Идентификатор* и множественной кратностью. Передайте в этот контакт объекты из хранилища *Таблица идентификаторов*. Уточните поведение выбора из *Таблицы идентификаторов*, указав, что имя идентификатора должно совпадать с именем узла: *«„selection“ идент-ификатор. имя = узел. имя»*.

8.2. (см. решение в §11) Первым действием деятельности является добавление элемента в заказ *AddItem*. Если заказ сформирован *ready*, то он направляется получателю действием *ShipOrder* и деятельность завершается. В противном случае добавляется еще один элемент заказа.

а. Используя управляющие узлы деятельности, добавьте действие отправки счета *SendInvoice* так, чтобы оно исполнялось после того, как заказ сформирован, независимо от отправки заказа. При этом деятельность не завершается без отправки счета.

б. Предоставьте возможность после внесения элемента, добавить еще элемент в заказ при условии *continue* или завершить выполнение деятельности при условии *cancel*.

в. Пусть действие добавления элемента выполняется за $t1$ тактов, действие отправки заказа – за $t2$ тактов, действие отправки счета – за $t3$ тактов. Выполнение следующего действия начинается на следующий такт после завершения предыдущего. Определите полное время выполнения деятельности, если в доставленном заказе k элементов.

8.3. Рассмотрим процесс управления продуктовой линейкой *RunSPL*. Сначала заинтересованные стороны *stakeholders* ожидают поступления изменений *WaitChange*. При поступлении изменения *change*, оно передается для оценки *ReviewApprove* комитету *ReviewCommittee*. Результатом оценки является спецификация *spec*, которая передается далее рабочей группе линейки *SPLTeam* для планирования *Plan*. В результате планирования получается проект *project*, который также является результатом процесса.

а. Покажите на подходящей диаграмме контекст, в котором происходит выполнение процесса, и структуру участников. Укажите, что проекты поступают в результате процесса постепенно, до завершения выполнения самого процесса.

б. Сохраняйте все спецификации для последующего анализа. На планирование отправляйте последнюю сохраненную спецификацию.

в. Покажите, что если в результате оценки изменение было отклонено *rejected*, то выполнение процесса управления возвращается к ожиданию изменений. Если же изменение было принято *approved*, то процесс переходит к планированию спецификации и ожиданию новых изменений.

г. Посчитайте, сколько меток управления и сколько объектных меток создается при выполнении процесса сначала до создания одного проекта и отклонения одного изменения.

8.4. Деятельность *ImplementSolution* по реализации автоматизированной системы (АС) начинается с общения *Communication* аналитика *Analyst* с заказчиком. Созданное описание проекта *SoW* используется аналитиком для моделирования *Modeling*. Разработанная в результате модель *Model* сохраняется в репозитории проекта и передается в разработку *Development* и тестирование *Testing*, выполняемые проектной командой *DevTeam*. Результат разработки *Product* передается на тестирование, после этого производится его внедрение *Deployment* интегратором *Implementer*. Отчет о внедрении *Report* является результатом деятельности и деятельность завершается.

а. Покажите, если тестирование обнаружило дефекты в продукте, деятельность возвращается к разработке.

б. В условии задачи не сказано, что описание проекта используется для подготовки к внедрению интегратором. Добавьте данное действие и укажите, что внедрение выполняется только после завершения подготовки.

в. Укажите, в каком порядке выполняются действия. Будет ли деятельность выполнена хотя бы один раз, есть ли случаи, когда деятельность попадает в тупик (dead-lock)? Если есть, как это исправить? Ответы поясните.

г. Покажите, что общение может выполняться вместе с моделированием.

д. Перечислите все упоминаемые классификаторы, предполагая модель согласованной (well-formed).

8.5. Клиент *Customer* обращается к кассиру действием *AddressCashier*, кассир *Clerk* приветствует клиента *GreetCustomer* и спрашивает у него документы действием *AskDocuments*. Клиент достает квитанцию на выдачу денег со счета *Receipt* и передает ее кассиру. Кассир проверяет квитанцию при получении в действии выдачи денег *GiveMoney* и возвращает клиенту несколько банкнот класса *Note* в одной пачке. Клиент получает и пересчитывает деньги *ReceiveMoney*, прощается *Farewell* и уходит, завершая выполнение деятельности.

а. Добавьте кассиру действие поблагодарить клиента *ThankCustomer*, выполняемое после передачи купюр, не ожидая завершения подсчета банкнот клиентом, но до того, как он попрощался.

б. Добавьте проверку паспорта кассиром в новом действии *CheckPassport*. Клиент передает паспорт вместе с документами, но кассир сначала проверяет паспорт, затем использует паспортные данные *Identity* при выдаче денег и возвращает паспорт вместе с банкнотами.

в. Изменяя свойства контактов, добейтесь, чтобы кассир передавал, а клиент пересчитывал все банкноты по одной.

г. Рассчитайте полное количество созданных и поглощенных меток, передаваемых по ребрам деятельности, если кассир передает клиенту одну банкноту.

д. Перечислите классы, экземпляры которых используются при выполнении деятельности.

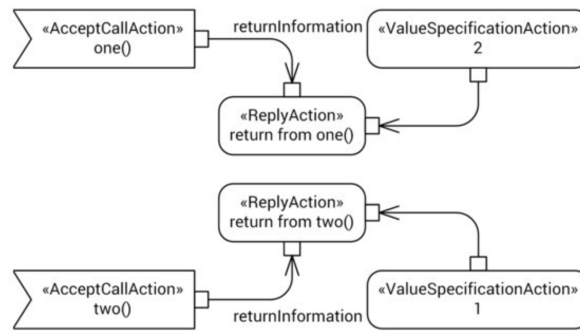


Рис. 23

8.6. В классе *Barrier* определены синхронные операции *one ()* и *two ()*. Поведение класса описывается реентерабельной *{isReentrant=true}* деятельностью, представленной на рис. 23.

а. Какие значения будут возвращены в результате вызова операций *one ()* и *two ()*? Ответ поясните.

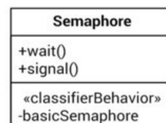


Рис. 24

б. Используя узлы разделения (fork) и объединения (join), укажите, что возврат из операции *one ()* возможен только после получения вызова операции *two ()* и наоборот.

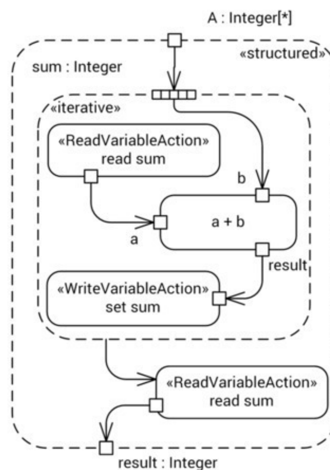


Рис. 25

8.7. Класс *Semaphore* реализует простой семафор с операциями *wait ()* и *signal ()*. Однократный вызов *signal* разрешает один вызов *wait*. Изначально *wait* будет ожидать вызова *signal*. Определение класса представлено на рис. 24.

а. Используя действия приема вызова операции *«AcceptCallAction»* и возврата *«ReplyAction»*, представьте поведение *basicSemaphore* в виде деятельности.

б. Измените модель таким образом, чтобы первый вызов *wait* не ожидал вызова *signal*.

в. Доработайте класс *Semaphore*. Реализуйте общедоступную операцию *signal (count: Integer)*, которая разрешает *count* вызовов *wait*.

8.8. На рис. 25 приведена диаграмма деятельности с условием задачи.

а. Что деятельность возвращает в результате?

б. Какой вид (kind) имеет действие *«a + b»*? Допустим, что в модели определено поведение *add*, реализующее сложение двух целых чисел. Какой вид будет иметь действие, вызывающее это поведение?

в. Доработайте деятельность так, чтобы она возвращала произведение элементов массива.

8.9. Деятельность по настройке подключаемых модулей начинается с построения *BuildPluginsList* и отображения *DisplayPluginsList* списка установленных подключаемых модулей. Затем деятельность продолжается только после получения сигнала от пользователя *UserActivityReceived*. В зависимости от выбора пользователя происходит либо конфигурирование конкретного модуля *ConfigureSelectedPlugin*, либо установка новых модулей *InstallNewPlugins*.

а. Укажите в модели, что операции *BuildPluginsList*, *ConfigureSelectedPlugin* и *InstallNewPlugins* реализованы в контроллере мультимедиа-проигрывателя *PlayerController*. В пользовательском интерфейсе *PlayerUI* реализована операция *DisplayPluginsList* и прием сигнала *UserActivityReceived*.

б. Когда список подключаемых модулей построен, необходимо проверить в фоновом потоке исполнения обновления модулей на сервере операцией *UpdatePlugins*, реализуемой контроллером, после чего поток управления завершается.

в. Реализуйте операцию обновления модулей *UpdatePlugins*, которая заключается в соединении с сервером *EstablishConnection* для каждого модуля проверки *CheckUpdates*, и загрузки обновлений *UpdatePlugin*, если они есть. Используя области расширения, добавьте в модель соответствующие элементы.

8.10. Из начального узла деятельности *buildTrafficLights* поток управления попадает в действие вызова поведения *createInstance*. Результат помещается в выходной контакт *result*. Поток управления из действия *createInstance* попадает в действие вызова поведения *createTrafficLights*. У действия есть выходящий контакт типа *TrafficLights*. Из него объектный поток попадает на входящий контакт действия вызова операции добавления светофора *addTrafficLights*. Операция вызывается у экземпляра типа *Controller*, передаваемого через второй контакт действия, на который попадает объектный поток из *createInstance*.

а. Предполагая, что деятельность выполняется в контексте класса *TrafficLightsBuilder*, перечислите все классификаторы, которые должны быть добавлены в модель, чтобы она стала согласованной.

б. Пусть поведение создания светофора принимает целочисленный параметр, а операция добавления светофора принимает в качестве параметра индекс для добавляемого светофора. Доработайте модель, укажите, что светофор создается с параметром *1* и добавляется действием *addTraficLights* с индексом *2*.

в. Укажите в модели, что деятельность имеет выходной параметр, значение которого является результатом действия создания экземпляра. В предположении, что деятельность должна быть согласованной, какие типы имеют входящие параметры действия *addTrafficLights*? Ответ поясните.

г. Будет ли построенная деятельность выполнена хотя бы один раз? Ответ поясните.

8.11. (*) В языках программирования, конструкция *try/catch* позволяет организовать структурированную обработку исключений. Если внутри защищенного блока *try* возникло *throw* исключение *exception*, то выполнение блока прерывается, а экземпляр исключения передается обработчику для данного класса исключения в одном из блоков *catch*. Выполнение программы продолжается после блока *try/catch*. Конструкция *try/finally* позволяет указать, что блок *finally* должен быть выполнен при выходе из блока *try*, в том числе при возникновении

исключения или возврате управления. После выполнения блока *finally* исключение выбрасывается повторно. Рассмотрим поведение *divide* с параметрами *A* и *B*, представленное на рис. 26.

а. Полагая модель согласованной (well-formed), перечислите операции класса *ArithmeticException*, переменные деятельности и обработчики исключений.

б. Пусть исключение класса *ArithmeticException* возникает при делении на ноль в действии *«a / b»*. Укажите порядок выполнения действий при выполнении деятельности с параметрами *a = 1, b = 0*.

в. Выпишите фрагмент кода на языке Java, соответствующий деятельности, представленной на диаграмме. Каким элементам языка соответствуют структурные действия?

г. Пусть класс *Throwable* является базовым для всех классов исключений. Используя переменные, блок *try/catch* преобразуйте в *try/finally*. Подсказка: исключения возникают в результате выполнения действия вида *RaiseExceptionAction*.

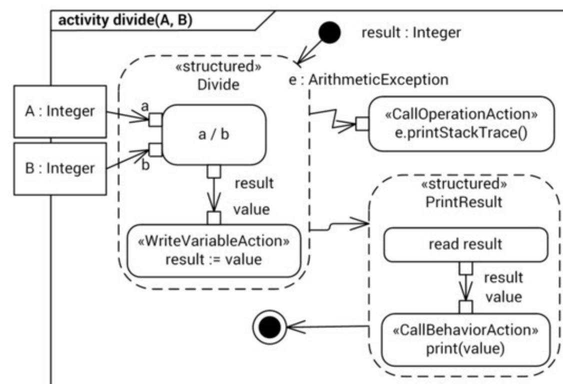


Рис. 26

§9. ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

ОСНОВНЫЕ ПОНЯТИЯ

Предметно-ориентированное проектирование (Domain-Driven Design, DDD) – совокупность принципов и практик объектно-ориентированного анализа и проектирования, направленных на создание моделей решения соответствующего потребностям заинтересованных лиц, в основе которых лежит принцип непосредственного соответствия модели системы понятиям предметной области.

Модель предметной области – упрощенное представление, абстракция предметной области, отражающая понятия, связи между ними, и представления модели в одной из известных нотаций. В настоящем разделе используется унифицированный язык моделирования UML.

Помимо модели предметной области существуют модели пользовательского интерфейса, хранения данных и другие, которые служит для представления соответствующих структур и в общем случае отличаются от модели предметной области.

Модель предметной области реализуется средствами конкретной объектно-ориентированной технологии и инкапсулируется в отдельном слое, называемом слоем бизнес-логики.

Принципы DDD заключаются в следующем. **Непосредственное соответствие понятий** предметной области элементам системы слоя бизнес-логики. Использование **единого языка (ubiquitous language)**, который служит для коммуникации между всеми заинтересованными лицами, прояснения понятий предметной области и построения модели предметной области.

Далее в данном разделе используются обозначения из профиля DDD, определенного в задаче 4.18. При определении понятий указываются также и названия стереотипов из профиля.

В рамках подхода DDD к анализу и проектированию используются следующие **виды классов**:

Сущность (entity) – это класс предметной области, экземпляры которого имеют уникальный идентификатор, обычно обладает собственным поведением.

Значение (value) – тип данных, экземпляр которого не обладает индивидуальностью, то есть экземпляр полностью определяется значениями структурных свойств. Зачастую значения неизменяемы.

Служба (domain service) – это интерфейс UML, определяющий поведение и обязанности, которые не относятся к сущностям или значениям, но являются важными для моделирования предметной области. При этом служба не имеет изменяемого состояния.

DDD определяет следующие структуры, виды классов и ограничения для управления жизненным циклом экземпляров и группировки экземпляров классов модели.

Модуль (module) – несколько классов предметной области, объединяемых для решения какой-либо общей задачи. В профиле DDD модуль – это стереотип пакета UML.

Агрегат (aggregate) – совокупность взаимосвязанных экземпляров классов модели, которые выступают как единое целое с точки зрения изменения состояния. **Корень агрегата (root) –** одна из сущностей агрегата, которая обеспечивает согласованность агрегата и предоставляет интерфейс для взаимодействия с агрегатом. Остальные классы агрегата являются внутренними. Границы агрегата определяются по связям между корнем агрегата, значениями и сущностями. Каждый экземпляр сущности относится только к одному агрегату.

Фабрика (factory) – класс предметной области, единственной обязанностью которого является создание экземпляров классов, составляющих агрегат, и конструирование из них агрегата.

Репозиторий (repository) – служба, которая используется для извлечения, сохранения и удаления сущностей.

ЗАДАЧИ

9.1. С целью разработки модели предметной области системы управления виртуальными машинами проводится интервью с пользователем системы и инженером центра обработки данных. Записанные высказывания приведены на рис. 27.

а. Перечислите способы, которыми можно прояснить смысл понятий предметной области. Примените один из способов для прояснения понятий «виртуальная машина» и «гипервизор», если эти понятия не знакомы.

б. Выделите в 1—7 предложения, которые не несут информации, относящейся к цели интервью, и выделите в каждом из оставшихся предложений понятия предметной области, среди них укажите в соответствии с основным принципом DDD классы и их виды.

в. Перечислите структурные свойства и операции класса виртуальной машины.

г. Если для группировки ВМ использовать пулы АР, с какими ограничениями столкнется пользователь? Приведите на диаграмме объектов пример нарушения данных ограничений.

д. Пусть в модели для группировки виртуальных машин использованы пулы АР. Согласно принципам DDD, должен ли пользователь использовать непосредственно класс пул АР, или предпочтительнее скрыть его за понятием «группа ВМ»? Объясните почему.

9.2. ЭКГ *Ecg* хранит сигнал *EcgSignal*, состоящий из массива целых чисел *values*. Процесс диагностики пациента по записанной ЭКГ *Ecg* предполагает очистку *clean* сигнала ЭКГ *EcgSignal* фильтром *Filter*, в результате чего создается новый экземпляр сигнала ЭКГ *EcgSignal*, прогнозирование математическими методами риска заболевания *analyze* анализатором *Analyzer* фильтрованного сигнала ЭКГ *EcgSignal*, и создание по результатам анализа *analyze* протокола *Protocol*. ЭКГ *Ecg* и протокол *Protocol* сохраняются и впоследствии могут просматриваться внешними клиентами.

а. Постройте модель предметной области. Для каждого класса модели определите его вид: сущность, значение или служба. Объясните выбор.

Пользователь	
1.	должна быть возможность управления виртуальными машинами (ВМ): создавать, удалять, включать и выключать ВМ.
2.	управление ВМ также включает изменение параметров ВМ.
3.	необходимо поддерживать группы ВМ с именами.
Инженер	
4.	над аппаратными ресурсами (АР) настроен гипервизор, с программным интерфейсом для управления ВМ.
5.	поддерживают и обслуживают гипервизор 2 инженера.
6.	параметры ВМ связаны с соответствующими АР: объем оперативной памяти, число процессоров.
7.	каждая ВМ относится к одному пулу АР с заданными объемом оперативной памяти и числом процессоров, ВМ использует ресурсы своего пула.

Рис. 27

б. Добавьте в модель вычисление *Computation*, связанное с протоколом *Protocol*, которое сохраняет информацию о процессе диагностики, в результате которого был получен протокол *Protocol*: какой фильтр *Filter* и анализатор *Analyzer* использовались, каков был сигнал ЭКГ *EcgSignal* после фильтрации.

в. К какому виду относится класс *Computation*? Почему?

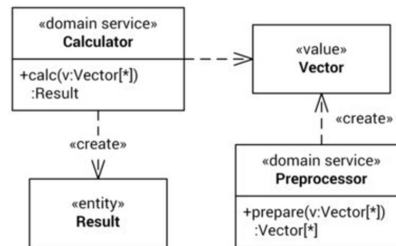


Рис. 28

9.3. На рис. 28 показана модель предметной области системы. Рассматривается процесс обработки данных, представленных в векторной форме.

а. Добавьте реализацию процесса обработки вычислением *Computation* (вид класса – значение), добавив необходимые отношения со службами предметной области, и определив поведения старта обработки *start* с параметром *Vector [*]*, подготовки *prepare*, расчета *calculate*. Последнее возвращает экземпляр результата *Result*.

б. Определите отношение между вычислением *Computation* и вектором *Vector* для поддержки описанного выше процесса обработки, не внося иных изменений в модель.

в. Добавьте реализации на псевдокоде операций вычисления *Computation*, не изменяя при этом ассоциации модели. Сколько всего экземпляров классов модели предметной области создается при выполнении всех трех методов?

г. Добавьте для мониторинга завершения процесса обработки внешними клиентами операцию *isFinished* в вычисление *Computation*. Добавьте ее реализацию на псевдокоде с минимумом необходимых изменений в модели, перечислите эти изменения.

д. Изменился ли вид класса вычисления *Computation* после внесенных в модель изменений? Почему?

9.4. В условиях задачи 9.2 со следующими дополнениями. Фильтр *Filter* и анализатор *Analyzer* работают с произвольными сигналами *CommonSignal*. При этом фильтр *Filter* требует знания о частоте дискретизации *frequency* сигнала. Для работы анализатора *Analyzer* это знание не требуется. При создании протокола *Protocol* анализатор *Analyzer* формирует список экземпляров класса риск заболевания *DiseaseRisk*, которые удовлетворяют медицинскому стандарту HL7[6]. Модель предметной области показана на рис. 29.

а. Объясните, почему фильтр *Filter* зависит от прибора *Device*, и почему эта зависимость является неудачным решением с точки зрения основной обязанности первого?

б. Определите модули DDD (тут и далее под модулем понимаются модули DDD предметной области), отнеся каждый класс модели к одному из них: математика *Math*, обработка сигналов *Signals*, медицина *Medicine*.

в. Устраните зависимость модуля обработки сигналов *Signals* от модуля медицины *Medicine* добавив в соответствующий модуль параметры сигнала *SignalParams* с информацией о частоте дискретизации.

г. Устраните зависимость между модулями математика *Math* и медицина *Medicine* добавив в соответствующий модуль прогноз *Prediction* как пару (строковый идентификатор класса прогноза *targetId*, численное значение прогноза *value*).

д. Классы в парах *SignalParams*, *DeviceParams* и *Prediction*, *DiseaseRisk* имеют схожую семантику, при этом устраняют зависимости между выделенными модулями. Для какого

[6] HL7 (медицинский стандарт) https://en.wikipedia.org/ wiki/Health_Level_7 (электронный ресурс, получено 17.07.2017).

одного класса нужно добавить в модель один или несколько парных в указанном смысле классов, чтобы выделенные модули оказались полностью независимыми?

е. Почему декомпозиция на модули по виду класса (сущности, значения, службы) неправильна?

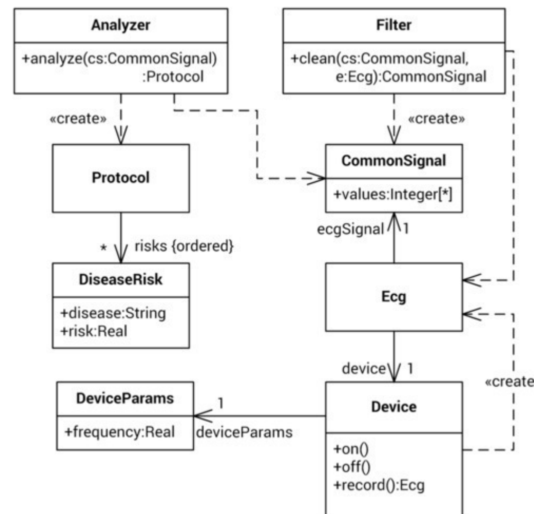


Рис. 29

9.5. Рассматривается процесс медицинской диагностики, который предполагает автоматическое создание протокола диагностики *Protocol*, который затем просматривается внешними клиентами, и добавление протокола *Protocol* в карточку пациента *MedicalRecord*. Протокол диагностики *Protocol* включает таблицу заболеваний *DiseaseTable*, которая составлена из рисков заболеваний *DiseaseRisk* (числовой характеристики риска заболевания согласно МКБ-10[7]).

а. Постройте модель предметной области и укажите вид каждого класса модели (сущность, значение, служба). Объясните выбор.

б. Укажите границы агрегатов и выделите корни каждого агрегата.

в. Добавьте класс модели текущее проверенное заболевание *VerifiedDisease*, который описывает подтвержденные врачом на настоящий момент времени заболевания *Disease* у пациента. Укажите вид класса, к какому агрегату он относится?

г. Добавьте в модель класс событие пациента *PatientEvent* с временем возникновения *dateTime* и описанием *description*, и укажите, что класс событие пациента *PatientEvent* создается при изменении врачом *Doctor* карточки пациента *MedicalRecord*. Укажите виды добавленных классов модели и агрегаты, к которым они относятся.

9.6. Категория *Category* имеет целочисленный идентификатор *id* и текстовое имя *name*. Категории *Category* организуются в древовидную иерархию, где каждая категория *Category* может иметь любое число потомков *children* и одного родителя *parent*.

а. Покажите модель предметной области согласно данному описанию на диаграмме классов.

б. Иерархия категорий хранится в реляционной базе данных в виде таблиц[8]. Постройте модель хранения данных для категорий *Category* используя паттерн Closure Table[9] для хранения связей *ClosureLink* между категориями *Category* в иерархии.

---

[7] МКБ-10, Международный классификатор болезней, https://en.wikipedia.org/wiki/International_Statistical_Classification_of_Diseases_and_Related_Health_Problems(электронный ресурс, получен 17.02.2017)

[8] В рамках данной задачи и в целях упрощения представления модель хранения данных описать в нотации классов UML, используя классы с суффиксом Table для обозначения таблиц, однонаправленные ассоциации – для обозначения связей

в. Для доступа к базе данных используется репозиторий категорий *CategoryRepository*. Определите в нем операции добавления *addChild* дочерней *child* к родительской *parent* категории *Category*, извлечения поддерева *subtree* категории *Category* глубины не более заданной *maxDepth*. Можно ли использовать в определении операций классы модели хранения данных? Объясните.

г. Реализуйте операции репозитория *CategoryRepository* в виде псевдокода при помощи экземпляра *dbContext* класса контекста базы данных *DbContext* (операции перечислены на рис. 30). Детали соединения с базой данных, подтверждение изменений, управление транзакциями можно опустить. Считать, что все ассоциации загружаются из базы данных автоматически.

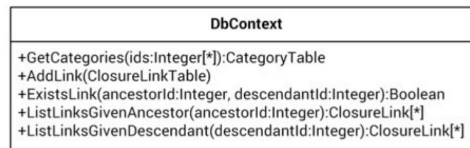


Рис. 30

д. Объясните, почему модель предметной области не переносится явно на структуру таблиц базы данных в данном случае? Объясните, почему транзитивное замыкание не представлено в модели предметной области? Резюмируйте, почему с точки зрения основной ответственности, в общем случае модели предметной области и хранения данных различаются.

9.7. Каждый заказ *Order* в интернет-магазине состоит из позиций *OrderItem*, которые определяют количество *amount* товара *Product*. Организуется маркетинговая кампания *MarketingCampaign* с выпуском купонов *Coupon*, которые при использовании уменьшают стоимость заказа *Order* на фиксированный процент. Продажи учитываются в отчете *DiscountedSoldItemsReport*, состоящем из проданных товаров с учетом скидки. Товар *Product* и заказ *Order* имеют штрих-коды (каждый свой) *Barcode*, используемые для идентификации.

а. Найдите несоответствия между описанием модели в условии и диаграммой (см. рис. 31), нарушения (согласно DDD) в определении видов классов модели, внесите в модель соответствующие исправления.

б. Найдите избыточные по отношению к описанию ассоциации модели, удалите их.

в. Найдите нарушения границ агрегатов.

г. Найдите несоответствия описанию в определение корней агрегатов, внесите исправления в определения корней и границ агрегатов.

по внешнему ключу. При этом в UML присутствует специальный профиль для описания реляционных моделей.

[9] Паттерн Closure Table описан в книге Karwin. B. SQL Antipatterns: Avoiding the Pitfalls of Database Programming и в ряде открытых источников. В основе паттерна лежит понятие транзитивного замыкания, что предполагает хранение в виде записей в таблице всех пар вершин дерева, между которыми существует путь.

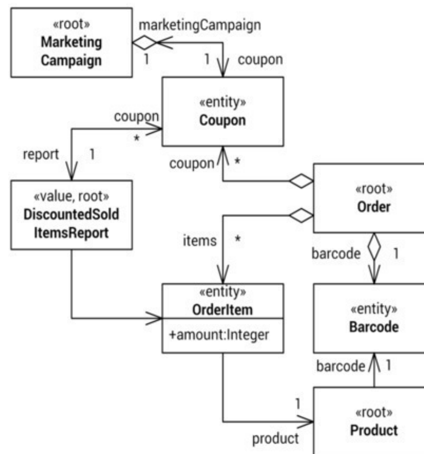


Рис. 31

9.8. На рис. 32 показана модель предметной области системы. Процесс построения на плоскости в декартовой системе координат многоугольника *Polygon* предполагает поочередное добавление вершин *Vertex* в список *vertices*.

а. Объясните, каким образом в модели может быть нарушено ограничение предметной области (инвариант) «Многоугольник *Polygon* умещается в области пространства определяемой системой неравенств: $|x| < 100$, $|y| < 100$»?

б. Дополните модель предметной области, чтобы указанный выше инвариант был выполнен.

в. Обеспечьте выполнение в модели инварианта «Многоугольник *Polygon* является правильным с числом вершин *vc*, радиусом описанной окружности *r* и центром в точке с координатами (*x, y)*» определением фабрики полигонов *PolygonFactory* с операцией *createRegular* и реализацией последней в виде псевдокода.

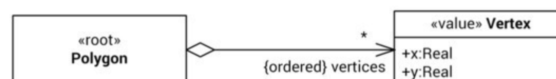


Рис. 32

г. В процессе построения правильного многоугольника *Polygon* фабрикой многоугольников *PolygonFactory* экземпляр агрегата правильного многоугольника *Polygon* может быть временно в несогласованном состоянии, так как вершины *Vertex* добавляются поочередно. Является ли это нарушением ограничений предметной области? Поясните ответ.

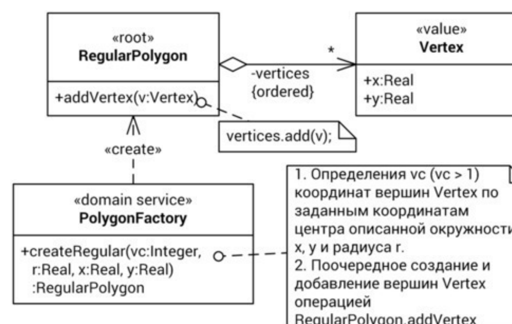


Рис. 33

9.9. Рассмотрим систему, модель предметной области которой показана на рис. 33.

а. Считая, что правильный многоугольник *RegularPolygon* реализует закрытые операции увеличения *expand* и уменьшения *reduce* числа вершин *vertices* на 1, определите и реализуйте псевдокодом операцию *resize* изменяющую число вершин *vertices* правильного многоугольника *RegularPolygon* в соответствии с параметром *vc*.

б. Как необходимо изменить модель, чтобы фабрика многоугольников *PolygonFactory* могла использовать операцию увеличения числа вершин на одну *expand* вместо операции *addVertex* при конструировании экземпляра правильного многоугольника *RegularPolygon*. В чем преимущества каждого подхода?

в. Можно ли в реализации операции *resize* правильного многоугольника *RegularPolygon* заменить использование операции *expand* использованием операции *addVertex*? Сформулируйте контекст применения операций *expand* и *addVertex*.

г. Сравните с точки зрения совмещения в классах модели поведения и данных полученную модель предметной области с моделью, приведенной в условии задачи 9.8, для которой регулярность многоугольника, операции конструирования и изменения числа вершин *resize* могут быть реализованы внешними клиентами. Почему классы без поведения нежелательны?

§10. МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ В СТРУКТУРЕ КЛАССОВ

ЗАДАЧИ

10.1. Структура производства самолетов для одной неизвестной авиакомпании приведена на рис. 34. Авиакомпания *Airline* закупает двигатели в *Engine Engineering*, корпус и оборудование – в *HullPlane Industries*.

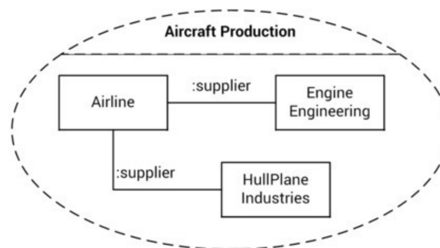


Рис. 34

а. Авиакомпания решила до начала производства консультироваться *estim ()* с фирмой *Aircraft Producer* для составления сметы *est*. Покажите поведение создания самолета на диаграмме, если авиакомпания ожидает по ее запросу завершения поставки корпуса *hull*, двух двигателей *engine* и оборудования *equip* самолета перед его самостоятельной сборкой *assemble*.

б. Стремясь оптимизировать время, авиакомпания решила передать сборку самолетов той же фирме *Aircraft Producer* так, чтобы не ждать поставок комплектующих, но получать уведомления о завершении этапов поставок и сборки. Воспользуйтесь паттерном Mediator.

в. Покажите на диаграмме, что авиакомпания оплачивает самолет, только если его стоимость *aircraft.price* равна цене, указанной в оценке *est.price*. При этом авиакомпания сначала отправляет платеж *pay ()* фирме *Aircraft Producer*, потом *HullPlane Industries*, потом в *Engine Engineering*.

г. В каком порядке указанные фирмы получат платежи? Ответ поясните.

10.2. Рассмотрим взаимодействие, происходящее при приеме пациента *patient* доктором *doctor* с заполнением карты *record*. Пациент несколько раз рассказывает жалобы *symptoms* доктору, а доктор несколько раз задает вопросы *questions*, в произвольном порядке, не дожидаясь ответа друг друга. Затем доктор заполняет карту *write*, сообщает пациенту диагноз *diagnosis* и передает рецепт *prescription*.

а. Покажите структуру взаимодействия и последовательность обменов сообщениями между его участниками. Воспользуйтесь подходящими фрагментами.

б. Врач, который проводит прием, уже как сотрудник *employee*, посещает летучки у ведущего врача *Lead*, на которых ведущий делает сотрудникам объявления и потом отвечает на один вопрос. Покажите структуру взаимодействия и порядок обмена сообщениями.

в. Зная, что у врача есть имя *name* и квалификация *rank*, предложите структуру классов для реализации указанных взаимодействий. Каким образом в данной ситуации применяются принципы ISP и SRP?

г. Опишите в виде структурированного текста варианта использования взаимодействие приема пациента врачом, полагая пациента актором, а врача и карту – элементами системы.

10.3. Помещение *Room* состоит из четырех поверхностей *Surface* как стен *walls*, одной как пол *floor* и одной поверхности как потолок *ceiling*. Маляр *Painter* умеет красить поверхности *paint*. Плиточник *Tiler* может укладывать их плиткой *tile*.

а. Примените паттерн Visitor так, чтобы маляр и плиточник по-разному работали с жилой комнатой *LivingRoom* и кухней *Kitchen*.

б. Покажите на диаграмме взаимодействие отделки квартиры из комнаты и кухни, когда сначала работает маляр, потом плиточник. В жилой комнате покрасить потолок и стены, на кухне уложить плиткой пол и стены, и покрасить потолок.

в. Изобразите кооперацию для отделки данной квартиры.

г. Рассчитайте метрики RFC и DIT для *Painter* и *LivingRoom*.

10.4. Рассмотрим программу для смартфона *Agent*, которая при встрече приветствует смартфоны собеседников согласно правилам этикета. По нажатию кнопки приветствия устанавливается подключение *connect* с другим смартфоном, затем выполняется приветствие, собеседники его получают *show*, и подключение завершается *close*.

а. Приведите диаграмму вариантов использования. Кроме этого, реализуйте варианты использования кооперацией *SimpleGreeting*, в которой участвуют субъект *subject* и объект *object* приветствия, представляющие собеседников в системе. Поясните взаимодействие в кооперации диаграммой, полагая все действия синхронными (требующими ожидания завершения).

б. Согласно правилам английского этикета двадцатых годов прошлого века, правильным официальным приветствием будет *«How do you do?»*, в ответ на которое может последовать *«How do you do?»*. Примените паттерн стратегия Strategy к субьектам, и паттерн, реализующий OCP для сокрытия базового алгоритма приветствия, покажите доработанные кооперацию и последовательность приветствия.

в. На бизнес встрече формальным приветствием будет *«Very glad to meet you»* или *«Very glad to see you»*, если встреча происходит в своем офисе, при этом для джентльменов принято обмениваться асинхронными *handshake*.

г. Доработайте модель, реализуйте приветствие на бизнес-встрече с несколькими партнерами. В трендах этого сезона рукопожатия с собеседниками следуют после приветствия их всех.

10.5. Маленькая рыбка *smallfish* класса *Fish* подплывает *floatNearby* близко к берегу моря *Sea*. Старик *OldMan* асинхронно забрасывает *throw* невод *seine* класса *Seine* в море *Sea*. Невод запрашивает у моря рыбу операцией *tryFish*, указывая себя в параметрах. Море асинхронно направляет *add* рыбку *smallfish* в невод *seine*. Старик вытягивает невод на берег операцией *pullAshore*.

а. Покажите, что после вытягивания невода, старик проверяет невод операцией *checkSeine*.

б. Добавьте еще одну рыбку *mediumfish*, она подплывает к берегу после маленькой рыбки и до забрасывания невода.

в. Используя фрагменты, доработайте модель взаимодействия и укажите, что море может передавать неводу как рыбу, так и тину *SeaScum*, если ни одна рыбка еще не подплыла к берегу.

г. (*) Изучите механизм разделения роли на части (part decomposition) при моделировании взаимодействий. Пусть невод, море и тина являются частями берега моря *SeaShore*. Представьте взаимодействие старика, берега моря и рыбки в том же сценарии, скрыв взаимодействие рыбки и старика с частями берега.

10.6. В переезд *RailCrossing* устанавливается *equipped* оборудование *Equipment*. Оборудование реализует операции открытия *open* и закрытия *close*. Консоль управления *Console* подключена в качестве контроллера *ctrl* к переезду и состоит *controls* из нескольких направлений движения *Way*. Консоль реализует операцию *addWay*, которая соединяет направление *way*

и экземпляры оборудования *eqs*, передаваемые в параметрах. Оборудование переезда может быть подсоединено только к одному из путей.

а. Добавьте светофор *TrafficLights*, семафор *RailLights* и шлагбаум *Barrier* как виды оборудования, и автоматическую и ручную виды консоли управления. Укажите, что оборудование и консоль управления могут быть только одного из указанных видов.

б. Выделите в отдельный класс *TwoWayRCBuilder* алгоритм создания переезда с ручным управлением, с направлением движения, оборудованным светофором и шлагбаумом, и направлением, оборудованным семафором. Примените паттерн Builder, включающий операции *addRailPath*, *addAutoRoad* и другие при необходимости.

в. Покажите алгоритм на диаграмме последовательности. Задание значение свойству считайте синхронной операцией, например, *setCtrl* для *ctrl*.

г. Приведите пример такого переезда. Решение поясните.

10.7. Кооперация обработка события *EventsHandling* включает событие *event*, источник *eventSource* и обработчик *eventListener*. У одного источника событий может быть несколько слушателей.

а. Отобразите данную кооперацию на диаграмме классов, используя соответствующие обозначения для кооперации.

б. (*) Используя представление взаимодействия, постройте для данной кооперации модель поведения обработки события управляющим компонентом плеера. Экземпляр *EngineSource* в роли *eventSource* генерирует событие и уведомляет о нем подключаемые модули *visPlugin* и *lyricsPlugin* в роли *eventListener*. Модуль *lyricsPlugin* обрабатывает событие, *visPlugin* игнорирует событие.

в. Реализуйте поведение менеджера событий с помощью кооперации *EventsHandling*. Отразите участие классов *EngineSource*, *EngineEvent* и *EngineListener* в кооперации *EventsHandling* с назначенными ролями *eventSource*, *event* и *eventListener* при условии, что каждый экземпляр *EngineSource* может быть связан с несколькими экземплярами *EngineListener*

10.8. Система управления движением поездов метро построена в архитектурном стиле Blackboard[10], поезда *Train* и семафоры *Semaphore* выступают агентами, связанными с общим событийным пространством *Blackboard* через порт типа *EventSpace*. Алгоритм управления поездом тормозит *brake* поезд до полной остановки *Steady* при получении события приближения к станции *station* с последующим открытием дверей *open*, и закрытием дверей *close*, разгоном и поддержанием *sustain* скорости *sp* поезда в движении *Moving* на команду об отправлении *go*. Параметры управления движением поездов и семафоров доступны компоненту управления *Control shell* через порт управления типа *Control* событийного пространства *Blackboard*.

а. Покажите на диаграмме компоненты системы, отношения между ними, а также передаваемые между компонентами сигналы и их параметры.

б. Представьте алгоритм управления поездом в виде схемы состояний. Переходное состояние торможения считайте вложенным в *Steady*.

в. Доработайте алгоритм управления поездом, укажите, что поезд должен останавливаться, если сигнал ближайшего семафора *state* в событийном пространстве красный, и продолжать движение после изменения сигнала на зеленый.

г. Добавьте в модель поезд с установленной системой кондиционирования. Помимо управления поездом, доработанный алгоритм может включить режим охлаждения *Cooling*, нагрева *Heating* или ожидания *StandBy* пока температура не отклонится от *comfy* на величину *val*. Агент датчик температуры *TempSensor* регулярно отправляет сигнал *temp* с параметром температуры *t*. Решение поясните.

---

[10] рхитектурный стиль Blackboard – это разновидность архитектуры, построенной на данных, в которой слабо связанные автономные агенты взаимодействуют посредством возбуждения событий в центральном репозитории (blackboard) [11].

10.9. Задача прототипа робота пылесоса – обходить прямоугольную комнату неизвестного размера и убирать пыль. Рассмотрим базовый алгоритм для пустой комнаты. Наложим на помещение сетку, робот может находиться в одной из клеток. Будем считать, что стены расположены по краям клеток. Робот сначала перемещается в левый нижний угол *ToDownLeft*, потом двигается по спирали *Spiral* в противоположный угол.

а. Проработайте алгоритм движения по спирали как машину Мура[11] по следующей схеме. Робот по одной клетке двигается вверх *Up* до получения уведомления о препятствии *obs*, делает шаг направо *Right*, если получает уведомление, что удачно *ok* – то двигается вниз *Down* до препятствия *obs*, иначе – заканчивает обход. Дойдя вниз до упора, робот идет направо. Если удачно – начинает виток спирали заново, иначе завершает обход помещения.

б. Предложите и реализуйте алгоритм перехода в левый нижний угол.

в. Соберите алгоритм обхода воедино. Добавьте уборку пыли: если после перемещения робот обнаруживает, что клетка требует уборки *dust*, то робот выполняет уборку *clean ()*.В начале работы робот проверяет текущую клетку. Чистые клетки не убирать.

г. Приведите пример последовательности событий, принимаемых роботом (робот заканчивает уборку и возвращается), и пример отвергаемой (поведение робота не завершается).

д. На основе робота с описанным поведением реализуйте робота, который возвращается в исходную позицию по завершению уборки за минимальное количество движений и прекращает работу. Покажите решение на диаграмме классов и диаграмме состояний.

е. Решите предыдущий пункт, добавив нового робота и не внося изменения в базового робота.

ж. (*) Прочитайте про различия между делегированием (delegation) поведения (или обязанностей) и перенаправлением (forwarding) вызовов операций. Поясните, почему применение паттерна Decorator для решения предыдущего пункта недопустимо?

10.10. Для регистрации на портале общества оригинальных проектов (ООП) нужно создать учетную запись *Account*, ввести личные данные и пройти проверку личности. До создания учетной записи все посетители портала считаются временными *Anonym* и уничтожаются при завершении сессии с порталом *sessionFinished*. Начав процесс создания учетной записи *register*, посетитель вводит свое имя *name* и электронную почту *email*. На почту отправляется письмо *sendEmail*, при переходе по ссылке из письма *verifyEmail* завершается создание учетной записи. Если не перейти по ссылке в течение суток, то создание учетной записи отменяется.

а. Покажите процесс создания учетной записи на диаграмме. После создания учетной записи посетителю становятся доступны вход *signin* и выход *signout* с портала.

б. Посетитель с учетной записью может продолжить регистрацию *verify* для получения подтвержденной учетной записи. Он вводит номер паспорта *pass* и адрес *addr*. Если выбрано подтверждение личности по почте *confirmMail*, то портал отправляет письмо с кодом *sendMail* по указанному адресу. Если затем введен код *pin*, то учетная запись становится подтвержденной, иначе по истечении 30 дней с отправки письма потребуется заново подтверждать учетную запись.

в. Реализуйте личное подтверждение учетной записи по предъявлению оригинала паспорта *confirmPass* в центре обработке данных. Добавьте возможность прервать *abort* подтверждение учетной записи в любой момент.

г. Покажите на диаграмме класс учетной записи со всеми операциями и сигналами.

10.11. Процедура моделирования с помощью карточек CRC начинается с выделения кандидатов классов *ElicitCandidates*. После этого, выбирается *SelectScenario* текущий сценарий *cs: Scenario*. Из сценария берется следующий шаг для анализа *AnalyzeStep*, полученные обязанно-

[11] В конечном автомате Мура (Moore) выполняемые действия определяются только состоянием, в котором находится автомат. В схеме состояний UML2 это эквивалентно указанию действий только при входе в состояние.

сти *resps* назначаются *Assign* классам. Если шагов не осталось, берется следующий сценарий, если сценариев больше нет, процедура моделирования завершается.

а. Покажите, что актуальное описание классов поддерживается в объектном узле вида *«centralBuffer»* и обновляется после каждого назначения обязанностей. Для сохранения текущего сценария между анализом шагов используйте объектный узел *«datastore»*.

б. Укажите в модели, что для перехода к следующему шага сценария *Step* необходим предыдущий шаг. Первый шаг сценария может быть выбран без предыдущего. Выбранный шаг сценария используется также при назначении обязанностей, отразите это в модели.

в. Доработайте модель, укажите, что именно менеджер продукта *ProductOwner* выбирает следующий сценарий, а после рассмотрения всех шагов всех сценариев проектировщик *Designer* на основе всех обязанностей строит по классам диаграмму *Diagram*, которая и является результатом процедуры. Остальные действия выполняет команда *Team*.

г. Представьте структуру взаимодействий в виде кооперации команды, проектировщика и менеджера продукта, примените паттерн Mediator. Какими сообщениями будут обмениваться участники взаимодействия? Покажите взаимодействие на диаграмме последовательности.

ГЛАВА 3. УКАЗАНИЯ, ОТВЕТЫ И РЕШЕНИЯ ЗАДАЧ

§11. МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ

1.5.

Решение задачи. Прежде всего, прочитаем условие задачи и представим описанную в нем модель на диаграмме классов. В условии сказано, что в модели присутствует интерфейс доступа к элементам коллекции, который называется *Collection*. Далее говорится, что интерфейс работы со списками с названием *List* уточняет интерфейс *Collection*, и наоборот, интерфейс *Collection* обобщает интерфейс *List*. Значит, в модели также присутствует отношение обобщения между интерфейсами *List* и *Collection*. Представим имеющиеся три элемента на диаграмме. Интерфейс является особым видом классификатора, изображается в виде прямоугольника с ключевым словом *interface* в двойных угловых кавычках. Оба интерфейса не имеют операций или атрибутов, поэтому соответствующие секции на диаграмме не показаны. Отношение обобщения указывается треугольной стрелкой, направленной от уточняющего к более общему элементу.

Далее в условии сказано, что в модели определен класс с названием *BaseCollection*, который является абстрактным. Класс реализует интерфейс *Collection*, значит в модели определено отношение реализации интерфейса *Collection* классом *BaseCollection*. Аналогично, в модели определен класс *BaseList*, отношение обобщения между классами *BaseList* и *BaseCollection*, и отношение реализации интерфейса *List* классом *BaseList*. Модель, описанная в условии задачи, представлена рис. 35. Класс является классификатором, изображается в виде прямоугольника, название абстрактного класса указывается курсивом, как для класса *BaseCollection*, или ключевым словом *abstract*, следующим после названия класса.

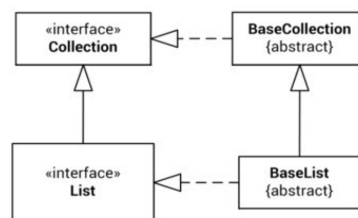


Рис. 35

Чтобы определить, каким образом писать ключевые слова: в кавычках или фигурных скобках, можно использовать простое правило. Ключевое слово *«interface»* пишется в двойных кавычках потому, что указывает на метакласс *Interface*, определенный в *UML*, в то время как *{abstract}* указывает на значение свойства *isAbstract* метакласса *Class*. Теперь перейдем к вопросам по задаче.

В пункте а. нужно добавить в модель класс *ArrayList*, который реализует операции работы со списками, при этом конкретные операции не перечислены. Кроме того указано, что необходимо использовать свойства наследования. Последнее означает, что мы должны добавить отношение обобщения между классом *ArrayList* и одним из элементов модели. Так как класс *ArrayList* реализует операции работы со списками, как и класс *BaseList*, мы укажем, что класс

ArrayList уточняет класс *BaseList* в том, что использует массив для хранения элементов. Таким образом, ответом на вопрос а. будет диаграмма классов модели, представленная на рис. 36.

Перейдем к вопросу б. Сказано, что в интерфейс *List* добавлена операция *get* для получения элемента списка в зависимости от некоторого параметра *k*. Вообще говоря, реализовать данное условие можно по-разному, но наиболее простым будет определить в операции *get* параметр *k*. Заметим, что тип возвращаемого значения не указан, в то же время на диаграмме нельзя указать, что операция возвращает значение, не указав его тип. Для разрешения данной ситуации придумаем подходящее название ListElement и укажем его в качестве типа возвращаемого значения. Диаграмма классов модели приведена на рис. 37.

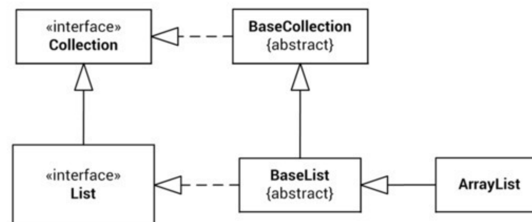


Рис. 36

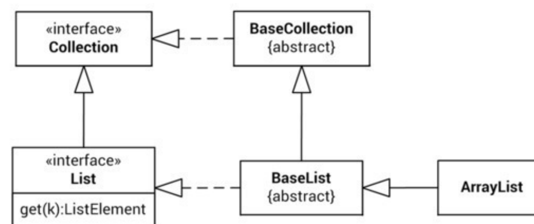


Рис. 37

Перейдем ко второй части вопроса. Если в интерфейсе *List* определена операция *get*, спрашивается, в каких классах операция также должна быть определена, при этом предполагается, что модель является согласованной (well-formed). Так как класс *BaseList* реализует интерфейс *List*, то он должен включать операцию *get*, в противном случае нарушаются требования отношения реализации интерфейса. Отношение обобщения, напротив, не указывает на необходимость определения операции *get* в классе *ArrayList*, но и не запрещает его. Следовательно, ответом на вопрос б. будет: «операция *get* должна быть определена в классе *BaseList* потому, что отношение реализации интерфейса требует определения всех черт интерфейса, в том числе операции *get*».

Пункт в. оставляется на самостоятельную проработку. Указанное в условии поведение может быть определено в любом из классов модели.

2.2.

Решение задачи. В условии указан основной актор *Author*, который взаимодействует с системой в варианте использования *SendPaper*. Вспомогательным актором является редактор *Editor*. Получив статью, редактор инициирует взаимодействие в варианте использования *Review*, в котором также участвует еще один актор – рецензент *Reviewer*. Так как взаимодействие в рамках *SendPaper* приводит к выполнению рецензирования *Review*, между этими вариантами использования имеется отношение включения.

Диаграмма вариантов использования представлена на рис. 38. Автор не участвует в рецензировании, поэтому ассоциация актора *Author* с вариантом использования *Review*

не указана. Отношение включения указано пунктирной стрелкой в направлении от базового варианта использования *SendPaper* к включаемому *Review*.

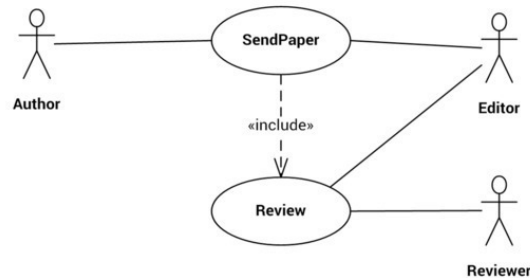


Рис. 38

В пункте а. в модель нужно добавить возможность подготовки статьи к публикации с участием автора и корректора. Добавляемая возможность моделируется вариантом использования *PrepareFor-Publishing*, в котором участвуют актор *Author* и новый актор *Proofreader*. При этом не указывается, каким именно образом происходит подготовка статьи к публикации. Ответом по пункту а. будет диаграмма, представленная на рис. 39.

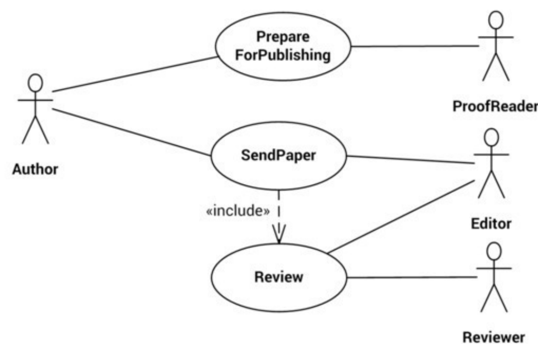


Рис. 39

Перейдем к пункту б. В условии пункта указано, что подготовка статьи к публикации, то есть вариант использования *PrepareFor-Publishing*, выполняется только при определенном условии: статья была одобрена редактором. При этом не указано, что подготовка статьи к публикации необходима для рассмотрения статьи, поэтому правильно будет указать, что вариант использования *PrepareForPublishing* расширяет вариант использования *SendPaper*. В результате получим модель, приведенную на рис. 40. Для того, чтобы правильно показать отношение расширения, в вариант использования *SendPaper* была добавлена точка расширения *Decision*. Для отношения расширения указано условие *Condition*, в фигурных скобках приведено логическое выражение, проверяемое в точке расширения. Если выражение истинно, то расширение происходит, если ложно, то расширения не происходит. Приведенная форма записи отношения расширения является полной. В случае, когда у варианта использования имеется только одна точка расширения или условие расширения несущественно для целей моделирования, можно опустить соответствующие элементы модели.

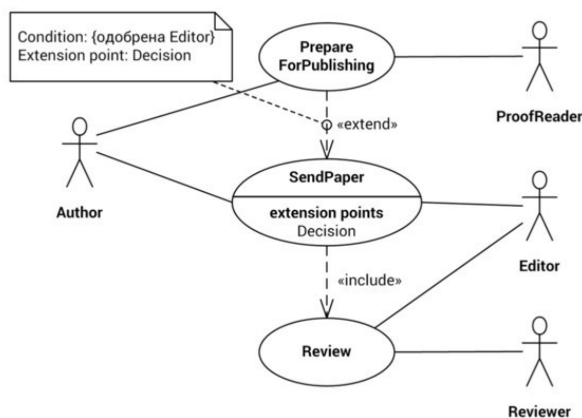


Рис. 40

3.1

Решение задачи. В условии задачи говорится, что в модели определена кооперация *Sale*, в которую роли *Salesman* и *Customer* входят как составные части. Представим модель на диаграмме внутренней структуры (рис. 41).

Кооперация изображается эллипсом с пунктирной линией, роли *Salesman* и *Customer* представлены прямоугольниками.[12]

В пункте а. необходимо показать, что роли взаимодействуют между собой в рамках данной кооперации. Для этого нужно добавить соединитель между *Salesman* и *Customer*, так как по определению, соединитель указывает возможные пути взаимодействия. В условии не говорится о типе соединителя и ролей кооперации, поэтому ответом по пункту а. будет диаграмма, представленная на рис. 42. Соединитель между ролями изображен сплошной линией.

Перейдем к пункту б. В условии сказано, что необходимо создать новую кооперацию *BrokeredSale*, в которой осуществляется продажа с посредником, и уточняется, что посредник представляется покупателю продавцом, а продавцу покупателем. В пункте а. была построена кооперация взаимодействия покупателя с продавцом, поэтому, следуя указанию использовать вхождение кооперации, определим кооперацию *BrokeredSale* так, чтобы она включала два вхождения кооперации *Sale*: одно для взаимодействия продавца с посредником, а второе для взаимодействия покупателя с посредником. Полученная кооперация представлена на диаграмме рис. 43.

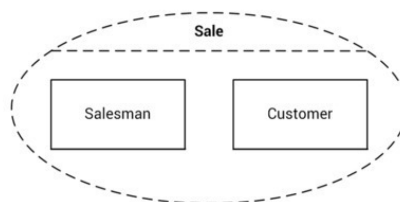


Рис. 41

[12] Согласно спецификации, роли в кооперации collaborationRole являются подмножеством ролей в структурном классификаторе, метаклассе, который уточняет кооперация. В структурном классификаторе прямоугольником со сплошной линией обозначаются только части, т.е. роли, которые классификатор включает через композицию. Остальные роли изображаются прямоугольниками с пунктирной линией. Тем не менее, в спецификации роли в кооперации также изображаются прямоугольниками со сплошной линией.

Роль *customer* при вхождении кооперации *Sales* связана с ролью *Broker* кооперации *BrokeredSale*, роль *salesman* – с ролью *Salesman*. Роль *salesman* при вхождении кооперации *Customers* связана с ролью *Broker* кооперации *BrokeredSale*, а роль *customer* – с ролью *Customer*.

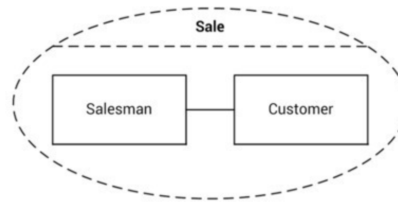


Рис. 42

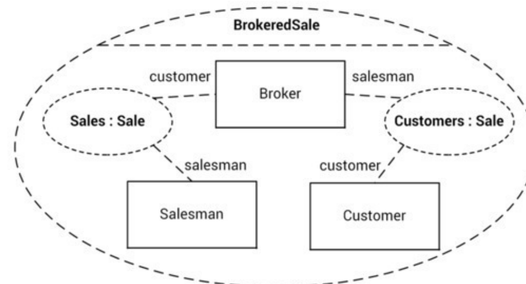


Рис. 43

3.3

Решение задачи. Условие задачи описывает взаимодействие и начинается с представления роли автора, который отправляет сообщение другому участнику – редактору. Указано, что автор ожидает подтверждения о получении, значит, сообщение является синхронным и подтверждение следует моделировать ответным сообщением. После отправки подтверждения автору, редактор отправляет сообщение рецензенту. Здесь не указано, что редактор ожидает результата или подтверждения, поэтому сообщение является асинхронным. После получения сообщения от редактора, рецензент направляет редактору сообщение с результатами рецензирования, которое также является асинхронным.

Обратите внимание на порядок возникновения событий в задаче. При изложении условия задачи подразумевается, что возникновение событий, описываемых в следующем предложении, происходит позднее всех событий для упоминаемых участников ранее.

Далее, редактор после получения сообщения с результатами от рецензента направляет сообщение автору и, после этого, сообщение с благодарностью рецензенту. Диаграмма последовательности построенной модели приведена на рис. 44.

Укажем общее правило, согласно которому следует добавлять в модель спецификации исполнения. Если в результате получения сообщения предполагается обработка или исполнение поведения участником взаимодействия, то следует добавить спецификацию исполнения. Если же такой обработки не предполагается или она несущественна для целей моделирования, спецификацию исполнения на диаграмме можно опустить. Также обратите внимание, что начало спецификации исполнения совпадает с получением обрабатываемого сообщения, и отправка ответного сообщения совпадает с завершением обработки. Поэтому во многих случаях ответные сообщения могут быть на диаграмме не показаны.

Перейдем к пункту а. В условии задачи не указывается контекст взаимодействия. Следовательно, допустимо предполагать, что взаимодействие происходит в контексте некоторой

кооперации. Значит, в пункте а. требуется построить модель внутренней структуры класса кооперации, в контексте которой могло бы быть определено описанное в задаче взаимодействие.

Взаимодействие происходит в контексте кооперации, значит, линиям жизни соответствуют роли в кооперации. В отсутствие дополнительных указаний, будем полагать, что роли соответствуют свойствам кооперации, а не параметрам или переменным.

Далее указано, что во взаимодействии участвуют пять рецензентов. Данное условие соответствует значению кратности соответствующего структурного свойства в кооперации. Ограничение, что статья направляется одному из пяти рецензентов, выполняется по-умолчанию. Так как между линиями жизни происходит обмен сообщениями, то между соответствующими ролями в кооперации создается сборочный соединитель.

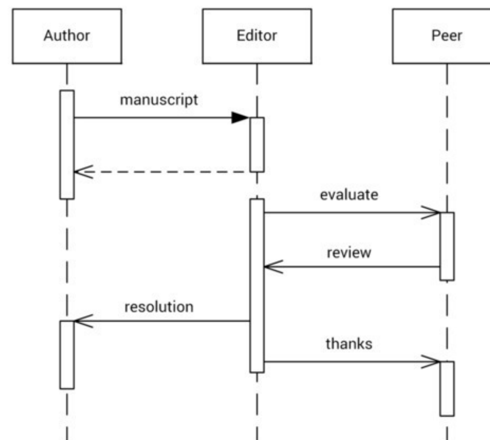


Рис. 44

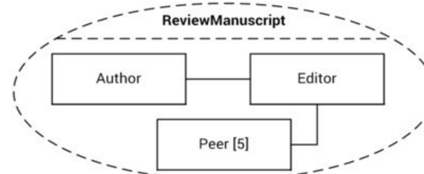


Рис. 45

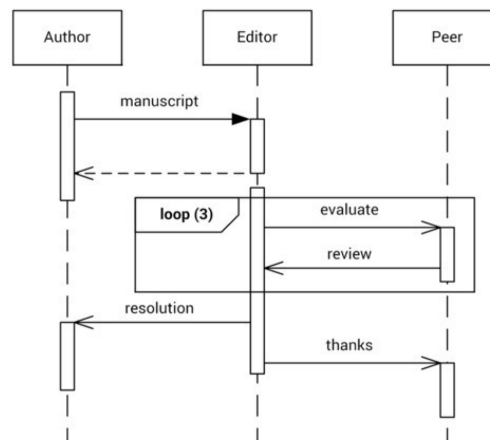


Рис. 46

Ответом по пункту а. будет диаграмма, представленная на рис. 45.

Перейдем ко второму пункту. В текущей модели, статья направляется только одному рецензенту. В условии сказано, что статью следует направить трижды, при этом не указано, кому из рецензентов. Для этого достаточно применить комбинированный фрагмент с оператором цикла и указать три итерации. В более сложном случае, когда требуется направить статью трем разным рецензентам, можно использовать ограничения или явно указать с помощью селектора, какому именно рецензенту направляется статья. Измененная модель взаимодействия представлена на рис. 46.

Следует заметить, что в данном случае статья направляется на рецензию следующему рецензенту только после получения результата от предыдущего.

Решение третьего пункта предоставляется читателю.

7.2

Решение задачи. В условии указан классификатор *TrafficLights* – светофор, поведение которого описывается конечным автоматом. После создания экземпляра светофора, автомат переходит из начального псевдосостояния в состояние выключен *Offline*. Переход выполняется безусловно, эффект при переходе отсутствует. Далее упоминается композитное состояние *Online*, в которое вложено состояние *Green*. При получении события *On* светофор должен перейти в состояние *Green*. Поэтому добавляем переход из состояния *Offline* в *Green*. О дополнительных условиях перехода ничего не сказано, поэтому сторожевое условие не указываем. Заметим, что вид события *On* не указан, поэтому можно полагать его событием получения сигнала.

Далее в условии указано, что по истечении пятидесяти секунд автомат переходит в состояние *Yellow*. Значит, в модель нужно добавить переход из состояния *Green* в состояние *Yellow*, также вложенное в *Online*, и указать триггер с событием времени. Последнее возникает по истечении интервала времени равного 50 секундам от момента, когда состояние *Green* стало активным в последний раз. Аналогично, добавляем переход из *Yellow* в состояние *Red* и указываем событие времени с интервалом в три секунды, и переход из *Red* в *Green* с интервалом в пятьдесят секунд. Диаграмма построенной схемы состояний приведена на рис. 47.

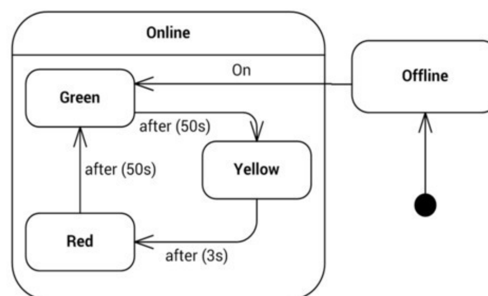


Рис. 47

Перейдем к пункту а. Нужно добавить возможность выключить светофор. По аналогии с включением добавляем переход из состояния включено *Online* в состояние выключено *Offline*, указываем приведенное в условии событие *Off*, которое можно полагать событием получения сигнала. Ответом в пункте а. будет модель, представленная на рис. 48.

В пункте б. задача немного усложняется. Необходимо обеспечить возможность настройки интервалов между сигналами светофора, в то время как в текущей модели интервалы заданы константами. Для этого в контекстный классификатор *TrafficLights* добавим три атрибута для хранения продолжительности сигналов светофора: *green* для состояния *Green*, *yellow* и *red* для состояний *Yellow* и *Red* соответственно. Заменим в триггерах переходов между

вложенными состояниями в *Online* константы на выражения времени, извлекающие значения из введенных ранее атрибутов.

По условию, длительности интервалов сигналов светофора, а значит добавленных атрибутов, следует задавать, когда светофор находится в состоянии *Offline*. Как отразить это в конечном автомате? Вспомним, что автомат описывает поведение классификатора и вызываемое в нем поведение в ответ на события. Поэтому автомат описывает, в том числе, какое поведение будет исполнено при получении сигнала или вызове операции (если явно не указано в поведении, что оно реализует данную операцию вне зависимости от состояния классификатора). Добавим в состояние *Offline* внутренние переходы в себя с триггером по событию вызова операций *setGreen*, *setYellow* и *setRed* с параметром *t*. Каждому из переходов укажем поведение в качестве эффекта – присваивание соответствующему операции атрибуту значения *t*. Построенная модель представлена на рис. 49.

Следует обратить внимание на следующие моменты. Во-первых, вызов операций *setGreen*, *setRed* и *setYellow* будет игнорирован, если светофор не выключен, так как поведение, вызываемое при приеме события вызова этих операций, указано только в состоянии *Offline*. Во-вторых, для обработки вызовов операций в модели использованы внутренние переходы, а не внешние переходы в себя, что можно считать хорошим тоном в моделировании.

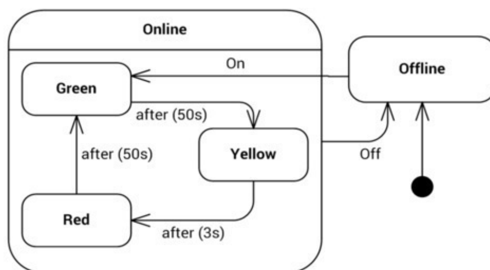


Рис. 48

Действительно, выполнение внешнего перехода, помимо выполнения эффекта, приводит к прерыванию действия, выполняемого в состоянии, выполнению действия при выходе, выполнению действия при входе в состояние, обновлении времени входа в состояние, что влияет на возникновение событий времени для триггеров, определенных в данном состоянии. Все эти моменты необходимо учитывать при использовании внешних переходов в себя для указания поведения, выполняемого при вызове операций. Решение пункта в. предоставляется читателю.

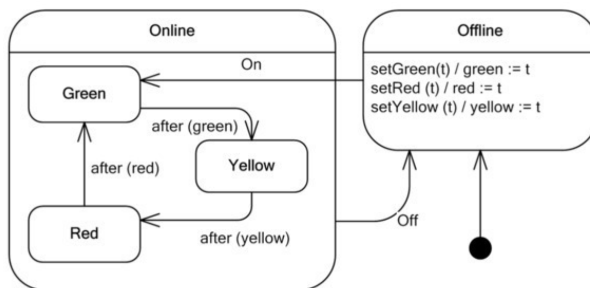


Рис. 49

8.2

Решение задачи. В условии задачи рассматривается деятельность, первым действием которой является нечеткое действие *AddItem*. Покажем, что действие *AddItem* выполняется в деятельности первым. Параметры или контакты действия не указаны, поэтому для выполнения действия достаточно получение метки управления. Добавим в модель начальный узел деятельности и укажем, что поток управления из начального узла попадает в действие *AddItem*.

После выполнения действия *AddItem*, по условию, вычисляется выражение *ready*, детали которого не указаны. Если выражение истинно, то выполняется действие *ShipOrder*. В условии ничего не говорится о контактах действия, поэтому для вызова достаточно получение действием метки управления. Добавим в модель управляющий узел выбора и нечеткое действие *ShipOrder*. Свяжем действие *AddItem* управляющим потоком с узлом выбора. Добавим поток управления из узла выбора в действие *ShipOrder* и укажем сторожевое условие *[ready]*, которое обеспечивает выполнение условия *ready* при передаче метки управления действию *ShipOrder*.

По условию, после выполнения действия *ShipOrder* деятельность должна завершиться. Добавим завершающий узел деятельности и входящий в него поток управления из действия *ShipOrder*.

Диаграмма деятельности для построенной модели представлена на рис. 50. Элементы на диаграмме деятельности принято размещать в предполагаемом порядке исполнения сверху вниз или слева направо. Поток управления, по которому передаются метки в случае невыполнения остальных сторожевых условий на управляющем узле выбора, указывается на диаграмме ключевым словом *else*. На диаграмме видно, что у действия *AddItem* два входящих потока управления. Тем не менее, вследствие того, что метки управления, приходящие по разным потокам неразличимы, для исполнения действия достаточно метки управления только по одному из входящих потоков управления. Приведенная на диаграмме нотация является более краткой, чем использование дополнительного управляющего узла слияния между начальным узлом деятельности и действием *AddItem*. Для того, чтобы показать, что действию для исполнения необходимы метки по двум потокам управления можно использовать различные контакты, тогда и потоки будут различны, или управляющий узел объединения для синхронизации двух потоков управления. В первом случае контакты потоков управления можно использовать для выбора набора контактов для вызова действия.

Перейдем к пункту а. В условии сказано, что когда выполнено условие *[ready]*, должно быть выполнено действие *SendInvoice*. Условие проверяется после завершения действия *AddItem*, поэтому действие *SendInvoice* должно выполняться после *AddItem* и метку управления должно получать по потоку управления, исходящему из узла выбора и отмеченному сторожевым условием *[ready]*. Значит, метка, передаваемая по этому потоку, должна приводить к исполнению действий *ShipOrder* и *SendInvoice*. При этом в условии указано, что эти действия исполняются независимо. Добавим управляющий узел разделения так, чтобы в него попадал поток управления *[ready]* из узла выбора. Из узла разделения направим по одному потоку управления в действия *ShipOrder* и *SendInvoice*. Во второй части пункта указано, что деятельность не завершается, пока не будет отправлен счет *SendInvoice*. Для этого, добавим между *ShipOrder* и завершающим узлом деятельности управляющий узел объединения и направим в него исходящий из *SendInvoice* поток управления. Таким образом, деятельность завершится только после того, как будут завершены действия отправки счета и доставки заказа. Решение пункта а. представлено на рис. 51.

Перейдем ко второму пункту задачи. В условии указано, что после добавления элемента в заказ действием *AddItem* вычисляется еще два выражения: *continue* и *cancel*. Если значение выражения *continue* истинно, деятельность возвращается к добавлению элемента в заказ. Если значение *cancel* истинно, деятельность завершается. Итак, после завершения действия *AddItem* проверяются три условия. Предполагая, что модель является согласованной, вместо одновременной проверки условий можем проверить сначала значение *ready*, а затем, если оно ложно,

проверить значения *continue* и *cancel*. Добавим в модель управляющий узел выбора и направим в него поток управления со сторожевым условием *[else]*. Отсюда, добавим поток управления со сторожевым условием *[continue]* и направим его в действие *AddItem*, и поток управления с *[cancel]* направим в завершающий узел деятельности. Результат приведен на рис. 52. Для удобства на диаграмме использовано два завершающих узла. Следует заметить, что в согласованной модели вычисление выражений, стоящих в сторожевых условиях не приводит к побочным эффектам, поэтому порядок вычислений не важен. Пункт в. оставляется для самостоятельного решения.

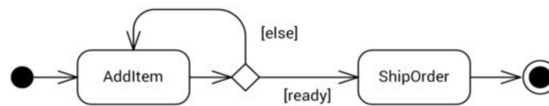


Рис. 50

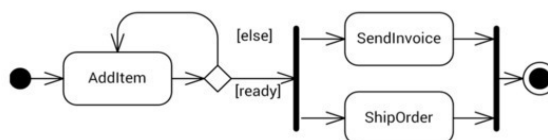


Рис. 51

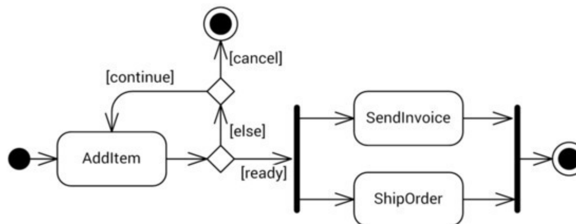


Рис. 52

§12. ПРИМЕРЫ ЛАБОРАТОРНЫХ И КОНТРОЛЬНЫХ РАБОТ

Практические работы помогают развивать навыки применения языка UML, осваивать объектно-ориентированное проектирование и моделирование с помощью задачника.

Часть работ направлена на развитие навыков понимания и интерпретации диаграмм на языке UML. Важным является освоение нотации часто используемых диаграмм классов, вариантов использования и взаимодействия.

Далее предлагаются практические работы, посвященные техникам, методам, паттернам, принципам объектно-ориентированного проектирования. Учебные задачи из этих практических работ позволяют отработать навыки, которые будут полезны при решении реальных задачам, возникающих на практике.

Оставшаяся часть работ посвящена моделированию с помощью языка UML. Навыки, полученные при решении задач этой группы, помогут при получении сертификатов по языку UML2.

12.1. Нотация UML. Классы и объекты

Цель практической работы – освоить базовый синтаксис UML для моделирования структуры. Вы освоите элементы нотации диаграммы классов: классы, атрибуты классов, методы, отношения между классами. Условия задачи часто используют названия элементов языка UML.

12.2. Нотация UML. Варианты использования

Выполнив эту практическую работу, Вы изучите синтаксис UML диаграмм использования. Используются следующие элементы нотации диаграммы использования: варианты использования, акторы, ассоциации, отношение обобщения, отношения включения и расширения вариантов использования.

12.3. Нотация UML. Моделирование поведения

Практическая работа посвящена моделированию поведения с применением UML. Рассматриваются элементы нотации диаграмм взаимодействия (последовательности), состояний (конечный автомат), деятельностей. Задачи предполагают выбор наиболее подходящего типа диаграммы.

12.4. ООП. Анализ предметной области, идентификация классов

Эта практическая работа содержит задачи на выделение классов и построения модели предметной области. Для успешного решения задач требуется обосновать выделение классов, например, применив один из методов: Аббота (Abbot method), именных групп, CRC или метод шаблонных классов.

12.5. ООП. Базовый уровень

Цель практической работы – отработать шаги моделирования от требований до полной модели. Для решения этой практической работы требуется освоить моделирование использования, структуры и поведения. Решение задач этой практической работы позволяет успешно выполнять шаги метода проектирования ICONIX на практике.

12.6. ООП. Принципы проектирования SOLID

Цель работы – отточить применение принципов SOLID для выделения классов, назначения ответственности классам, построения иерархии классов. Задачи позволяют лучше понимать ситуации и порядок применения принципов проектирования на практике.

12.7. ООП. Паттерны проектирования

В этой практической работе собраны задачи, для корректного решения которых требуется применить один из паттернов проектирования. Вы сможете применить наиболее часто используемые паттерны в задачах, приближенных к реальным.

12.8. ООП. Предметно-ориентированное проектирование. Анализ предметной области

Базовыми понятиями DDD являются: сущность, значение, служба и модуль. В этой практической работе собраны задачи, которые предполагают анализ и моделирование предметной области при помощи базовых понятий. Решение данных задач позволит понять принципы DDD, научиться выражать элементы предметной области через базовые понятия и понимать разницу между ними.

12.9. ООП. Предметно-ориентированное проектирование. Применение паттернов

В этой практической работе представлены задачи на применение основных паттернов DDD, к которым относятся: агрегат, фабрика и репозиторий. Задачи также иллюстрируют достаточно типичные практических вопросы, которые тесно связаны с DDD: наличие и поддержку различных моделей – предметной области и хранения, согласованность состояния, поддержку инкапсуляции и «дружественные» отношения между классами.

12.10. Моделирование с помощью UML. Создание сложных моделей из нескольких диаграмм

В этой работе собраны задачи, развивающие навыки использования разных представлений (диаграмм) для построения модели системы. Задачи предполагают разработку структурной модели, построения связанной с ними модели поведения, а также понимание свойств полученной модели.

12.11. Моделирование с помощью UML. Расширенные возможности диаграмм классов

Часто отношения между частями системы имеют сложную семантику, и богатство языка UML позволяет выразить очень тонкие аспекты. В этой практической работе собраны задачи, для решения которых требуется применить продвинутые техники моделирования структуры с помощью UML.

12.12. Моделирование с помощью UML. Реализация вариантов использования

Задачи этой практической работы отрабатывают моделирование взаимодействия классов при реализации вариантов использования с помощью коопераций. Успешное освоение задач из этого раздела позволит лучше понять связь между структурной моделью, вариантами использования и их реализацией во взаимодействиях классов.

12.13. Моделирование с помощью UML. Применение конечных автоматов для описания поведения

Представление (viewpoint) схем состояний – мощное выразительное средство, с помощью которого можно описать интерфейс системы, поведение системы или объекта, зависящее от прежних состояний, асинхронность и параллелизм. Задачи в этой практической работе покрывают разные возможности диаграммы состояний.

12.14. Моделирование с помощью UML. Моделирование бизнес-процессов диаграммами деятельности

Деятельности в UML являются механизмом моделирования, с помощью которого можно выразить бизнес-процессы. Умение моделировать такие процессы позволяет проводить симуляцию и анализ до этапов реализации и внедрения.

12.15. Моделирование с помощью UML. Реализация алгоритмов диаграммами деятельности

В этой практической работе собраны задачи, позволяющие отработать моделирование вычислительных алгоритмов с помощью диаграмм деятельности.

12.16. Моделирование с помощью UML. Архитектурное проектирование

В этой практической работе собраны задачи, требующие выделения крупных компонентов системы, размещения этих компонентов на узлах выполнения. Успешное освоение задач позволит представлять структуру сложных систем средствами UML.

Таблица 1. Состав практических работ. Нотация UML 2

Практическая работа	Номера задач
12.1. Нотация UML. Классы и объекты	1.1, 1.2, 1.3, 1.4, 1.6
12.2. Нотация UML. Варианты использования	2.1, 2.3, 2.4, 2.6
12.3. Нотация UML. Моделирование поведения	3.2, 7.1, 7.3, 8.1

Таблица 2. Состав практических работ. ООП с UML 2

Практическая работа	Номера задач
12.4. ООП. Анализ предметной области, идентификация классов	5.1, 5.2, 5.3, 5.5
12.5. ООП. Базовый уровень	2.9, 3.4, 3.6, 3.11, 5.9, 5.13
12.6. ООП. Принципы проектирования SOLID	4.5, 4.15, 6.8, 10.2, 10.4
12.7. ООП. Паттерны проектирования	4.3, 4.6, 4.14, 4.16, 10.1, 10.3, 10.6
12.8. ООП. Предметно-ориентированное проектирование. Анализ предметной области	9.1, 9.2, 9.3, 9.4
12.9. ООП. Предметно-ориентированное проектирование. Применение паттернов	9.5, 9.6, 9.7, 9.8, 9.9

Таблица 3. Состав практических работ. Моделирование на UML 2

Практическая работа	**Номера задач**
12.10. Моделирование с помощью UML. Создание сложных моделей из нескольких диаграмм	10.7, 10.9, 10.10, 10.11
12.11. Моделирование с помощью UML. Расширенные возможности диаграмм классов	4.1, 4.2, 4.4, 4.7, 4.8, 4.10, 4.11, 4.12, 4.17
12.12. Моделирование с помощью UML. Взаимодействия в кооперациях	3.7, 3.8, 10.4, 10.5, 10.8
12.13. Моделирование с помощью UML. Применение конечных автоматов для описания поведения	7.6, 7.7, 7.8, 7.9, 7.13
12.14. Моделирование с помощью UML. Моделирование бизнес-процессов диаграммами деятельности	8.3, 8.4, 8.5, 8.9
12.15. Моделирование с помощью UML. Реализация алгоритмов диаграммами деятельности	8.6, 8.7, 8.8(*), 8.10
12.16. Моделирование с помощью UML. Архитектурное проектирование	6.1, 6.3, 6.7, 6.8, 6.9

§13. ОТВЕТЫ И РЕШЕНИЯ

В данном разделе приводятся ответы к задачам и указания по решению. В начале работы со сборником рекомендуется ознакомиться с представленными решениями в §11.

Нумерация задач и вопросов к ним сохранена.

13.1. КЛАССЫ И ОБЪЕКТЫ

1.1.

Указание. Не следует дублировать ассоциации атрибутами.

1.2.

Указание. Структурные свойства, имеющие тип класса или типа данных рекомендуется показывать ассоциацией, при этом для типа данных или простого типа может быть использована нотация атрибута. При реализации интерфейса классом класс определяет все операции, определенные в интерфейсе, и участвует во всех ассоциациях, в которых участвует интерфейс.

1.3.

Указание. Если в задаче не указана область видимости, она задается по умолчанию и обычно считается общедоступной. Видимость по-умолчанию определяется при построении модели в виде договоренности. Обратите внимание, что класс *Date* остается неизменяемым. В таких случаях атрибуты класса не могут быть изменены после инициализации экземпляра и могут быть отмечены *{readOnly}*. В этом случае обычно нет необходимости вводить операции получения значений этих атрибутов.

1.4.

Указание. В UML принято соглашение по именованию элементов. Названия классов и классификаторов в целом начинаются с заглавной буквы, и каждое слово в составном названии начинается с заглавной, например *MyWindow*. Имена операций начинаются с прописной буквы, и далее каждое слово в составном имени начинается с заглавной, например, *setClickListener*.

Для того чтобы показать на диаграмме, что класс является абстрактным, лучше использовать украшение *{abstract}* по сравнению с использованием курсива в названии класса *Window*. При составлении диаграмм от руки на бумаге отличить курсив от прямого текста не всегда возможно.

1.5. См. §11.

1.6.

а. В класс *Collections* добавить атрибут *empty: Collection {readOnly}*.

б. См. рис. 53.

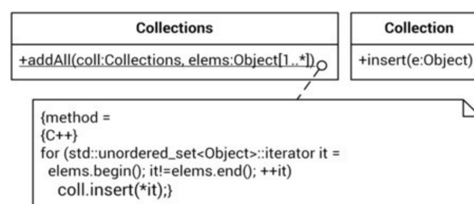


Рис. 53

1.7

а. Модель содержит семь экземпляров класса *Node*, соединенных в виде бинарного дерева: экземпляр в вершине дерева связан с двумя экземплярами второго уровня дерева, каждый из которых в свою очередь связан с двумя экземплярами третьего уровня дерева.

б. Указать кратность полюса ассоциации класса *Node* самим с собой *2..4*.

в. Добавить перечисление *NodeType* со значениями *Red* и *Black*. В класс *Node* добавить направленную к *NodeType* ассоциацию.

1.8.

а. Между *Subscription* и *Ticket* нужно добавить ассоциацию. В этой ассоциации кратность у полюса *Ticket 3..6*, у полюса *Subscription* кратность *0..1*.

б. Ограничение на естественном языке присоединяется к обеим ассоциациям: *{Ticket is linked with either Single either Subscription}*. Можно короче: *{xor}*.

в. 6.

1.9.

а. Добавить в модель классы *Brick* и *Plank*, которые уточняют класс *Material*. Добавить в модель перечисления *BrickKind* и *PlankKind* со значениями *red*, *white* и *pine*, *oak* соответственно. Определить ассоциации кратности 1 от классов *Brick* и *Plank* к перечислениям *BrickKind* и *PlankKind* соответственно. *Указание*. В более полной модели, в которой поведение разных материалов различается, предпочтительным будет решение с отдельными дочерними классами для каждого вида материала.

б. Добавить в модель агрегацию классом *Wall* класса *Brick* с ролью полюса *material* со стороны *Brick*. Добавить в модель класс *Frame*, который класс *Roof* агрегирует с кратностью 1. Добавить перечисление *FrameKind* со значениями *attic*, *flat*, *triangle*, с ассоциацией кратности 1 к нему со стороны класса *Frame*. Добавить в модель агрегацию классом *Frame* класса *Plank* с кратностью 0..40.

в. Добавить в модель класс *Tiling*, уточняющий класс *Material*, и агрегируемый классом *Frame* с кратностью *.

г. Потребуется один экземпляр универсального материала, так как агрегация позволяет вхождение элемента в несколько составных элементов (композитов). Если вместо агрегации использовать композицию, то потребуется не менее четырех экземпляров материала для четырех стен, а так как каркас и крыша не обязательны, всего потребуется не менее четырех экземпляров универсального материала.

13.2. СЦЕНАРИИ И ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ

2.1.

Указание. Различие между включением одним вариантом использования другого варианта использования и расширением вариантом использования другого варианта использования состоит в направлении зависимости между вариантами использования и в том, какой из них определяет условия расширения или включения. В случае включения эти условия определяет включающий вариант использования. В случае расширения – расширяющий.

2.2. См. §11.

2.3.

а. Актор *OCR* ассоциирован с вариантами использования: проверка слова *CheckWord*, приведения к определенной форме *GetForm*, восстановление нормальной формы *GetNormalForm*, получение грамматического значения *GetGrammarValue*. *GetForm* уточняет *GetNormalForm*.

б. Добавить вариант использования *SuggestWord*, расширяющий *CheckWord* в точке *wordNotFound*. Связать его с *OCR*.

в. Добавить абстрактный базовый для *CheckWord*, *SuggestWord*, *GetForm* и *GetGrammarValue* вариант использования *DictionaryLookupBase* и вариант использования *SearchWord*, включаемый в *DictionaryLookupBase*.

г. Добавить вариант использования проверки поддержи языка, и включить его в *DictionaryLookupBase*.

2.4.

а. Добавить абстрактный вариант использования *AttachDocument*, обобщающий варианты использования *AttachToIssue* и *AttachToResolution*. Добавить ассоциацию между ответственным лицом и *AttachDocument*.

б. Добавить в модель актора *Operator*, уточняющего *ResponsiblePerson*, и вариант использования *Delete-Document*. Связать их между собой ассоциацией.

в. Добавить в модель вариант использования *SendAnnouncement*, включаемый в *AttachDocument*. Добавить ассоциацию, соединяющую *SendAnnouncement* и *Operator*, полюс ассоциации у актора имеет множественную кратность и роль *controller*.

г. Указать ограничение *{xor}* между ассоциациями актора с *AttachToIssue* и *AttachToResolution*.

д. Ответственное лицо играет в ассоциации роль *user*. Указать для полюса ассоциации *user*, что он является производным множеством, объединяющем *author* и *chairman*.

2.5.

а. Добавить вариант использования *SelectPlugin*, расширяющий *ConfigurePluigns* в точке расширения *select*, и вариант использования *ChangeSettings*, расширяющий в точке расширения *config*. Связать оба варианта использования с *User*.

б. Добавить актора *PluginsServer*, вариант использования *UpdatePlugins* и связать их ассоциацией.

в. Добавить актора *SuperUser*, уточняющего *User*. Добавить вариант использования *UpdatePluginsList*, который включает варианты использования *DeletePlugin*, *InstallPlugins*, *CheckPluginList*. Последний вариант использования связать с *PluginsServer*.

2.6.

а. Акторы: *Researcher*, *ContentPartner*. Варианты использования *FindPapers*, с которым работает *Researcher*, и *LoadPapers*, с которым работает *ContentPartner*. Библиотека используется следующим образом: партнеры загружают статьи в библиотеку, исследователи могут осуществлять по ним поиск.

б. Добавить актора *Analyst*, действующего в рамках варианта использования *IndexPaper*, включенного в *LoadPapers*.

в. Добавить вариант использования *AdvancedSearch*, уточняющий *FindPapers*.

г. Добавить ко всем вариантам использования иконку в виде морских волн.

2.7.

а. Основной актор *Client*. Вспомогательный актор *Payment-System*. Оба актора взаимодействуют с системой в рамках варианта использования *PerformOperation*.

б. Добавить актора *BankCustomer*, который уточняет *Client*. Создать два конкретных варианта использования: получить наличные *GetCash* и оплатить услуги *PerformPayment*. Оба варианта использования сделать потомками *PerformOperation*. Наконец, нужно добавить ассоциацию между *Client* и *GetCash* и ассоциацию между *BankCustomer* и *PerformPayment*.

в. Нужно добавить в *GetCash* точку расширения *Another-Currency* варианта использования *ChooseCurrency* расширяет *GetCash* в точке расширения *AnotherCurrency*. *GetCash* включает в себя вариант использования *ConfirmOperation*, в котором реализовано подтверждение списания средств.

2.8.

а. Добавить актора *Programmer*, уточняющего *Linguist*. Добавить и связать ассоциацией с *Programmer* вариант использования *ExportData* для *MorphoDPS*.

б. Добавить вариант использования *ExportWordList*, уточняющий *ExportData*. Добавить актора *Semantics,* и связать его с *ExportWordList*.

в. Добавить три конкретных варианта использования, уточняющие *ModifyData*: *AddWord*, *ModifyWord*, *RemoveWord*.

г. Добавить вариант использования *CheckIntegrity*, расширяющий *ModifyData* в точке расширения *verify*.

д. Будет, если выполняется условие расширения.

2.9.

а. Добавить варианты использования авторизация *Authorization*, завершения сессии *EndSession*, абстрактный вариант использования обслуживания *Maintenance*. *Maintenance* включает *Authorization*, *EndSession*. Варианты использования *CollectCash*, *ChangeWater*, *ChangeGas* уточняют *Maintenance*.

б. Добавить вариант использования заправить сиропом *ChangeSyrop*, уточняющий *Maintenance*. *Loader* связан ассоциацией с *ChangeSyrop*.

в. Может, если выполняет роли обоих акторов *Cashier* и *Loader*.

г. Добавить вариант использования видеонаблюдение *Monitoring*. Между *Cashier* и *Monitoring* – ассоциация. Добавить вспомогательный актор датчик *Sensor*. Соединить его ассоциацией с *Monitoring*. *Cashier* использует систему в рамках этого варианта использования. Камера является частью системы.

2.10.

а. Добавить варианты использования *CreateCourseFrom-Template*, *CopyCourse*. Добавить ассоциацию между ними и актором *Professor*.

б. Пометить *ReviewDocument* как абстрактный. Добавить варианты использования, уточняющие *ReviewDocument*: *GuidedReview*, *JointReview* и *BasicReview*. Добавить актора *Student*. Добавить варианты использования, доступные *Student*: *Upload*, *Enroll*. Добавить ассоциацию между *Student* и *JointReview*.

в. Добавить абстрактного актора *Teacher*, конкретного актора *TA*. *TA* и *Professor* уточняют *Teacher*. Ассоциации от *CreateDocument*, *ReviewDocument* перенаправить вместо *Professor* к *Teacher*.

13.3. КООПЕРАЦИИ И ВЗАИМОДЕЙСТВИЯ КЛАССОВ

3.1. См. §11.

3.2.

Указание. На диаграмме последовательности роли (или линии жизни) представляют участников взаимодействия. Каждая линия жизни в каждый момент времени соответствует только одному экземпляру.

3.3. См. §11.

3.4.

а. *Person* связан ассоциацией *Attends* с *Therapist* и ассоциацией *Takes* с *Medicine*. Показать корректное направление прочтения имени ассоциации.

б. Создать кооперацию *Visit*. На диаграмме ее внутренней структуры разместить роли *Doctor* типа *Therapist*, *Patient* типа *Person*, *Subscription* типа *Medicine*. Соединители в кооперации есть между *Doctor* и *Patient*, между *Doctor* и *Subscription*.

3.5.

а. Добавить соединитель *drivetrain* между двигателем *Engine* и парой передних колес *front*.

б. Добавить в модель класс *FourWdCar*, уточняющий *Car*, у которого роль двигателя *engine* играет *DoubleEngine*, потомок *Engine*. Добавить два соединителя *drivetrain* между *engine* и *front* и *rear* парами колес.

3.6.

а. В цикле прохода по этажам указать оператор *loop (5)*. Указать, что параметр сообщения *pressButton* не *Integer*, а *«enumeration»* со значениями *1..5*.

б. Добавить в последовательность сообщение *pressDoors* от *Person* к *Lift*.

в. Сообщения начиная со *startMoving*, заканчивая *stop-Moving* обернуть во фрагмент *ignore {pressDoors}*.

г. Добавить временное ограничение на цикл прохода по этажам *{3 sec.}*.

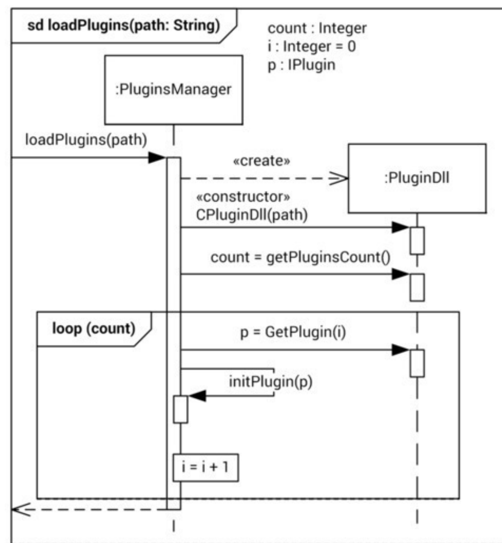


Рис. 54

3.7

а. См. рис. 54.

б. Создать новое взаимодействие *initPlugin* с параметром *pl* типа *IPlugin*, линиями жизни *pm* типа *PluginsManager*, *pl* и *ui* типа *PlayerUIPresenter*. Используя фрагмент *ref* включить новое взаимодействие в *loadPlugins*. Во взаимодействии *initPlugin* указать вызов операции *isUIHandled*, используя фрагмент *opt*, в случае, когда операция возвращает *true*, вызвать в *ui* операцию *addListener*, передав *pl* в параметрах.

3.8.

а. Добавить во взаимодействие атрибут *number: Integer*. У экземпляра класса кнопки этажа заменить селектор на *number*.

б. Добавить фрагмент *loop*. В этом фрагменте происходит посылка сообщения от линии жизни кабины к линии жизни кнопки этажа *isPressed ()*, посылка ответного сообщения, посылка сообщения *selectFloor ()* от кабины алгоритму, а также сообщения *number = number +1* от линии жизни кабины на ту же линию жизни. Добавить условие *[for all floors]* во фрагменте цикла на линии жизни кабины. См. рис. 55.

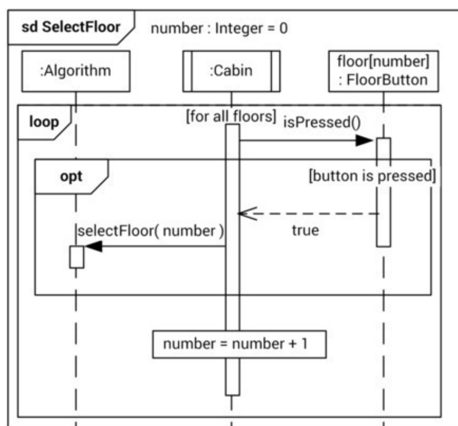


Рис. 55

в. Добавить создание и уничтожение объекта *Algorithm.*

г. Разрешенная траектория: отправка сообщения *isPressed ()* – прием сообщения *isPressed ()* – отправка ответного сообщения с параметром *true* – прием ответного сообщения – отправка сообщения *selectFloor ()* – прием сообщения *selectFloor ()* – отправка сообщения *number=number+1* – прием этого сообщения. Неразрешенная траектория: отправка сообщения *isPressed ()* – отправка сообщения *selectFloor ()*.

3.9.

а. Добавить две линии жизни типа *IUIListener* и комбинированный фрагмент *par* с двумя фрагментами. В первом и втором фрагментах указать вызов операции *handleEvent* на первой и второй линиях жизни соответственно.

б. См. рис. 56.

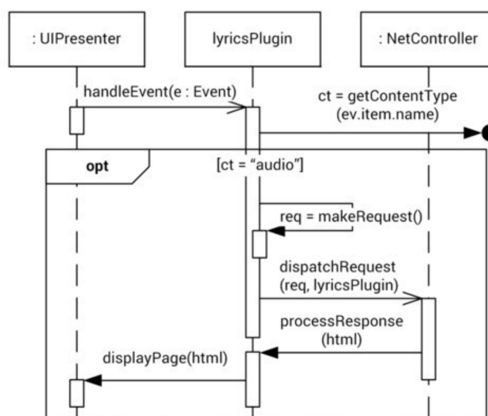


Рис. 56

3.10

а. Добавить роль *Node* кратности больше единицы, связанную с собой квалифицированным символом *ch* соединителем. Кратность *0..1*. Анонимная роль класса *Dictionary* соединена с Node, кратность у *Dictionary – 0..1*, у *Node – 1*.

б. *Указание.* Используйте комбинированный фрагмент *loop* по всем буквам слова, в котором от линии жизни класса *Dictionary* осуществляется вызов операции *getNextLetter* у текущего узла *node* типа *Node*. Начальное значение *node* получено в результате вызова операции *getRoot*.

в. Для описания исключительной ситуации используйте комбинированный фрагмент *alt*, один фрагмент которого соответствует обычной последовательности событий, а другой – последовательности с возникновением исключения.

3.11. Линии жизни: *AC*, *UI*, *DataSource*. Переменная взаимодействия *records: Record [*]*.

а. Добавить в контекст взаимодействия переменную *listSize*. Добавить синхронное сообщение *listSize=getListSize ()* до сообщения *readRecords ()*.

б. Добавить после сообщения *readRecords ()* фрагмент *loop*. Сторожевое условие фрагмента *[для всех i]*. Уточнить селектор в параметре действия *acceptRecord (records [i])*. После фрагмента *loop AC* посылает сообщение линии жизни UI *show (records)* без селектора.

в. См. рис. 57.

г. Соединители между: *AC* и *UI*, *AC* и *DataSource*. Передаются *show*, *getListSize*, *readRecords*, *acceptRecord*, *show*.

д. Все роли активных объектов могут быть реализованы объектом одного класса, неактивные объекты – объектом второго класса.

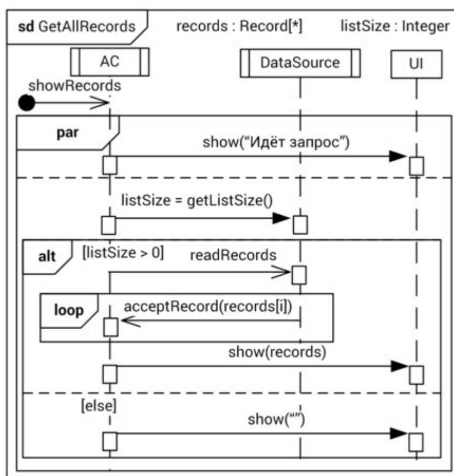


Рис. 57

13.4. РАСШИРЕННЫЕ КЛАССЫ И ОБЪЕКТЫ

4.1

а. Интерфейс *Entry* добавить в пространство имен интерфейса *Map*. На диаграмме отметить вложение с помощью отношения принадлежности пространству имен. У *Map* параметры шаблона оставить, у *Entry* параметры убрать.

б. Между *Map_StringInteger* и *Map* добавить отношение реализации со стереотипом *«bind»*. Дополнительно указать в угловых скобках <*K-> String, V-> Integer*>.

в. *Map_StringInteger* содержит ровно столько операций, сколько их содержит интерфейс *Map*, то есть не меньше трех.

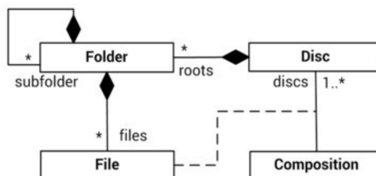


Рис. 58

4.2

а. См. рис. 58.

б. Сделать класс *Composition* абстрактным, добавить классы *Picture*, *Music*, *Movie*. Добавить отношение обобщения между каждым из этих классов и *Composition*

в. Не могут. Для того чтобы добиться такого поведения, нужно указать ограничение *{nonunique}* у полюса ассоциации *File* со стороны *Disk*.

г. *Указание.* Связи в UML идентифицированы значением своих полюсов, если для полюсов не указано *{nonunique}*, тогда связи снабжаются идентификатором.

4.3.

а. Вопрос по постановлению *ResolutionIssue*, отдельный вопрос *SingleIssue*, сложный вопрос *ComplexIssue* – подклассы *Issue*.

б. Ассоциация *Issues* между *Resolution* и *ResolutionIssue* с полюсом *topic* у *Resolution*, ассоциация *Attached* между *Resolution* и *Artifact* с полюсом *documents* множественной кратности у *Artifact*.

в. *ComplexIssue* агрегирует несколько *Issue* с полюсом *issues*, отношение обобщения добавлено в пункте а.

г. См. рис. 59.

4.4.

а. Между *HashMap* и *Map* добавить связь реализации. Между *HashEntry* и *Entry* добавить связь реализации.

б. *HashEntry* должен быть расположен в пространстве имен *HashMap*.

в. Так как *Entry* располагается в пространстве имен *Map*, *Entry* использует параметры *Map*.

4.5.

а. Получение указателя на функцию предполагает получение интерфейса, состоящего из одной операции вызов этой функции. Добавить интерфейс *IFunction* с операцией вызова *call* с параметром *params* кратности не меньше нуля без указания типа. Добавить класс *CDynamicLibrary*, в него добавить операцию *GetFunction* со строковым параметром и возвращаемым значением типа *IFunction*. Кроме этого, в *CDynamicLibrary* определены операции, указанные в условии задачи: *GetPluginsCount* типа *Integer*, *GetPlugin* типа *IPlugin* с параметром *index* типа *Integer*.

б. Добавить классы *CVisualizationPlugin* и *CSongLyrics-Plugin*, реализующие интерфейс *IPlugin*.

в. В первом случае класс *CPluginDll* уточняет *CDynamicLibrary* и перекрывает операции API. Во втором – включает его с помощью агрегации, вызов операций *CPluginDll* приводит к вызову операций *CDynamicLibrary*. В первом случае нарушается принцип подстановки (LSP).

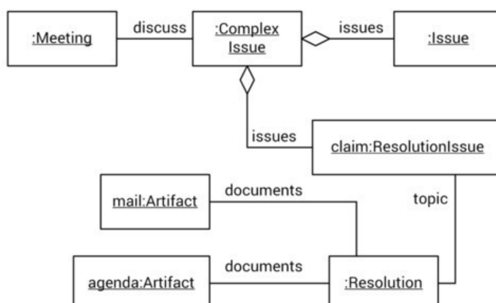


Рис. 59

4.6

а. Добавить классы *EngineSource* и *UISource*, уточняющие класс *EventSource*

б. Добавить интерфейсы *IEngineListener* и *IUIListener*, уточняющие *IEventListener*. Добавить классы *EngineEvent* и *UIEvent,* уточняющие *Event*, добавить отношения зависимости «create», между *EngineSource* и *EngineEvent*, и *UISource* и *UIEvent*. Указать, что *EngineSource* и *UISource* переопределяют операции *EventSource* и уточняют их параметры.

в. Добавить класс *PluginEventHandler*, включить в класс агрегацией не более одного *IEngineListener* и *IUIListener*. Добавить в *IPlugin* общедоступное свойство *handlers* типа *PluginEventHandler* кратности не меньше нуля.

4.7.

а. По свойству полиморфизма будет выполнен метод класса *Rectangle*.

б. Добавить в класс *Button* в операцию *equals* с параметром *Button,* перекрывающую одноименную операцию базовых классов. Добавить нечеткое поведение *«opaqueBehavior» equals*, реализующее операцию *equals*. Код метода приведен далее (см. ниже):

```
if (dynamic_cast<Rectangle*>(obj) != 0) {
   return Rectangle::equals(obj);
} else if (dynamic_cast<Clickable*>(obj) != 0) {
   return Clickable::equals(obj);
} else {
   return Object::equals(obj);
}
```

4.8

а. Указать у полюса класса *Element* ограничение *{ordered, nonunique}*.

б. Указать направление ассоциации от *Table* к *Element*.

в. Добавить квалификатор у полюса у класса *Table* с типом *Element*.

4.9.

а. Добавить в модель классы *Lecture* и *Practice*, которые класс *CourseOffering* агрегирует с кратностями *1* и *1..** соответственно.

б. Добавить ассоциации между классом *Teacher* и классами *Lecture* и *Practice*, такие, что у полюсов соответствующих ассоциаций со стороны *Teacher* указаны роли *lecturer* и *assistant* соответственно.

в. В ассоциациях между *CourseOffering* и *Lecture*, *Practice* указать роль *course* у соответствующего полюса ассоциаций. Полюс *teaches* ассоциации между *Teacher* и *CourseOffering* сделать производным и указать, что он является объединением множеств *assistant.course* и *lecturer.course*.

4.10.

а. См. рис. 60.

б. В класс *Controller* добавить операцию *+execute (cmd: Command)* типа *Result* и приемы сигналов: *TrainSpotted*, *TrainLeft*. Добавить типы данных *Command* и *Result*.

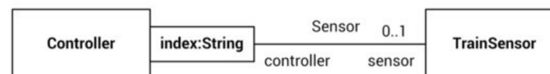


Рис. 60

в. Добавить класс *DigitalResult* и класс *DigitalController*, уточняющий класс *Controller*. В *DigitalController* определить операцию *+executeDigital ()* возвращающую *Digital-Result,* указать *{redefines execute}*.

г. См. рис. 61.

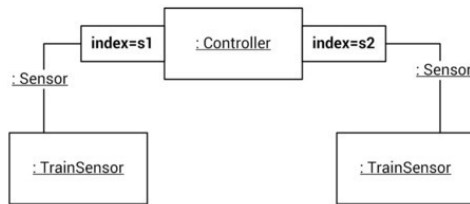


Рис. 61

4.11

а. Добавить в модель класс *Item*, содержащий структурное свойство *statement: String*, который является частью композиции класса-ассоциации *Contract*. Добавить класс *Manager* и ассоциацию *manages* от него к классу-ассоциации *Contract*.

б. Добавить в модель классы *Forward*, *Guard*, *Center*, уточняющие класс *Player* при помощи множества обобщений с ограничением *{disjoint}*.

в. Добавить в модель класс *Coach*, операции *attack*, *guard*, *join* классу *Player*, которые класс *Coach* вызывает асинхронно. Операция *join* имеет параметр *peer* типа *Player*. Указать зависимость класса *Coach* от класса *Player*. Добавить ассоциацию *Train* от класса *Coach* к классу *Player*.

г. Добавить класс *Role*, обобщающий классы *Forward*, *Guard*, *Center*. Вместо обобщения *Forward*, *Guard*, *Center* классом *Player* указать ассоциацию от *Player* к *Role* с кратностью 1 и именем *role* полюса со стороны *Role*. Использование ассоциации вместо обобщения позволяет изменять специализацию экземпляра класса *Player*.

4.12.

а. У квалифицированной ассоциации между *FAT* и *Cluster* нужно указать множественность *0..1* со стороны *Cluster*.

б. Добавить классификатор со стереотипом *«primitive»*, именем *Byte*, и ограничением *{value in 0..255}*. Добавить класс *File*. Связать его отношением обобщения с *Cluster*. Добавить этому классу операцию *getData ():Byte [*]*.

в. Добавить классы *Bad*, *Reserved*. Связать их отношением обобщения с *Cluster*.

г. См. рис. 62.

4.13.

а. Добавить к классу *Conversation* производное свойство *participants* типа *User*, которое является объединением *msgs.sender* и *msgs.recipients*.

б. Указать роль *msg* соответствующему полюсу ассоциации между *Document* и *Message*, и роль *conversation* полюсу ассоциации между *Message* и *Conversation*. Добавить ассоциацию между *Review* и *Document* и обозначить роль документа *manuscript*. На данную ассоциацию наложить ограничение *{manuscript.msg.conversation = author}*.

в. Для создания экземпляра класса *Review* необходимы экземпляр класса *Conversation* и экземпляр класса *User*.

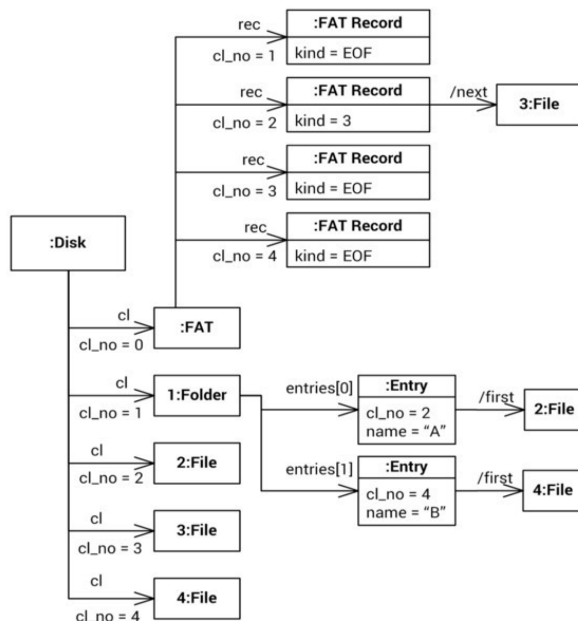


Рис. 62

4.14

а. Добавить перечисление *«enumeration» Floors* со значениями от одного до девяти. Добавить квалифицированную ассоциацию между *Cabin* и *Button*. Квалифицировать ассоциацию типом *Floors*. У полюса *Button* ограничение *{ordered}*, кратность *1*.

б. Добавить три ассоциации между *Cabin* и *UtilityButton*. У первой ассоциации имя полюса у *UtilityButton* – *call*, у второй – *stop*, у третьей – *openDoors*. Добавить ограничение *{xor}*, связанное со второй и третьей указанными ассоциациями.

в. Добавить классы *SmokeSensor*, *DoorsSensor*, *Overload-Sensor*. Все три связать с абстрактным классом *Sensor* множеством обобщений с параметрами *{disjoint, complete}*.

г. 10.

4.15.

а. См. рис. 63.

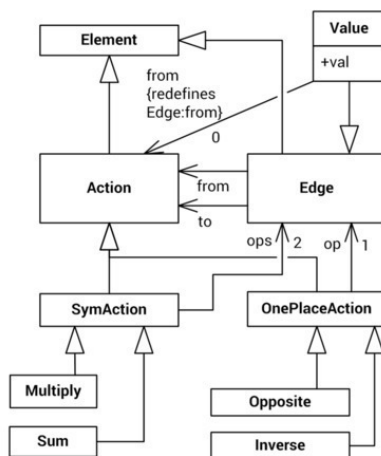


Рис. 63

б. См. рис. 64. В соответствии с SRP классы *Action* и *Edge* не могут реализовывать построение визуального представления графа. Поэтому создается отдельная модель визуального представления из прямоугольников *Rectangle* и стрелок *Arrow*, между которыми устанавливается соответствие через *Mapper*.

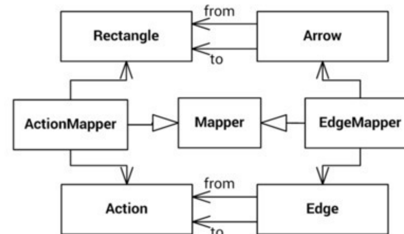


Рис. 64

в. См. рис. 65. Следует определить классы *IntValue* и *VarValue* уточняющие класс *Value*.

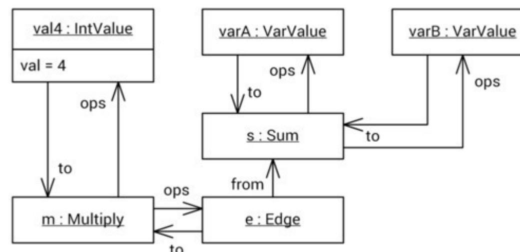


Рис. 65

4.16

а. Следует добавить в модель класс *Figure*, который обобщает классы *Arc*, *Line*. См. рис. 66.

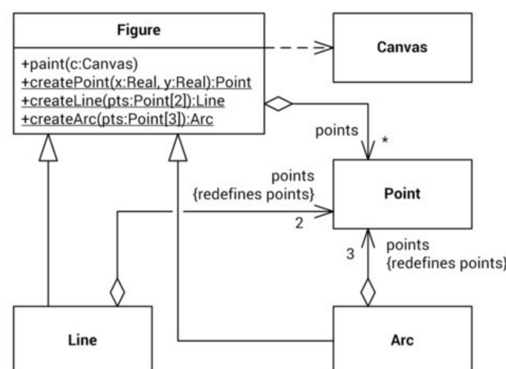


Рис. 66

б. См. рис. 67. *Figure::createPolyline* поддерживает выполнение ограничения.

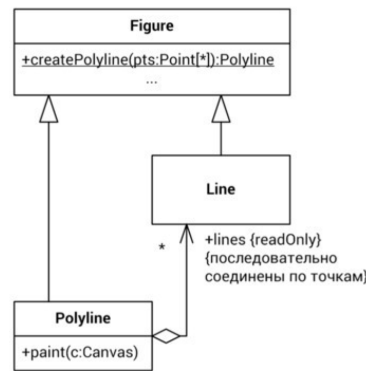


Рис. 67

в. Чтобы воспользоваться алгоритмом рисования отрезков следует определить и реализовать производное свойство *lines*. В сравнении с предыдущим решением такой подход позволяет изменять ломаную легче – через изменение множества точек – поддерживать ограничение последовательного соединения отрезков по точкам не требуется.

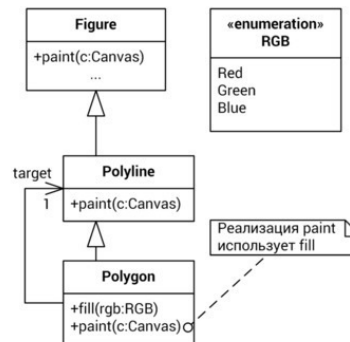


Рис. 68

г. См. рис. 68. Проверку свойств замкнутости и отсутствия самопересечения следует реализовать в классе *Polygon*.

4.17

а. См. рис. 69.

б. См. рис. 70.

в. См. рис. 71. Экземпляр *da* класса *DynAccount* ссылается на экземпляр *cpa* классов *Permanent* и *Corporate*.

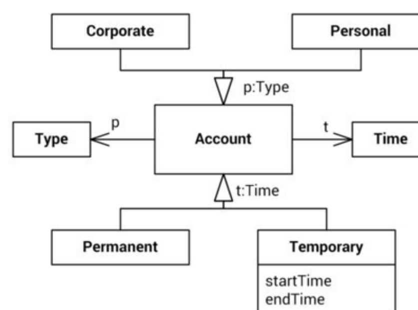


Рис. 69

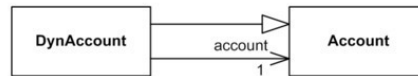


Рис. 70

4.18

а. Стереотип *Entity* уточняет *Domain Class*, стереотип *Root* уточняет *Entity*.

б. См. рис. 72.

в. Добавьте расширение метакласса *Package* стереотипом *Module*, в котором определена операция *encloses* с параметром типа *PackageableElement* и возвращаемым значением типа *Boolean*.

г. См. рис. 73.

д. Абстрактный стереотип *Service Interface* расширяет метакласс *Interface*. Стереотип *Repository* и *Domain Service*, уточняющие *Service Interface*, отношения обобщения образуют множество обобщений с параметрами расширения *{disjoint, complete}*.

е. *Указание*. Добавьте абстрактный базовый стереотип *Domain Stateless* и укажите ограничение на отсутствие свойств типа *Value Base* для него.

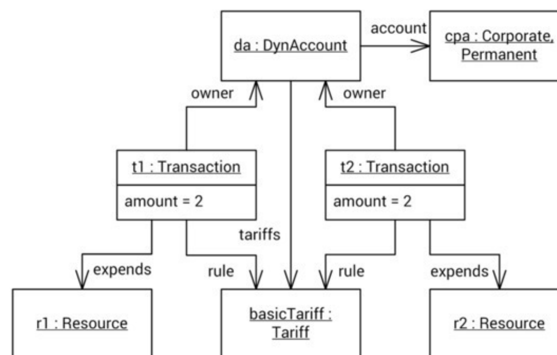


Рис. 71

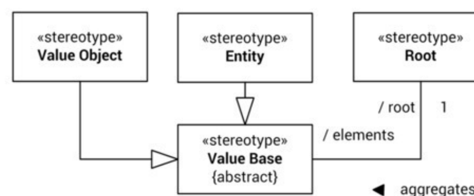


Рис. 72

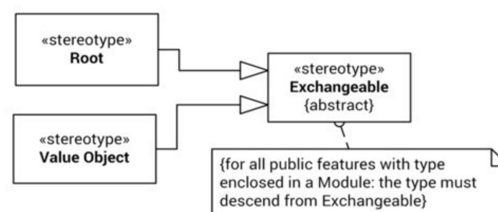


Рис. 73

13.5. АНАЛИЗ И ВЫДЕЛЕНИЕ КЛАССОВ

5.1.

а. См. рис. 74. В класс *Folder* добавить следующие операции:

+CreateSubFolder (path: String),

+CreateFile (name: String),

+DeleteSubFolder (path: String),

+DeleteFile (name: String).

б. Добавить в модель вспомогательный класс *Buffer*, в класс *File* добавить следующие операции:

+ReadToBuffer (buf: Buffer, pos: Integer),

+WriteToFile (buf: Buffer, pos: Integer).

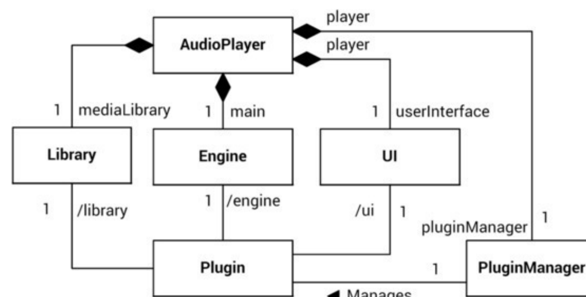


Рис. 74

5.2

а. См. рис. 76.

б. См. рис. 75.

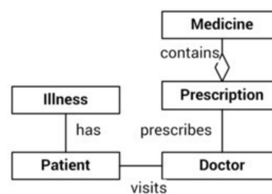


Рис. 76

Рис. 75

5.3

а. Характеристика качества, этап разработки, фактор, критерий, метрика.

б. См. рис. 77.
в. См. рис. 78.
г. См. рис. 79.

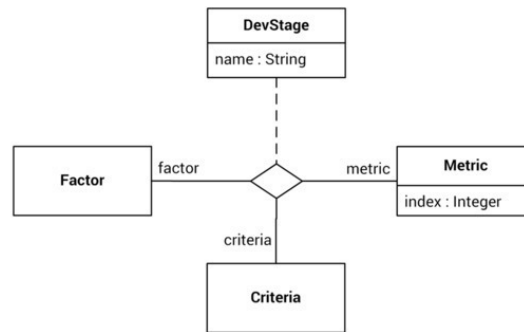


Рис. 77

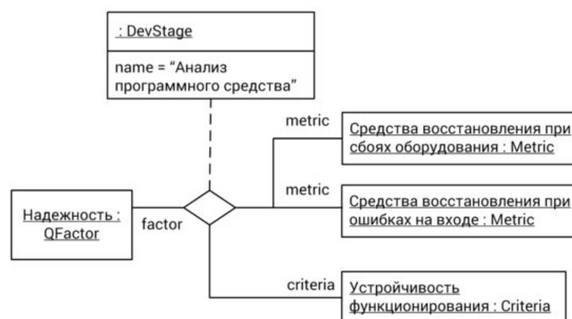


Рис. 78

5.4

а. См. рис. 80.

б. В класс *PluginManager* добавить операции *loadPlugin*, *enablePlugin* и *disablePlugin* с параметром *plugin*. Добавить в модель сигнал *UserPluginCommand*. с параметрами *plugin* и *cmd*. Класс *UI* сделать активным, добавить получение сигнала *UserPluginCommand*. В класс *Plugin* добавить частный атрибут *isEnabled*, и операции *enable* и *disable*.

в. Добавить классы *ConnectionManager* и *PluginLibrary*. Связать тройной ассоциацией загрузки подключаемого модуля *UpdatePlugin* добавленные классы с классом *PluginManager*.

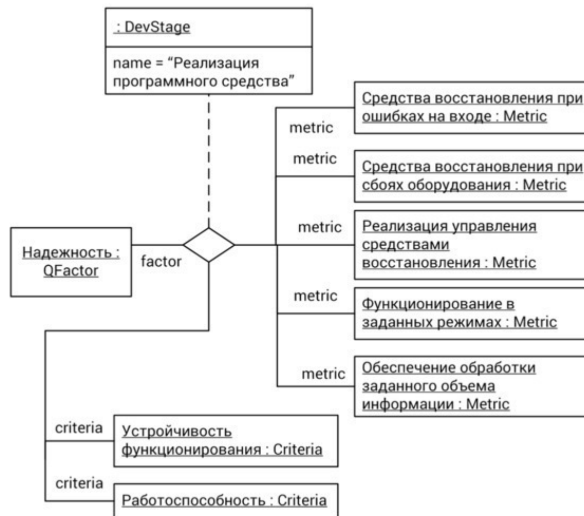


Рис. 79

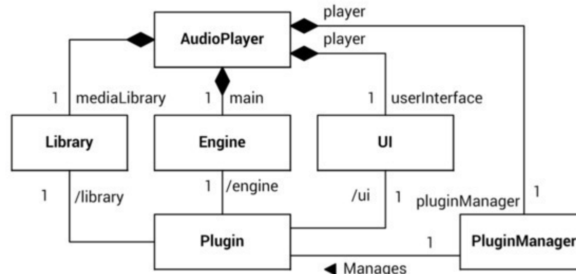


Рис. 80

5.5

а. См. рис. 81.

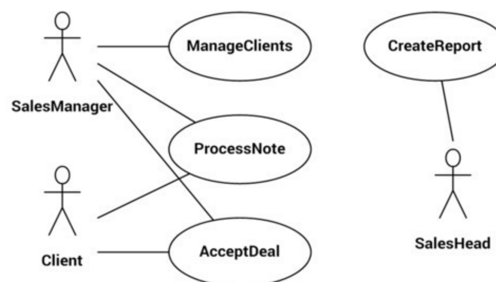


Рис. 81

б. Классы-кандидаты:

– Выделение существительных и выражений, составленных из существительных. Результаты без избыточных элементов и приведенные к единственному числу: управление клиентом, клиент, управление карточками клиента, карточка клиента, обработка заявки, заявка, проведение сделки, сделка, подготовка отчета, отчет, менеджер по продажам, руководитель отдела, реквизиты клиента, поданная заявка, дата заявки, время заявки, текст сообщения заявки, текст ответа по заявке. Не включены в результат элементы, не являющиеся существительными и следующие элементы, как не отражающие предметную область системы: функция системы, система, работа.

– Группировка элементов. Нерелевантные элементы (соответствуют функциям или процессам): управление клиентом, управление карточками клиента, обработка заявки, проведение сделки, подготовка отчета. Атрибуты классов: реквизиты клиента, дата заявки, время заявки, текст сообщения заявки, текст ответа по заявке. Классы с определениями: поданная заявка (синоним заявки). Классы: клиент, карточка клиента, заявка, сделка, отчет, менеджер по продажам, руководитель отдела.

в. См. рис. 82.

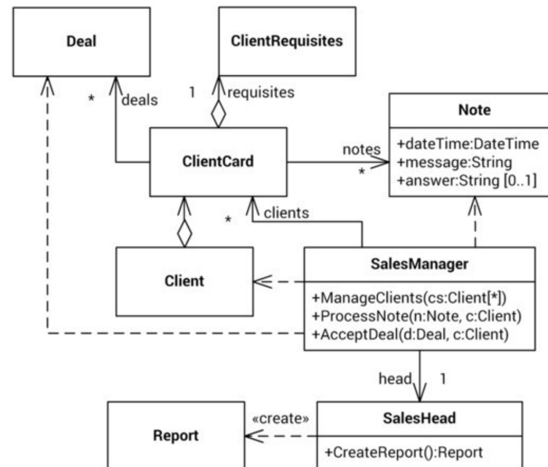


Рис. 82

13.6. АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ, КОМПОНЕНТЫ

6.1

а. См. рис. 83.

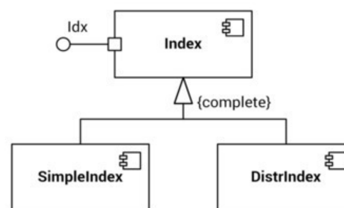


Рис. 83

б. См. рис. 84.

в. См. рис. 85.

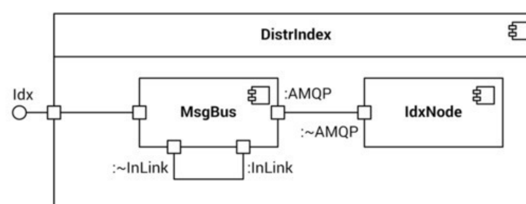


Рис. 84

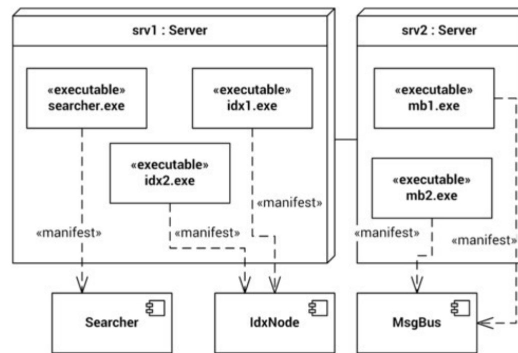


Рис. 85

6.2

а. Создать артефакты со стереотипом файл *«fr. lng»* и *«de. lng»*, добавить зависимость к ним от артефакта со стереотипом библиотека *Morphology. dll*. Последняя материализует *MorphoEngine* и зависит от *RootObjects. dll*.

б. Добавить артефакт со стереотипом файл *MorphoLocalize. dll*, который материализует компонент *MorphoLocalizeRu*, предоставляющий интерфейс *IMorphoLocalize*. Добавить зависимость *MorphoEngine* от интерфейса компонента *MorphoLocalize*.

в. *Указание*. Так как атрибут «версия» не определен, нужно создать профиль, в котором добавить обязательное расширение метакласса *Artifact* в виде стереотипа *VersionedArtifact*. В этом расширении создать атрибут *version* типа *String*. Затем применить профиль к модели, используя отношение *«apply»* между пакетом, в котором размещена модель и профилем. После этого указать версию библиотеки в теговом значении.

6.3.

а. См. рис. 86.

б. См. рис. 87.

в. См. рис. 88.

г. Элементы пространства имен *Elevator*: *engine: Engine*, *cage: Cage*, *controlUnit: ControlUnit*, набор *floorControl: FloorControl [1..*]*. Также в пространстве имен *Elevator* расположены порты с интерфейсами *Power*, *CageControls*, *Operations*, и набор портов типа *FloorButtons*, соединители *controls*, *cable*, *cageWire*, *floorWire* между частями, анонимные соединители частей с портами.

д. Порт *CageControls* соответствует методу

+getCageControls (): CageControls.

Порт *Operations* соответствует методу

+getOperations (): Operations.

Порты *FloorButtonsPort* соответствуют методу

+getFloorButton (floorNumber: int): FloorButtons.

Порт *Power* соответствует методу

+setPower (power: Power).

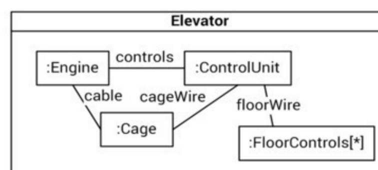


Рис. 86

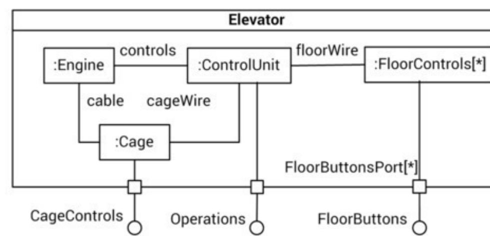


Рис. 87

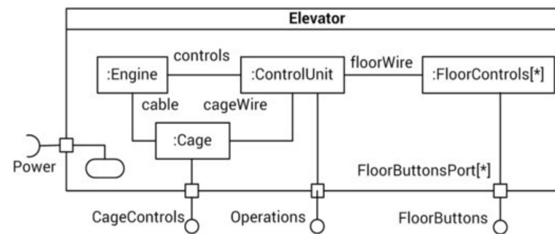


Рис. 88

6.4

а. См. рис. 89.

б. Создать по артефакту для каждого компонента. Артефакт базы данных разместить на узле *MorphoDB*, остальные – на узле *LinguistWorkPlace*. Узлы соединить ассоциацией.

в. См. рис. 90.

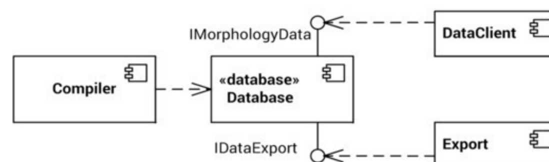


Рис. 89

6.5

а. См. рис. 91.

б. См. рис. 92.

в. Добавить в *MainApp* порт *ComputationEndpoint* с предоставляемым интерфейсом *Computation*. Соединить порт с *main*.

г. Добавить поведенческий порт *ConfigurationEnpoint*, предоставляющий интерфейс *Configuration*.

д. См. рис. 93.

е. Поведенческие черты в *MainApp* не упоминаются. Структурные черты: части: *ComputeBean*,:P*rocessorBean*, а также порты *ComputationEndpoint*, *ConfigurationEndpoint*. Соединительные черты: соединители: *ComputeLink* между процессными модулями, анонимный соединитель между модулем *main* и портом *ComputationEndpoint*.

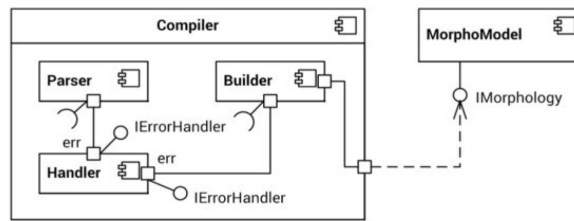


Рис. 90

6.6

а. Следует выделить отдельные интерфейсы в *MonolithService* для *WebSite* и API для поддержки соответствующих методов. См. рис. 94.

б. Модульная декомпозиция согласно принципу сокрытия информации (Information Hiding) позволяет абстрагировать соответствующую логику, упростить зависимости между элементами системы и облегчить внесение изменений в систему. На диаграмме ниже каждый из модулей уточняет базовый модуль *Module*, что позволяет выполнить маршрутизацию запроса от внешнего порта к подходящему для обработки запроса конкретному модулю. См. рис. 95.

в. См. рис. 96.

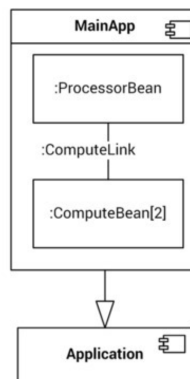


Рис. 91

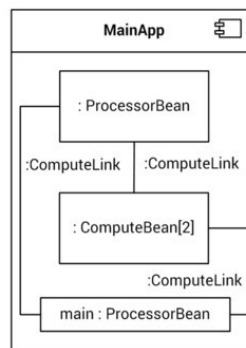


Рис. 92

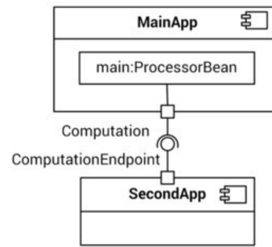


Рис. 93

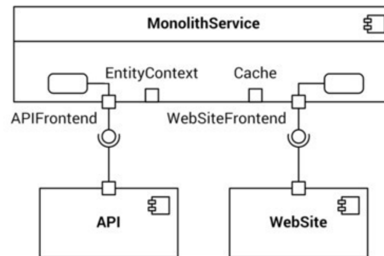


Рис. 94

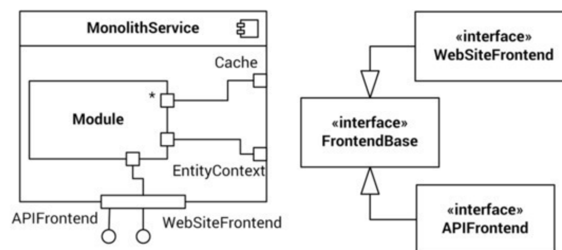


Рис. 95

6.7

а. См. рис. 97.

б. См. рис. 98.

в. См. рис. 99 и рис. 100.

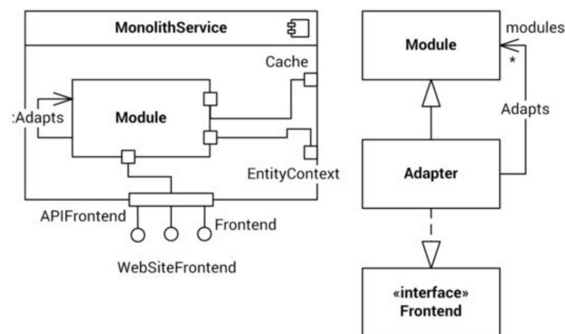


Рис. 96

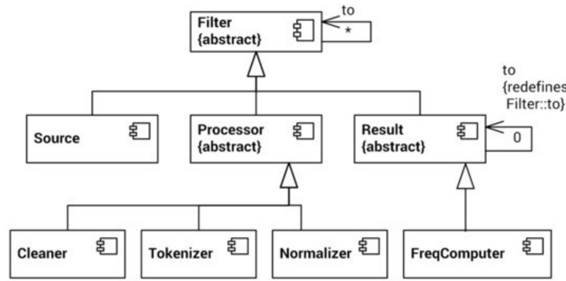


Рис. 97

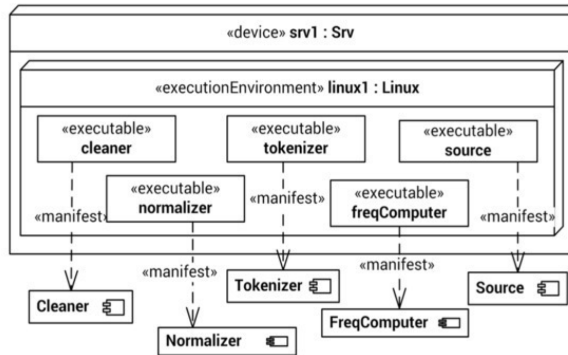


Рис. 98

6.8.

а. Модулями на диаграмме являются: ввод (*readf*), вывод (*writef*), преобразование одних структур данных в другие (*Main*). Для построения отчета следует добавить еще один модуль *createReport*, который в соответствии с принципом Information Hiding скрывает детали построения отчета. См. рис. 101.

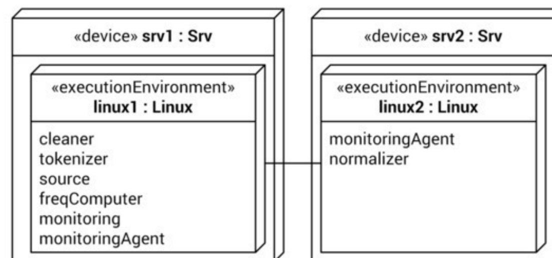


Рис. 99

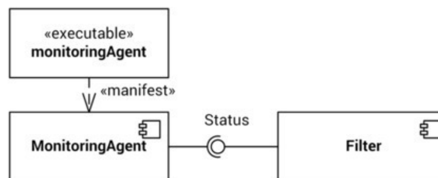


Рис. 100

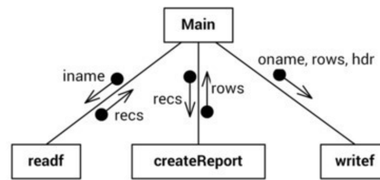


Рис. 101

б. См. рис. 102.

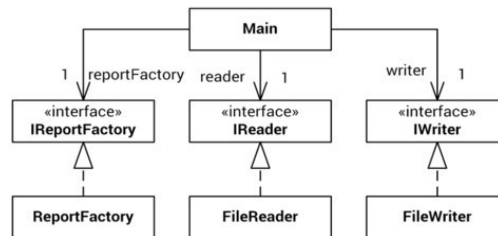


Рис. 102

в. Необходимо определить внешние порты *WebUI* и *RESTService* компонента *Main*. Чтобы реализовать чтение из базы данных вместо файла, необходимо реализовать класс *DbReader* с альтернативной реализацией интерфейса *IReader*. Чтобы реализовать построение другого отчета, необходимо реализовать класс с альтернативной реализацией интерфейса *IReportFactory*. См. рис. 104.

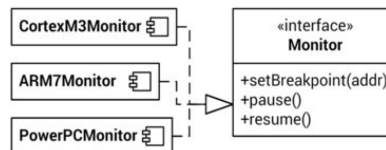


Рис. 103

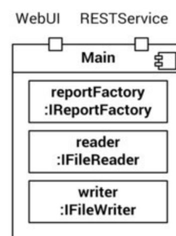


Рис. 104

6.9

а. См. рис. 103.

б. См. рис. 105.

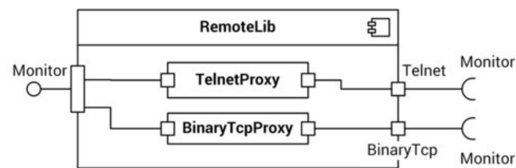


Рис. 105

в. См. рис. 106. В стиле клиент-сервер.

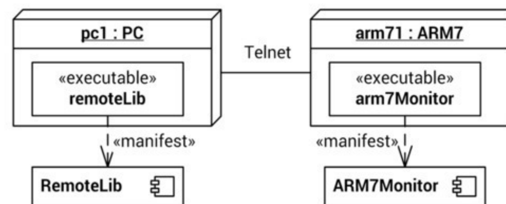


Рис. 106

13.7. СХЕМЫ СОСТОЯНИЙ И КОНЕЧНЫЕ АВТОМАТЫ

7.1

Указание. Обратите внимание на существенные различия между переходом по событию, когда для перехода определен триггер, и переходом по умолчанию с указанием сторожевого условия о наступлении события. В первом случае переход может быть выполнен в любой момент нахождения в данном состоянии, в то время как во втором случае условие наступления события будет проверено только один раз по завершении выполнения деятельности, связанной с состоянием или достижении конечного состояния во вложенных регионах. Если условие не выполнено, переход по умолчанию не будет произведен.

Кроме указания интервала времени, по истечении которого осуществляется переход, можно указывать любое однозначное описание моментов времени, когда должен срабатывать переход. В этом случае используется обозначение *at*, например, *at (за час до каждого нового года)*.

7.2. См. §11.

7.3.

а. Структура состояния *OnReview* приведена на рис. 107.

б. Добавить переход из *OnCorrection* в *Draft* по триггеру *after (10 days)*.

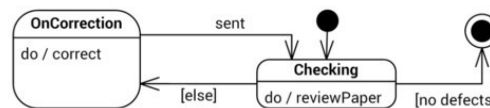


Рис. 107

7.4.

а. Переход из *Boarding* в *TakeOff* по *permit*, откуда по *airborne* в *InAir*. Из него по *landingPermit* в *Landing* и по *landed* в Boarding.

б. Вложить *TakeOff* и *Landing* в On*R*unway, в котором действие при выходе *exit/ freeRunway*, деятельность при нахождении *do/ radioComm*.

в. Вложенные в *InAir* из начального состояния в *Wait*, из него по событию времени *after (1h)* с условием *[longflight]* в *Dinner*, из него по событию *after (1h)* в финальное состояние в *InAir*.

г. Добавить ортогональный регион в *InAir*, в нем из начального в *Normal*, откуда по malfunction в *Damaged*. Переход по *landingPermit* теперь из *Normal*.

д. Обед прервется.

7.5.

а. В состояние *Ready* добавить вложенные *Stopped*, *Playing* и *Pause*. В *Stopped* переход по умолчанию. Из *Stopped* переход в *Playing* по *play*. Из *Pause* переход в *Stopped* по *stop*, в *Playing* по *play*. Из *Playing* переход в *Stopped* по *stop*, в *Paused* по *pause*.

б. В состояние *Ready* добавить новый ортогональный регион, в него добавить вложенные состояния *Searching* и *Downloading*. В *Searching* переход по умолчанию. Из *Searching* переход в *Downloading*, если найден новый контент *[contentFound]*, иначе в конечное состояние в регионе. Из *Downloading* переход по умолчанию в конечное состояние в регионе. В состоянии *Searching* выполняется деятельность *search*, в состоянии *Downloading – download.*

7.6.

а. Добавить состояние *Damaged*, в которое объект переходит из *Light* и *Off* при наступлении события *fell* и из *Light* при наступлении события *burnDown*.

б. Добавить терминальное состояние *Destroyed*, в которое лампа переходит из *Damaged* по событию *dispose*.

7.7.

а. Из вложенного в *Active* состояния *Tone* переход в новое, вложенное *Timeout* по событию *after (15s)*. Аналогично переход из *Ringing*.

б. Переход из *Active* в *Available* по *returnHandset*. Выполняется при нахождении в любом вложенном в *Active* состоянии.

в. Добавить вызов деятельности *playBusy* при нахождении *do/* в состоянии *Busy* и деятельности *playRing* при нахождении *do/* в *Ringing*.

г. Убедиться, что выделено состояние разговора *Talk*. Добавить переход по *hangUp* из *Talk* в *Busy*.

д. Отдельное состояние *Connect* с действием при входе *connect* и переходами по умолчанию с условиями *[busy]* в *Busy* и *[free]* в *Ringing*. Так как *connect* вызывает события в телефоне, они могут быть обработаны только после завершения перехода, в результате которого был вызван *connect*.

7.8.

а. См. рис. 108. Состояния: *Closed*, *Listen*, *SYN_Rcvd*, *Established*, *Active Close*, *Passive Close*. Переходы:

– *listen* по событию вызова операции;
– *SYN* из *Listen* с эффектом *SYN+ACK* по событию приема сигнала;
– *ACK* из *SYN_Rcvd* по событию приема сигнала;
– *close* из *Established* с эффектом *FIN* по событию вызова операции;
– *FIN* из *Established* по событию приема сигнала;
– *after (2s)* из *ActiveClose* по событию времени;
– *ACK* из *PassiveClose* по событию приема сигнала.

б. Переход по *connect* с эффектом *SYN* из *Closed* в новое состояние *ActiveOpen*, откуда по *SYN+ACK* в *Established*.

в. Из *PassiveClose* по *close* с эффектом *FIN* в конечное состояние. Из *SYN_Rcvd* по *close* с эффектом *FIN* в *ActiveClose*, из *ActiveOpen* по *close* в *Closed*. Вкладывать в одно состояния нельзя, так как целевые состояния различны.

г. Для каждого перехода с эффектом создать промежуточное состояние, в котором действие при входе – указанный эффект, и переход по умолчанию в целевое состояние исходного перехода.

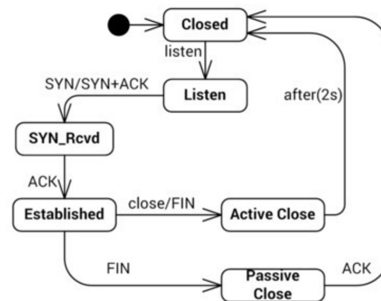


Рис. 108

7.9

а. Из начального состояния теперь переход в *Initiate*, в котором внутренние переходы по *estim* с эффектом *preparePlan* при условии *[budget> 100K]* и эффектом *sketch* иначе *[else]*. Из *Initiate* переход в *Run* по *approve*.

б. Добавить ортогональный регион в *Run*, в котором из начального состояния переход в *Communicate*, откуда по *meetingRequest* с эффектом *schedule* в *RunMeeting* в том же регионе. Из последнего по событию *done* обратно в *Communicate*.

в. Из *Run* по *suspend* переход в *Suspended*, откуда по событию времени *after (1y)* переход в *Shutdown*. Добавить состояние *ResumableRun* и вложить в него *Run*. По *resume* сложный переход в *ResumableRun* во вложенное глубокое историческое псевдосостояние, по умолчанию из которого в *Run*. *Указание*. Можно обойтись и без *ResumableRun*, если определить исторические псевдосостояния в каждом регионе.

г. Определить *SeniorPM* как подкласс *PM*, собственное поведение обоих классов задается схемой состояний. Для *PM* определена выше. Для *SeniorPM* в состояние *Run* добавляется еще один ортогональный регион. В нем из начального псевдосостояния переход в *AcceptCR*, по *accept* переход в *Estimate*, по *reject* и *approve* возврат в *AcceptCR*.

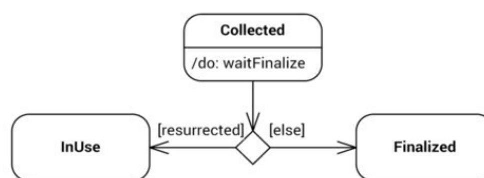


Рис. 109

7.10

а. Добавить состояние *Invisible*, в которое вместо состояния *Unreachable* объект переходит по *outOfScope*. В *Unreachable* из *Invisible* объект переходит по выходу из метода – *outOfMethod*.

б. См. рис. 109.

в. Добавить псевдосостояние выбора, в которое объект попадает из *Unreachable*. Далее при условии *[finalize () is not defined]* объект переходит в *Finalized*.

г. Добавить в модель композитное состояние *Resurrectable*, вложенными состояниями которого нужно сделать *Invisible*, *Unreachable*, *Collected*, *Finalized*.

д. Произвольное число раз.

7.11

а. В состояние *On* добавить действие при входе *indicatorOn* и действие при выходе *indicatorOff*.

б. Вложенные в *Ready* состояния поместить в один ортогональный регион, в другом создать начальное псевдосостояние с переходом в состояние *CheckWater* с внутренними переходами *heat* с триггером *when (cold)*, и *cool* с триггером *when (hot)*.

в. Добавить внешний переход в себя из *Ready* по *cancelButtonPressed*.

г. См. рис. 110.

д. См. рис. 111.

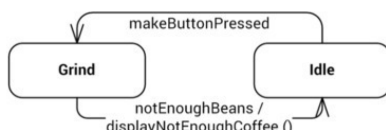


Рис. 110

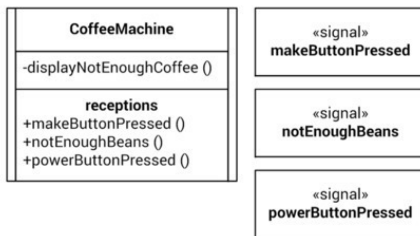


Рис. 111

7.12

а. Добавить состояние *WaterTankRemoved*, вложенное в *On*. В него переход из *Ready* по *waterTankRemoved* с эффектом *displayWaterTankRemoved ()*. Создать в *Ready* полное возвратное псевдосостояние, в которое сделать переход из *WaterTankRemoved* по *waterTankFixed*. Из псевдосостояния по умолчанию переход в *Idle*.

б. Из *SelfTest* переход по умолчанию в новое состояние *Error*, вложенное в *On*, со сторожевым условием *[not testSuccess]* и эффектом *displayErrorMessage ()*.

в. *Указание*. См. рис. 111., добавить частную операцию *displayErrorMessage ()*, убрать прием сигнала *notEnough-Beans* и добавить *waterTankRemoved*. Аналогично с сигналами.

7.13.

а. Добавить переход из *On* в *Off* по событию *turnOff*. если выключить, а затем включить печь, атрибут *cookingTime* не изменится. При включении ему будет установлено значение *10*, такое же, как и было до выключения.

б. Добавить внутренний переход из *On* в себя по событию *turnOn*. Эффектом такого перехода назначить *cookingTime = cookingTime +10*.

в. Добавить в *On* вложенные состояния *Working*, *NotWorking*. Из начального состояния композитного состояния *On* сразу осуществляется переход в *Working*. Добавить переход из *Working* в *NotWorking* по сигналу o*pen* и обратный переход по сигналу c*lose*. Убрать переход из *On* в себя по триггеру *after (1 sec)* и по сигналу t*urnOn*. Добавить эти переходы (с соответствующими эффектами) во вложенное состояние *Working*.

г. Добавить в *On* ортогональные регионы. В одном из них оставить *Working*, *NotWorking*. В другом разместить вложенные состояния *DoorOpen*, *DoorClosed*. Добавить между *DoorOpen*,

DoorClosed переходы по сигналам o*pen*, *close*. По сигналу t*urnOn* из *Off* происходит переход в псевдосостояние ветвления, далее если сторожевое условие *[isOpen]* выполнено, то происходит комплексный переход в *DoorOpen* и в *NotWorking*, если же *isOpen* ложно, то одновременный переход в *DoorClosed* и *Working*.

7.14.

а. См. рис. 112.

б. Добавить сигнал *Call* с параметром *pos: Integer*. В состояние *Online* добавить внутренние переходы в себя по получении сигнала *floor* с эффектом *fromFloor. push (floor. pos)* и сигнала *call* с эффектом *fromCabin. push (call. pos)*.

в. См. рис. 113.

г. Добавить переход по *off* из *NotClosed* в *Offline*. В состоянии *Online* указать, что обработка *off* откладывается.

д. Сигналы *on*, *off* без параметров. Сигнал *floor* с параметром *pos* типа *Integer*, сигнал *call* с параметром *pos* типа *Integer*, сигнал *button* с параметром типа *Integer*.

Класс очереди *Queue* содержит операцию *push* с параметром *w* типа *Integer*, операцию *pop* без параметров, возвращает *Integer*.

Класс *Elevator* содержит атрибут *dest* типа *Integer*, свойства с *fromCabin* и *fromFloor* типа *Queue* кратности единица. *Elevator* принимает сигналы *on*, *off*, *floor*, *call*, *button*. *Elevator* содержит операции и реализующие их методы *cleanup ()*, *openDoors ()*, *closeDoors ()*, *accelerate ()*, *slowdown ()*, *nextDestFloor ()*.

е. Отвергаемая последовательность: *off*. Принимаемая последовательность: *on*, *call*, *floor*, *off*.

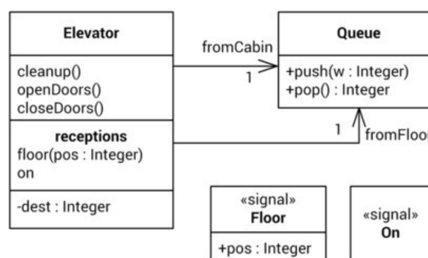


Рис. 112

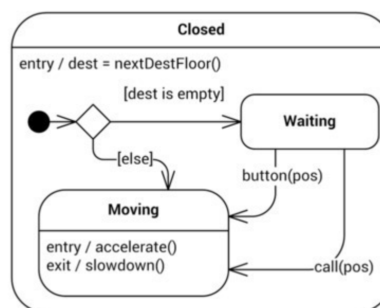


Рис. 113

13.8. ПРЕДСТАВЛЕНИЕ ДЕЯТЕЛЬНОСТИ И ПОТОКОВ РАБОТ

8.1

Указание. Обратите внимание на различия в использовании объектных потоков для передачи ссылок на экземпляры и потоков управления для передачи меток управления (вызова действия). Метки управления и ссылки на экземпляры не могут проходить по одним и тем же ребрам деятельности.

Контакты действия обозначают места подсоединения ребер деятельности, принимают и передают метки и ссылки. Контакты могут соответствовать параметрам вызываемого поведения. Имя контакта должно совпадать с именем параметра.

8.2. См. §11.

8.3.

а. Выделить на диаграмме деятельности разделы *Stakeholders*, *Review committee*, *SPL Team*. Прием сигнала *Wait change* в разделе *Stakeholders*, деятельность *Review approve* в разделе *Review committee*, *Plan* – в *SPL Team*. Добавить в структурную модель классы *Stakeholders*, *Review committee*, *SPL Team*.

б. Добавить на диаграмму деятельности объектный узел *Specifications* со стереотипом *«datastore»*. Перенаправить объектный поток *spec* в узел *Specifications*. Из *Specifications* направить метку *spec* в деятельность *Plan*.

в. См. рис. 114.

г. 5 объектных меток, 4 меток управления.

8.4.

а. После действия *Testing* поток управления переходит в узел ветвления. При условии *[обнаружены дефекты]* управление переходит в *Development*. Иначе *[else]* управление переходит в *Deployment*.

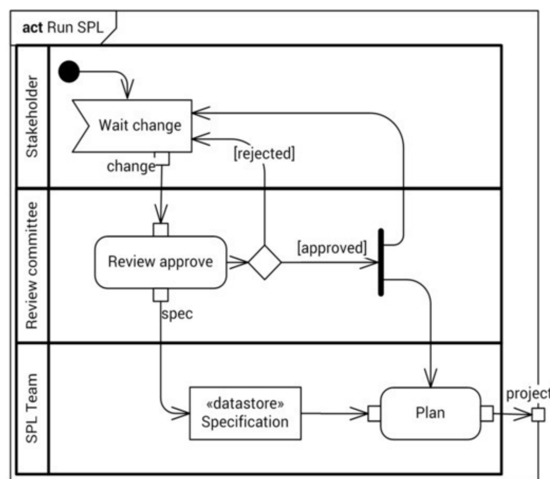


Рис. 114

б. Добавим действие *DeployPrepare*. Объектный поток описания проекта *SoW* из *Communication* попадает на узел разделения. Один поток приходит в *Modelling*, а второй в *DeployPrepare*. Поток управления из *DeployPrepare* направить в *Deployment*.

в. Порядок действий: *Communication*, *Modelling*, *Development*, *Testing*, *DeployPrepare*, *Deployment*. (Порядок между *DeployPrepare* и действиями *Modelling*, *Development*, *Testing* может быть произвольным). Если в продукте *Product* обнаружены ошибки, метка управления будет передана в *Development*, но чтобы запустить действие *Development* требуется также метка *Model*, которая по условию задачи приходит единожды из *Modelling* и поглощается в *Development*. Чтобы избавиться от тупика, нужно передавать метку *Model* в *Development* из репозитория *«datastore» Model-Repository*, упомянутого в условии. См. рис. 115.

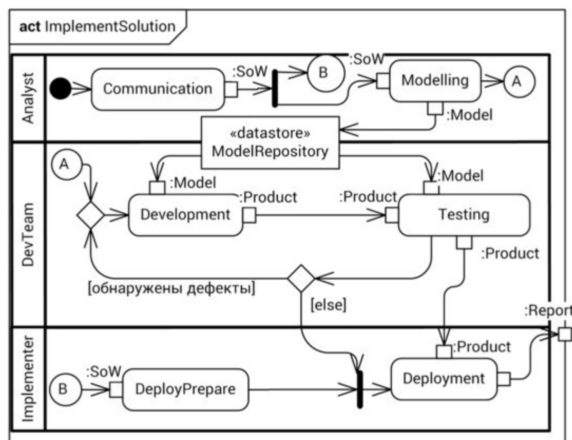


Рис. 115

г. Использовать узел разделения, пометить контакты, передающие метки: *SoW* между *Communication* и *Modelling* как *{stream}*. См. рис. 116.

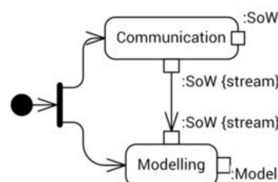


Рис. 116

д. *Analyst*, *DevTeam*, *Implementer*, *SoW*, *Report*, *Product*, *Model*, *ModelRepository*, *ImplementSolution*.

8.5

а. Поток управления из действия *GiveMoney* направить в действие *ThankCustomer*. Добавить управляющий узел объединения, направить в него потоки управления из *ThankCustomer* и *ReceiveMoney*. Поток управления из узла объединения направить в *Farewell*.

б. Добавить объектный поток типа *Identity* из *AskDocuments* в *CheckPassport*. Добавить узел разделения, в который направить объектный поток типа *Identity* из *CheckPassport*. Из узла разделения направить один поток в *GiveMoney* и другой в *ReceiveMoney*.

в. Указать свойство *stream* у контактов типа: *Note* у действий *GiveMoney* и *ReceiveMoney*. См. рис. 117.

г. 14.

д. Классы: *Customer*, *Clerk*, *Identity*, *Receipt*, *Note*.

Рис. 117

8.6

а. Вызов *one ()* вернет 2, *two ()* вернет 1.

б. См. рис. 118.

8.7.

а. См. рис. 119.

б. Добавить начальный узел деятельности и узел слияния. В узел слияния направить поток управления из начального узла и поток из *return from signal ()* вместо узла объединения. Направить поток управления из узла слияния в узел разделения.

в. *Указание.* Используйте уже имеющиеся операции класса.

8.8.

а. Сумму всех элементов массива.

б. Действие является нечетким *«OpaqueAction»*, после изменения станет действием вызова поведения *«CallBehaviorAction»*.

в. Определить поведение *mul*, реализующее умножение двух элементов, и заменить вызов поведения *add* на вызов *mul*.

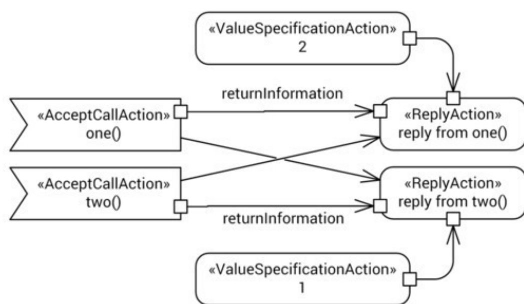


Рис. 118

8.9

а. Добавить раздел деятельности, соответствующий *Player-Controller*, в него поместить действия вызова операций *BuildPluginsList*, *ConfigureSelectedPlugin* и *InstallNewPlugins*. Добавить раздел, соответствующий *PlayerUI*, и поместить в него действия вызова операции *DisplayPluginsList* и приема сигнала *UserActivity-Received*. На диаграмме классов показать операции и прием сигнала в соответствующих классах.

б. После действия *BuildPluginsList* добавить узел разделения в объектный поток из контакта со списком подключаемых модулей *PluginsList*. Из разделения один поток направить в действие *DisplayPluginsList*, второй в новое действие *UpdatePlugins*. Поток управления после *UpdatePlugins* попадает в конечный узел деятельности.

в. См. рис. 120.

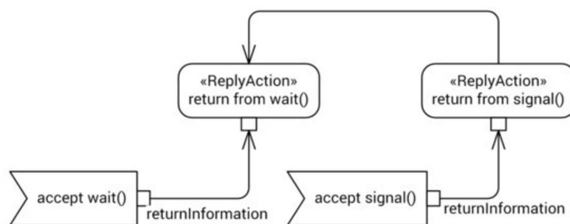


Рис. 119

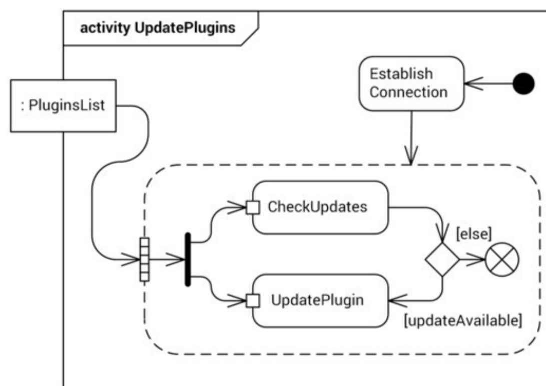


Рис. 120

8.10

а. *TrafficLightsBuilder*, *TrafficLights*, *Controller*.

б. См. рис. 121.

в. Поток данных из *CreateInstance* в *AddTrafficLights*, используя узел разделения, разбить на два: один по-прежнему направлен в *AddTrafficLights*, а второй направить на выходной параметр деятельности типа *Controller*.

г. Да.

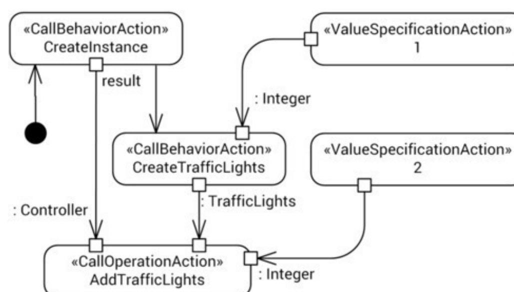


Рис. 121

8.11.

а. Операции класса *ArithmeticException*: *printStack-Trace ()*; переменные деятельности: *result: Integer*; обработчик исключения: *«e.printStackTrace»*.

б. В действие *«a / b»* приходит метка управления, метка данных *«a»* (*a = 1)*, метка данных *«b»* (*b = 0)*. Структурное действие *Divide* прерывается, передавая поток управления и метку *e* типа *ArithmeticException* в обработчик исключения. Выполняется действие *e.printStackTrace ()*. Затем происходит переход в структурное действие *PrintResult*. Выполняется действие *read result*. В *result* нет значения (*null)*. Выполняется действие *print (result)*. Деятельность завершается.

в. См. ниже.

```
void divide( int A, int B ) {
   int result;
   try {
      result = A / B;
   } catch( ArithmeticException e ) {
      e.printStackTrace();
   }
   print( result );
}
```

Первое структурное действие соответствует блоку кода
try {result = A / B;}
Второе структурное действие соответствует строке кода
print (result);
г. См. рис. 122.

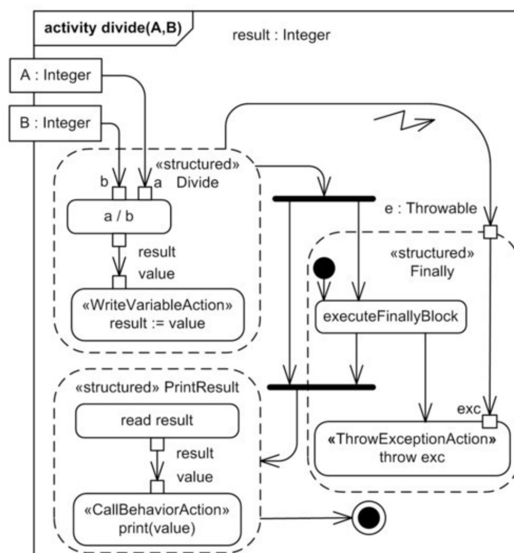


Рис. 122

13.9. ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

9.1

а. Способы прояснения понятий: изучение документации других систем, изучение аналитических шаблонов, общение с экспертом предметной области, изучение документов, связанных с предметной областью, разработка модели на основе текущего понимания и проверка на соответствие требованиям.

б. Предложение 5), так как исследуется предметная область системы управления виртуальными машинами, что не включает детали поддержки и обслуживания гипервизора. Сущности: 1) ВМ, 3) группа ВМ, 4) АР, гипервизор, 6) ресурс оперативная память, ресурс процессор, 7) пул АР.

в. Структурные свойства класса ВМ: объем оперативной памяти, число процессоров. Операции: создания, удаления, включения, выключения.

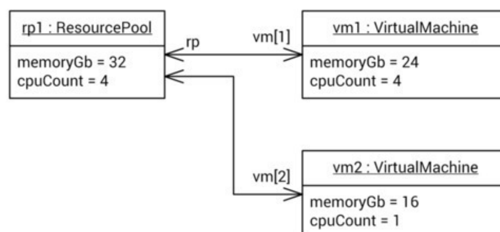


Рис. 123

г. Группа как пул АР будет содержать только виртуальные машины, общий объем памяти и процессоров которых не будет превышать объем ресурсов пула АР. Это пример реализации бизнес-правила в структуре модели. Пример нарушения приведен на рис. 123.

д. Пользователь должен использовать непосредственно пулы АР. То есть понимать природу реализации группировки ВМ. С позиций DDD это правильно, так как: 1) все элементы модели системы должны соответствовать понятиям предметной области непосредственно, а какого-либо иного элемента, используемого для группировки ВМ, в предметной области не существует, 2) всюду должен использоваться единый язык, тогда как понятие «группа ВМ» инженеру неизвестно. Сокрытие, переименование, искажение понятий на практике приводит только к проблемам: усложнение коммуникаций, возникновение риска недопонимания заинтересованными лицами друг друга, ошибки при работе пользователей из-за непонимания настоящей реализации функциональности.

9.2.

а. См. рис. 124. ЭКГ *Ecg*, протокол *Protocol* – сущности, так как идентифицируются глобально, характеризуются непрерывностью и индивидуальностью существования (а не значениями свойств), имеют изменяемое состояние. Фильтр *Filter*, анализатор *Analyzer* – службы, так как соответствуют процессам, алгоритмам, не имеют состояния. Сигнал ЭКГ *EcgSignal* – класс-значение, определяется значением свойств, а не непрерывностью и индивидуальностью существования, при необходимости изменения создается новый экземпляр.

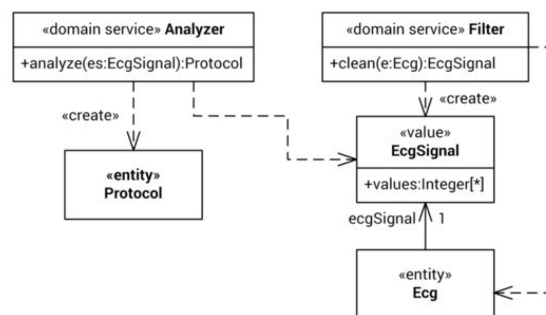


Рис. 124

б. См. рис. 125.

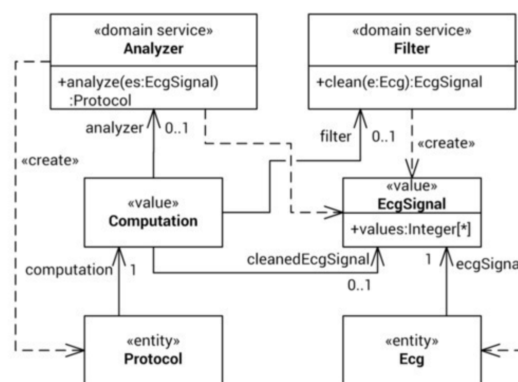


Рис. 125

в. Вычисление *Computation* – значение, определяется значениями свойств, а не непрерывностью и индивидуальностью существования. То есть по условиям задачи два вычисления

Computation можно считать одинаковыми, если они ссылаются на одни и те же экземпляры анализатора *Analyzer*, фильтра *Filter*, одинаковые сигналы ЭКГ *EcgSignal*.

9.3.

а. См. рис. 126.

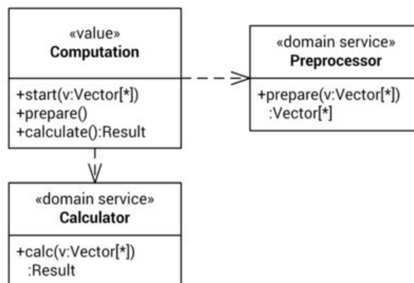


Рис. 126

б. См. рис. 127.

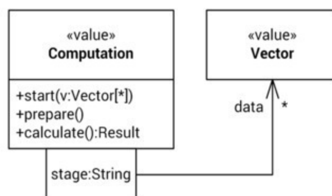


Рис. 127

в.

```
Computation.start(v:Vector[*]) {
   this.data[raw] = v
}
Computation.prepare() {
   Preprocessor preprocessor = new Preprocessor();
   Vector[*] prepared = preprocessor.prepare(
      this.data[raw]);
      this.data[prep] = prepared;
}
Computation.calculate():Result {
   Calculator calculator = new Calculator();
   return calculator.calc(this.data[prep]);
}
```

Всего создается 4 экземпляра (2 вычислением *Computation* и по одному службами предметной области *Preprocessor и Calculator)*.

г. Необходимо добавить ассоциацию от вычисления *Computation* к результату *Result* кратности *0..1*. В поведении *calculate* инициализировать ассоциацию созданным экземпляром класса *Result*. Тогда реализация поведения *isFinished* вычисления *Computation* будет состоять в проверке наличия значения у полюса ассоциации. См. рис. 128.

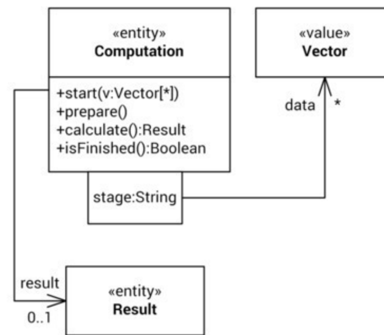


Рис. 128

д. Вычисление *Computation* после внесенных изменений стало сущностью, так как характеризуется непрерывностью и индивидуальностью существования. Оно имеет смысл процесса диагностики, в мониторинге состояния которого заинтересованы различные внешние клиенты. То есть два экземпляра вычисления *Computation*, даже если они ссылаются на одни и те же экземпляры служб предметной области *Preprocessor*, *Calculator*, одинаковые данные *Vector [*]*, не являются одинаковыми, так как согласно предметной области соответствуют разным процессам диагностики, которые должны быть различимы.

9.4.

а. Фильтр *Filter* использует частоту дискретизации *frequency*, единственный способ узнать которую – обратиться к прибору *Device* ЭКГ *Ecg*. Назначение фильтра *Filter* – обработка сигнала ЭКГ *EcgSignal*, осведомленность о приборе *Device* не нужна, а будучи избыточной, позволит фильтру *Filter* выполнять нежелательные операции, такие как включение *on* или выключение *off* прибора *Device*.

б. См. рис. 129.

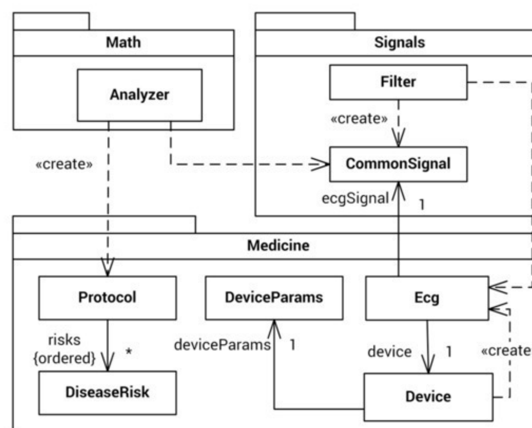


Рис. 129

в. См. рис. 130.

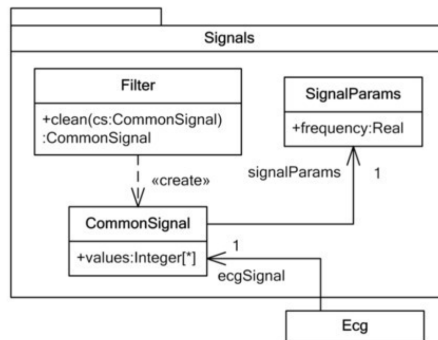


Рис. 130

г. См. рис. 131.

д. Необходимо добавить новые классы с семантикой массива чисел аналогичной *CommonSignal* в модули математика *Math* и медицина *Medicine*.

Дополнительная информация: Данный пример иллюстрирует подход к модульной декомпозиции системы на основе соответствия модулей подобластям предметной области. Такие модули также называются ограниченными контекстами (bounded contexts). Ограниченные контексты необходимы при разработке больших систем или систем со сложной предметной областью, унификация которой на уровне всей системы невозможна. Вместо этого разрабатывается несколько слабо связанных ограниченных контекстов, которые определяют область существования каждого понятия, и между которыми выполняется трансляция понятий.

е. Согласно принципам DDD, каждый элемент модели должен непосредственно соответствовать элементам предметной области, и модули в этом смысле не являются исключением. В рассматриваемой задаче модули соответствуют предметным областям.

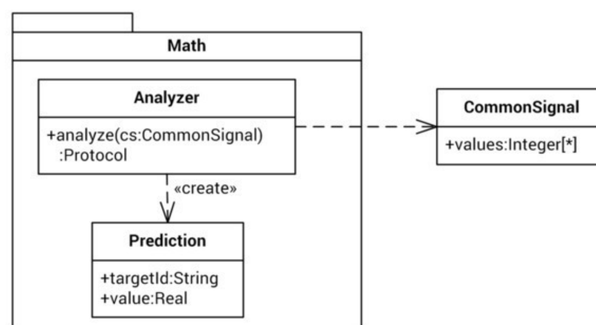


Рис. 131

9.5

а. Модель предметной области представлена на рис. 132. Карточка пациента *MedicalRecord* – сущность, так как соответствует пациенту, имеет уникальный идентификатор, характеризуется непрерывностью и индивидуальностью существования, заболевание *Disease* – сущность, так как по условию соответствует классификатору МКБ-10, протокол диагностики *Protocol* – сущность, так как по условию протоколы просматривается внешними клиентами, а значит, каждый протокол должен быть уникально идентифицируем. Таблица заболеваний *DiseaseTable* и риск заболевания *DiseaseRisk* – значения, так как характеризуются значениями своих свойств. Служб нет.

б. Всего три агрегата: заболевание *Disease*; карточка пациента *MedicalRecord*; протокол диагностики *Protocol* включающий также таблицу заболеваний *DiseaseTable* и риск заболевания *DiseaseRisk*.

в. Текущее проверенное заболевание *VerifiedDisease* – значение (так как описывает текущее состояние пациента, то есть характеризуется значениями свойств) внутри агрегата карточка пациента *MedicalRecord*.

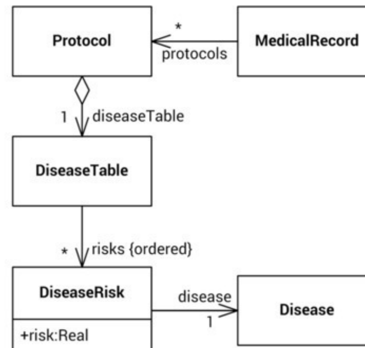


Рис. 132

г. Класс событие пациента *PatientEvent* – сущность (так как нельзя считать два события пациента с одинаковыми значениями свойств идентичными с точки зрения предметной области) внутри агрегата карточка пациента *MedicalRecord* (так как не требуется прямой доступ или изменение событий пациента, минуя карточку пациента). Врач *Doctor* – сущность, корень агрегата, и единственный класс агрегата. См. рис. 135.

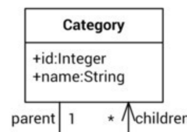


Рис. 133

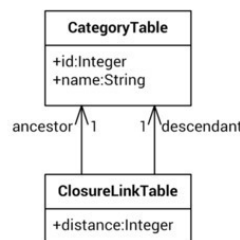


Рис. 134

9.6

а. См. рис. 133.

б. См. рис. 134.

в. Операции: *addChild (child: Category, parent: Category); subtree (category: Category, maxDepth: Integer): Category [*].*

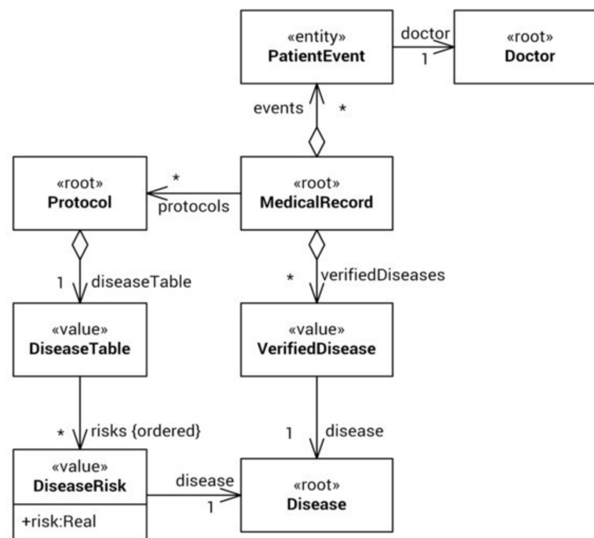


Рис. 135

В определении операций репозитория можно использовать только классы модели предметной области, поскольку одно из назначений репозитория – изоляция слоя предметной области от механизмов хранения, возможность подмены реализации хранилища, например, для нужд тестирования. Использование в определении операций репозитория элементов модели хранения данных приводит к смешиванию моделей и зависимости слоя бизнес-логики от технологий реализации хранения агрегатов.

г. См. рис. 136 и рис. 137.

д. Перенос модели предметной области явно на структуру таблиц невозможен, так как реляционная структура не предусматривает конструкции, аналогичные необходимым двунаправленным ассоциациям. Некоторые более близкие по структуре, чем транзитивное замыкание, модели хранения данных оказываются значительно менее эффективными с точки зрения работы с иерархиями. Транзитивное замыкание не представлено в модели предметной области, так как соответствующее понятие в предметной области отсутствует. *Дополнительная информация.* Модели предметной области и хранения данных в общем случае различаются, так как служат различным целям. Модель предметной области содержит ограничения и понятия предметной области, служит для реализации функциональности, полезной для заинтересованных лиц. Модель хранения данных обеспечивает эффективное использование внешней памяти или базы данных для поддержки функционирования модели предметной области.

```
addChild(child: Category, parent: Category) {
// Проверить, что у child и parent категорий нет
общих подкатегорий:
   ClosureLinkTable[*] cAs =
      dbContext.ListLinksGivenAncestor(child.id);
   ClosureLinkTable[*] pAs =
      dbContext.ListLinksGivenAncestor(parent.id);
   if (cAs.Intersect(pAs).Count() != 0)
      return;
   foreach (ClosureLinkTable cl in cAs) {
      ClosureLinkTable newCl =
      new ClosureLinkTable(ancestor = parent,
      descendant = cl.descendant, distance =
      cl.distance + 1);
      dbContext.AddLink(newCl);
   }}
```

Рис. 136

9.7

а. Значение не может быть корнем агрегата (ошибка также и в синтаксисе UML), а, значит, отчет *DiscountedSoldItemsReport* является сущностью, штрих-код *Barcode* не является глобально идентифицируемым, так как сам по себе представляет идентификатор (аналогично *Integer* и другим примитивным типам), а значит – является значением. Позиция заказа *OrderItem*, которая представляет собой пару (товар, количество) характеризуется значением структурных свойств, а значит, также является значением.

б. Купону *Coupon* в описанном сценарии не требуется ссылаться на маркетинговую кампанию *MarketingCampaign*, к которой он относится, также купону *Coupon* не требуется иметь ассоциацию к отчету *DiscountedSoldItemsReport*.

в. Нарушение границ агрегатов происходит, когда имеет место ассоциация, направленная извне агрегата к некорневой сущности или значению агрегата. В данном случае это направленные ассоциации: от отчета *DiscountedSoldItemReport* к купону *Coupon*, от кампании *MarketingCampaign* к купону *Coupon*, от отчета *DiscountedSoldItemsReport* к позиции заказа *OrderItem*, от товара *Product* к штрих-коду *Barcode*.

```
subtree(category: Category, maxDepth: Integer)
: Category[*] {
   ClosureLinkTable[*] ancestors =
      dbContext.ListLinksGivenAncestor(category.id);
   ClosureLinkTable rootLink;
   CategoryTable rootTable;
   Category root;
   for (Integer index = 0; index < ancestors.Count();
   index++) {
      ClosureLinkTable cl = ancestors[index];
      if (cl.distance <= maxDepth)
         ancestors.RemoveAt(index--);
      if (cl.distance == 0) {
         rootLink = cl;
         rootCategoryTable = cl.ancestor;
            root = new Category(id = rootTable.id,
            name = rootTable.name); } }
//далее рекурсивно проходя по списку ссылок
ancestors начиная с rootLink строится иерархия
категорий начиная с root;
return root; }
```

Рис. 137

г. Ошибки перечислены ниже, все исправления приведены на диаграмме (рис. 138):

– Ошибка в определении корней агрегатов: купон *Coupon* должен быть самостоятельным агрегатом, состоящим из одной сущности, поскольку может существовать отдельно от заказа *Order*. Указанное исправление также решает проблему нарушения границ агрегата *Order* направленными ассоциациями от отчета *DiscountedSoldItemReport* к купону *Coupon*, от кампании *MarketingCampaign* к купону *Coupon*.

– Нарушение границы агрегата направленной ассоциацией от отчета *DiscountedSoldItemsReport* к позиции заказа *OrderItem* может быть исправлено добавлением в агрегат *DiscountedSoldItemsReport* значения (пусть *SoldItem*), которое соответствует записи учета о продаже товара *Product* со скидкой и заменяет в этом смысле в агрегате отчет *DiscountedSoldItemsReport* позицию заказа *OrderItem*.

– Нарушение границ агрегата направленной ассоциацией от товара *Product* к штрих-коду *Barcode* может быть исправлено совместным использованием сущности штрих-код *Barcode* заказом *Order* и товаром *Product*. На диаграмме классов модели оба агрегата будут включать

штрих-код *Barcode* как внутреннее значение, на уровне экземпляров модели штрих-код разделяться агрегатами не будет. Это редкий пример «пересечения» агрегатов на уровне классов модели.

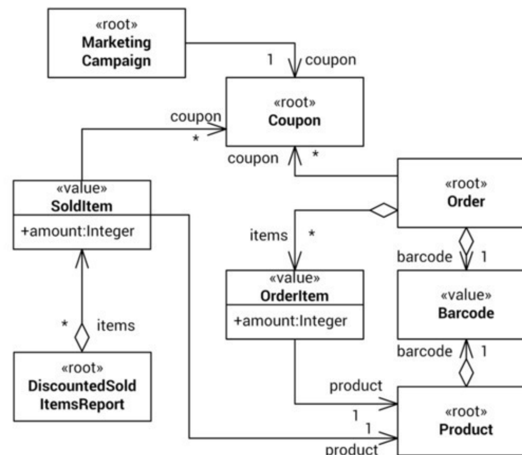


Рис. 138

9.8

а. Ограничение может быть нарушено двумя способами: 1) при добавлении новой вершины в список *vertices*, 2) при получении содержимого списка *vertices* и далее при прямом изменении координат *x*, *y* экземпляров вершин *Vertex*.

б. Для выполнения инварианта в модели необходимо: 1) сделать свойство *vertices* агрегата *Polygon* закрытым, добавить в сущность *Polygon* операцию *listVertices ():Vertex [*]*, возвращающую список копий *vertices* (это возможно, так как вершина *Vertex* является значением) 2) добавить операцию *addVertex (v: Vertex)* с проверкой значений координат *x*, *y* параметра *v*.

в. *Указание*. Псевдокод реализации операции *createRegular (vc: Integer, r: Real, x: Real, y: Real):Polygon* не приводится. Выполнение требуемого инварианта возможно благодаря установлению зависимости между фабрикой и конструируемым ею агрегатом, и обеспечению доступа фабрики к реализации агрегата. В данном случае операцию *addVertex* правильного многоугольника *Polygon* может использовать только фабрика многоугольников *PolygonFactory*.

г. Временная несогласованность агрегата в процессе конструирования не является нарушением ограничений предметной области. Но перед «выпуском» экземпляра фабрикой согласованность должна быть обеспечена и далее должна поддерживаться в течение всего времени жизни агрегата, но уже самим агрегатом. Именно поэтому часто фабрики имеют более тесную связь с конструируемым агрегатом – «знают» о внутреннем устройстве агрегата, нарушая в определенном смысле инкапсуляцию.

9.9.

а. Реализация метода в псевдокоде (см. ниже):

```
RegularPolygon.resize(vc:Integer) {
   Integer cvc = vertices.Count();
   if (vc == cvc)
      return;
   if (vc < cvc)
      for (Integer i = 0; i < cvc - vc; i++)
         this.expand();
   if (vc > cvc)
      for (Integer i = 0; i < vc - cvc; i++)
         this.reduce();
}
```

б. Фабрика многоугольников *PolygonFactory* может использовать операцию увеличения числа вершин на одну *expand* вместо операции *addVertex* при наличии у экземпляра правильного многоугольника *RegularPolygon* хотя бы двух вершин *Vertex* или наличия в правильном многоугольнике *RegularPolygon* координат центра описанной окружности *x*, *y* и радиуса *r*. Операция *addVertex* только добавляет вершины *Vertex*, не гарантирует сохранения условия регулярности («правильности») многоугольника *RegularPolygon*. Фабрика многоугольников *PolygonFactory* при использовании операции *addVertex* сама рассчитывает координаты вершин *Vertex* и, таким образом, гарантирует выполнение условия регулярности после завершения процесса конструирования. Операция *expand* обеспечивает выполнение условия регулярности, трансформирует один экземпляр правильного многоугольника *RegularPolygon* в другой. Конструирование экземпляра правильного многоугольника *RegularPolygon* фабрикой многоугольников *PolygonFactory* с использованием операции *addVertex* алгоритмически менее трудоемко, чем при многократном использовании операции *expand*. С другой стороны, если снижение производительности приемлемо, а реализация операции *expand* уже имеется, конструирование экземпляра правильного многоугольника *RegularPolygon* при помощи многократного использования операции *expand* возможна, так как приводит к меньшему объему программного кода и повторному использованию реализации поведения.

в. Нет, в реализации операции *resize* правильного многоугольника *RegularPolygon* заменить использование операции *expand* использованием операции *addVertex* нельзя. Объяснение см. выше. Операция *expand* применима, когда необходимо обеспечить сохранение условия регулярности многоугольника, например, в процессе жизненного цикла экземпляра агрегата правильного многоугольника *RegularPolygon*, после его создания. Операция *addVertex* применима, когда обеспечение условия регулярности многоугольника не требуется, например, в процессе конструирования, когда для экземпляра агрегата правильного многоугольника *RegularPolygon* инварианты могут не выполняться, то есть, когда регулярность обеспечивает другой участник модели, в приведенном примере – фабрика многоугольников *PolygonFactory*.

г. Совмещение поведения и данных в классах модели предметной области – важный принцип ООП, который является следствием инкапсуляции внутреннего состояния в случае абстрактного типа данных. Несоблюдение данного принципа приводит к созданию, так называемых, анемичных моделей предметной области, в которых классы либо превращаются в контейнеры данных, либо становятся контейнерами функций без состояния. Совмещение поведения и данных в одном классе модели или агрегате позволяет сохранить инкапсуляцию, обеспечить выполнение инвариантов, что становится проблемой в противном случае, так как поддержка инвариантов становится неявной, оказывается рассредоточенной по множеству классов модели. Все это влияет на удобство внесения изменений в модель и наличие в модели ошибок.

13.10. МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ В СТРУКТУРЕ КЛАССОВ

10.1

а. На диаграмме последовательности *Airline* посылает *Aircraft Producer* синхронное сообщение *est=estim ()*. Затем последовательно:

– *Airline* посылает *HullPlane Ind.* синхронное сообщение *requestHull ()*;

– В возвратном сообщении передается *hull*;

– *Airline* посылает *Engine Eng.* синхронное сообщение *requestEngines ()*;

– В возвратном сообщении передается *engine [2]*;

– *Airline* посылает *HullPlane Ind.* синхронное сообщение *requestEquip ()*;

– В возвратном сообщении передается *equip*;

– *Airline* посылает себе синхронное сообщение *assemble ()*.

б. *Airline* будет медиатором. Все асинхронные сообщения проходят через него. Чтобы не ограничивать порядок поставки запчастей, воспользоваться фрагментом *par*.

в. Добавить на диаграмму фрагмент *opt* после сообщения *assemble ()*. Сторожевое условие *[aircraft.price=est.price]*. Добавить синхронные сообщения *pay ()*: от *Airline* к *Aircraft Producer*, от *Airline* к *HullPlane Ind.*, от *Airline* к *Engine Engineering* в указанном порядке.

г. Порядок не определен.

10.2.

а. Структуру взаимодействия в виде кооперации показать на диаграмме внутренней структуры. Роли в кооперации: *record*, *patient* и *doctor*, две последние – активные. Соединители между *doctor* и *patient*, *doctor* и *record*. Взаимодействие показать на диаграмме последовательности. Линии жизни соответствуют ролям кооперации. Воспользоваться фрагментами *loop*, вложенными во фрагмент *par*. Условия циклов *loop* не уточняются. В циклах сообщения *symptoms* от *patient* к *doctor* и сообщения *question* от *doctor* к *patient* асинхронные. После фрагмента *par* синхронные сообщения: *write* от *doctor* к *record*, *diagnosis* и *prescription* от *doctor* к *patient*.

б. Кооперация *Meeting* содержит роль *employee* с кратностью больше нуля, а также безымянную роль типа *Director* с кратностью единица. На диаграмме последовательности во фрагменте *loop* роль: *Director* отправляет асинхронное сообщение *announcement*. После фрагмента *loop: Director* принимает от *employee* ровно одно синхронное сообщение *question* и отвечает на него ответным сообщением.

в. Интерфейс *IDoctor* принимает сигнал *Symptoms*. Класс *Patient* принимает сигнал *Questions*, содержит операции *diagnosis ()*, *prescription ()*. Класс *MedicalRecord* содержит операцию *write ()*. Интерфейс *IEmployee* содержит операцию *announcement ()*, класс *Director* содержит операцию *question ()*. Класс *Doctor* реализует интерфейсы *IDoctor*, *IEmployee*, содержит атрибуты *name*, *rank*. Ассоциация между *Director* и *Doctor*. Со стороны *Doctor* имя полюса ассоциации *employee*, кратность любая. Со стороны *Director* кратность единица. Ассоциация между *Patient* и *Doctor*. Имя полюса со стороны *Doctor* – *doctor*. Согласно ISP интерфейс класса *Doctor* разделён на *IDoctor* для взаимодействия с *Patient* и на *IEmployee* для взаимодействия с *Director*. Согласно SRP сущности предметной области инкапсулированы в отдельные классы.

г. Описание варианта использования.

Акторы: *Patient* (Пациент).

Цель: Узнать свой диагноз.

Предусловия: Есть жалобы/симптомы.

Постусловия: Получил диагноз, получил рецепт.

Основной сценарий:

1. Пациент сообщает системе свои жалобы.
2. Система задает вопрос пациенту и не дожидается ответа.
3. Система сообщает пациенту диагноз.
4. Система передает пациенту рецепт.

Альтернативные сценарии: -.

10.3.

а. Добавим класс *Worker* с операциями *visit (r: LivingRoom), visit (r: Kitchen)*. *Painter* и *Tiler* уточняют *Worker* и переопределяют операции *visit*. Добавим в класс *Room* операцию *accept (visitor: Worker)*.

б. См. рис. 139.

в. Кооперация *Repair* содержит роли: *Worker* и *Room*. Между ними соединитель. Кооперация *AppartmentRepair* содержит роли: две комнаты и два работника. См. рис. 140.

г. RFC (*Painter*) = 3, RFC (*LivingRoom*) = 2, DIT (*Painter*) = 1, DIT (*LivingRoom*) = 2.

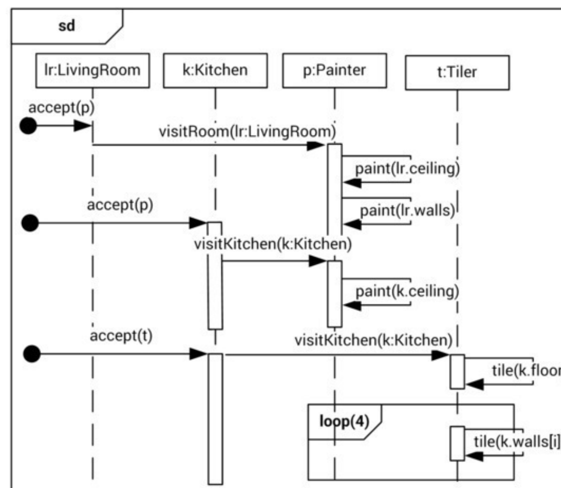


Рис. 139

10.4

а. Диаграмма использования: актор пользователь *User* связан ассоциацией с вариантом использования приветствовать *Greet*. Кратность полюса ассоциации со стороны актора *1..**.

Диаграмма внутренней структуры: *Greet* реализован кооперацией *SimpleGreeting*. В *SimpleGreeting* четыре роли: два пользователя *User*, две программы *Agent*. Одна роль типа *User* кратности единица связана соединителем с ролью *subject: Agent*. *Subject* связана с *object: Agent*. Роль *object* связан соединителем со второй ролью типа *User*. Поведение показано на рис. 141.

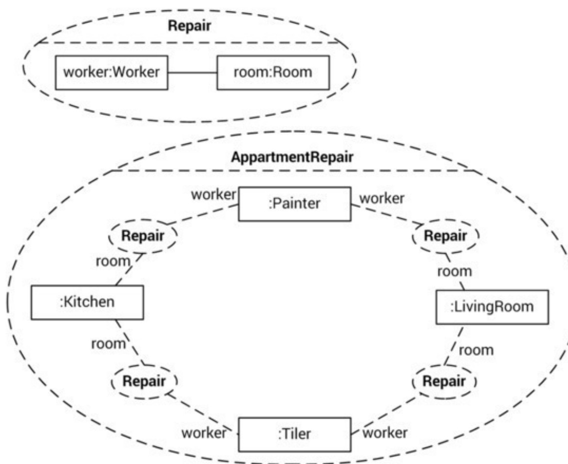


Рис. 140

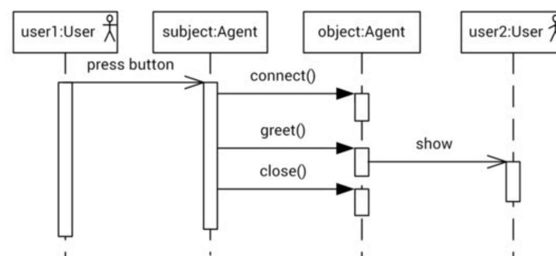


Рис. 141

б. Воспользуемся паттернами Стратегия (Strategy) и Шаблонный метод (Template Method). Параметризация приложения стратегией приветствия будет гибко менять поведение приветствия, при этом *subject* будет работать единообразно со всеми вариантами приветствия.

Шаблонный метод будем использовать, т. к. основной сценарий не меняется: сначала указать собеседников, затем *connect ()*, затем *greet ()*, в конце *close ()*.

Создадим кооперацию *Greeting* с ролями::S*ubject* (кратность 1),:O*bject* (кратность *) и *g: GreetingStrategy* (кратность 1). *GreetingStrategy* – абстрактный класс с операциями *addObjects (os: Object [*])* и *greet (o: Object)*. *GreetingStrategy* реализует операцию *greet* последовательным вызовом *o.connect ()*, *doGreet (o)*, *o.close ()*, реализуя протокол приветствия. Операция *doGreet* – абстрактная. Ее реализация требуется от наследников стратегии. Например, в пункте а. *SimpleGreetingStrategy* реализует операцию *doGreet (o: Object)* вызовом метода *o.show ()*.

Кооперация *EnglishGreeting* уточняет *Greeting*, замещая роль *GreetingStrategy* классом-наследником *English-GreetingStrategy*. См. рис. 142 и рис. 143.

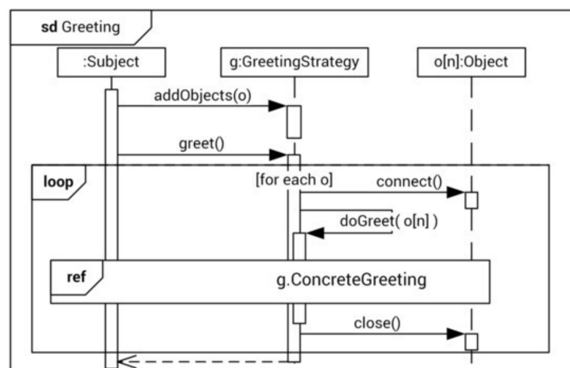


Рис. 142

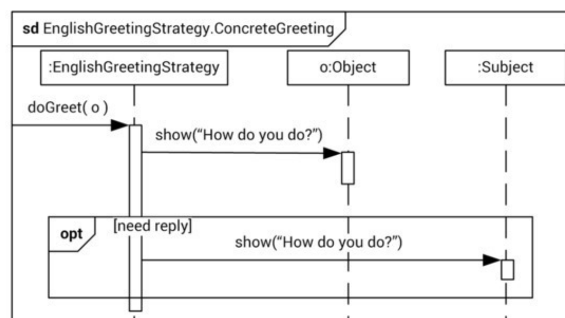


Рис. 143

в. Паттерн Стратегия открыт для дополнения. В случае встречи в своем офисе, уточним кооперацию *Greeting* кооперацией *HomeBusinessGreeting*. В этой кооперации заместим *GreetingStrategy* классом *HomeGreetingStrategy*. Операция *doGreet ()* в этом классе реализована синхронным сообщением *show («Very glad to see you»)*, после которого посылается асинхронное сообщение *handshake*. Стратегия в чужом офисе реализуется аналогично.

г. *Указание*. В реализации стратегии *ModernBusiness-GreetingStrategy* посылка *handshake* может быть отложена до завершения всех приветствий.

10.5.

а. Добавить сообщение *checkSeine ()* от: *OldMan* к *seine* после сообщения *pullAshore ()*.

б. См. рис. 144.

в. Поместить сообщение *add (f = smallfish)* во фрагмент *alt* с условием *[fish is nearby]*. Добавить в этот фрагмент секцию *[else]*. В секции *[else]* поместить сообщение *add (f = scum)* от *sea* к *seine*. Добавить на диаграмму линию жизни *scum: SeaScum*.

г. См. рис. 145.

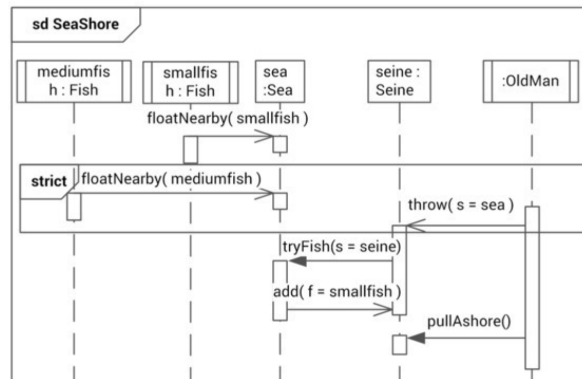


Рис. 144

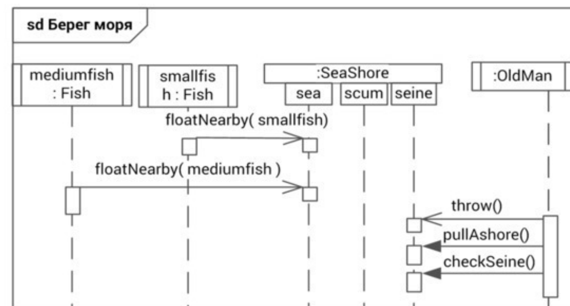


Рис. 145

10.6

а. Добавить в модель классы *TrafficLights*, *RailLights*, *Barrier*, которые уточняют класс *Equipment*. Отношения обобщения объединить во множество обобщений с исключением пересечений в наследовании *disjoint*. Добавить в модель классы *AutoConsole*, *ManualConsole*, которые уточняют класс *Console*.

б. Необходимо также добавить операцию *addConsole ()*. Различные классы, уточняющие *RCBuilder*, могут конструировать различные виды переездов, состоящих из железнодорожного пути и автомобильной дороги. Каждый такой класс может возвращать в результате специфичный класс переезда, уточняющий *RailCrossing*. См. рис. 146.

в. См. рис. 148.

г. См. рис. 147.

10.7.

а. См. рис. 149.

б. *Указание.* Следует использовать диаграмму последовательности для отображения взаимодействия. На диаграмме показать линии жизни *event*, *eventSource*, *visPlugin* в роли *eventListener* и *songPlugin* в роли *eventListener*. Указать вызов операции *generateEvent* в *eventSource*, указать вызов операции *dispatch* с параметром *event* от *eventSource* к *visPlugin*, который приводит к исполнению поведения, что отразить наличием спецификации выполнения. Далее, указать вызов *dispatch* в *lyricsPlugin*, при получении которого не указана спецификация выполнения.

в. См. рис. 150.

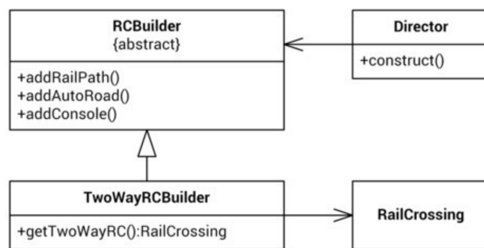


Рис. 146

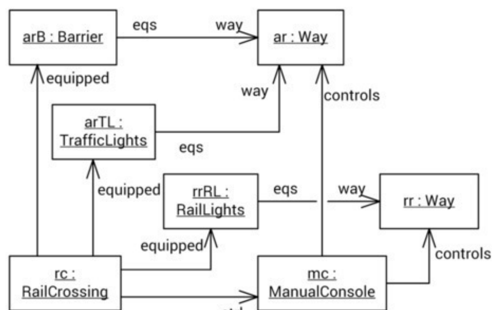


Рис. 147

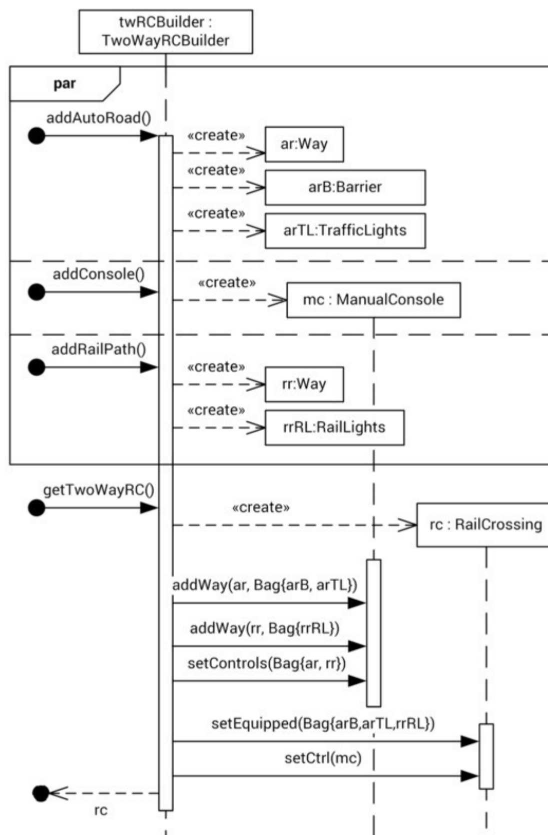


Рис. 148

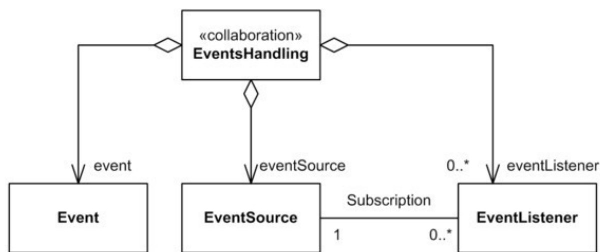


Рис. 149

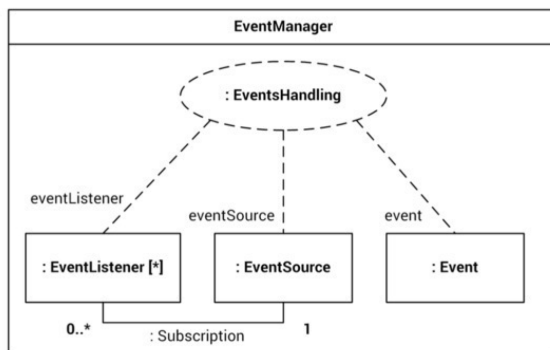


Рис. 150

10.8

а. См. рис. 151.

б. *Указание*. Обратите внимание на использование переходов по умолчанию и выполнение деятельностей *sustain*, *brake*, *open*, *close*. См. рис. 152.

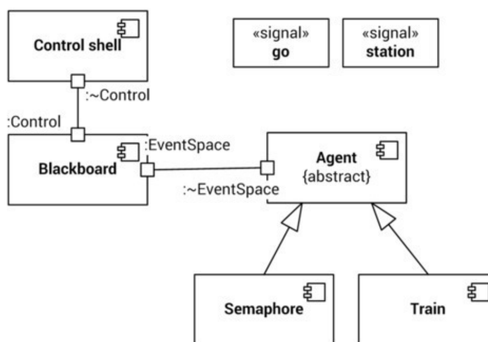


Рис. 151

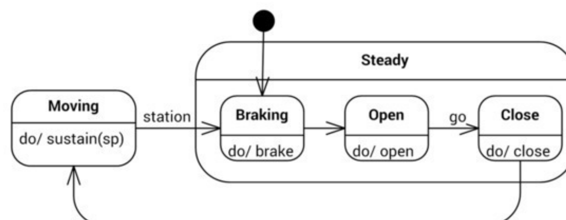


Рис. 152

в. В состояние *Steady* добавить состояние *Semaphore*. Из *Moving* переход по триггеру *state (s)* со сторожевым условием *[s=red]* в *Semaphore* и обратно по *state (s)* со сторожевым условием *[s=green]*. Деятельность при нахождении в *Semaphore* – *break*.

г. Компонент датчик температуры *TempSensor* уточняет *Agent*. Вложить построенную ранее схему состояний в новое состояние и добавить в нем ортогональный регион управления кондиционером. Поезд получает сигнал *temp* регулярно. См. рис. 153.

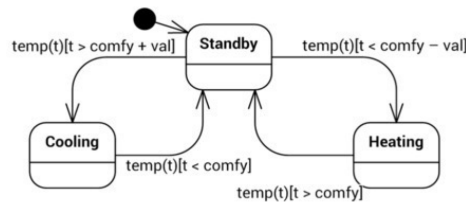


Рис. 153

10.9

а. Ввести перечисление *Dir* с направлениями движения робота *left*, *right*, *up*, *down*. Диаграмма состояний на рис. 154.

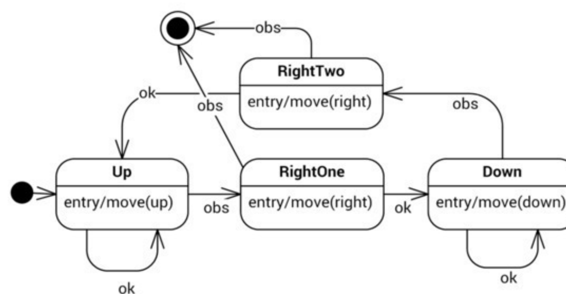


Рис. 154

б. Достаточно определить два состояния *Down* и *Left*. В *Down* переход из начального состояния, действие при входе – *move (down)*, по *ok* внешний переход в себя, по *obs* переход в *Left*. В *Left* действие при входе – *move (left)*, по *ok* внешний переход в себя, по *obs* переход в финальное состояние.

в. Определить состояние уборки *Work*, в которое поместить два вложенных *ToDownLeft* и *Spiral*. Оба эти состояния содержат соответствующие алгоритмы. Определить ортогональный регион в *Work*, в который поместить схему состояний, показанную на рис. 155.

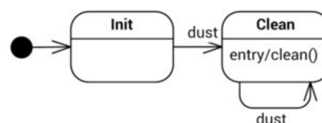


Рис. 155

г. Например, для комнаты из одной клетки с пылью. Принимаемая последовательность: *dust*, *obs*, *obs*, *obs*, *obs*. Отвергаемая: *obs*, *obs*, *obs*, *obs*, *ok*.

д. К классу *Robot* добавить структурное свойство *shift* типа *Integer* с квалификатором *dir: Dir* у *Robot* и кратности *1* у *shift*. Значения свойства инициализировать нулями. Переопределить метод *move* и записывать в *shift* количества шагов разного типа, которые делал робот. Вместо перехода в конечное состояние в регионе после *Spiral* добавить новое состояние *ReturnBack*, содержимое которого показано на рис. 157.

е. Добавить класс *ReturningRobot*, уточняющий *Robot*, определить в нем структурное свойство *shift* с квалификатором *dir*. Воспользоваться расширением схем состояний при наследовании для добавления состояния *ReturnBack* и переопределения перехода из *Spiral*. Обратите внимание на нотацию: пунктиром показаны элементы, которые не затронуты расширением. См. рис. 156.

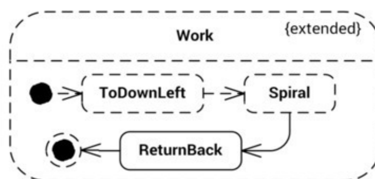


Рис. 156

ж. *Указание*. Обратите внимание на проблемы, возникающие с поддержанием общего состояния декоратора и экземпляра самого класса.

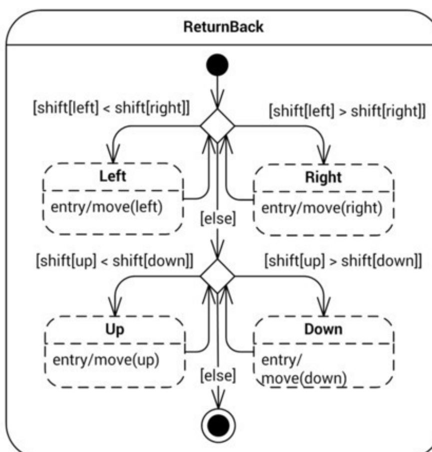


Рис. 157

10.10

а. См. рис. 158.

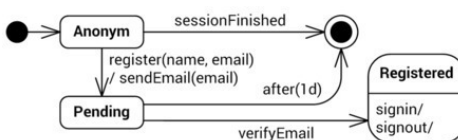


Рис. 158

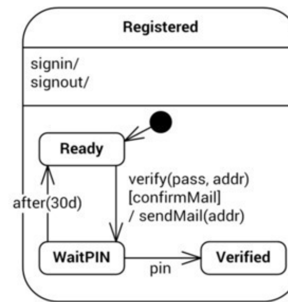


Рис. 159

б. См. рис. 159. На протяжении всего времени подтверждения учетной записи посетителю доступен вход и выход с портала.

в. Добавить состояние *WaitPass*, в него переход из *Ready* по *verify (pass, addr)* при условии *[confirmPass]*. Вложить *WaitPass* и *WaitPin* в новое *WaitConfirm* в том же регионе. Переход по *abort* из *WaitConfirm* в *Ready*.

г. Активный класс *Account* определяет операции:
register (name, email),
signin и *signout* без параметров,
sendEmail (email),
sessionFinished без параметров,
verify (pass, addr),
sendMail (addr),
pin без параметров.

Класс *Account* может не быть активным, если переходы по событиям времени заменить на вызовы операций, например, вместо перехода из *Pending* в конечное состояние по *after (1d)* указывать переход по вызову операции *emailExpired*.

10.11.

а. См рис. 160.

б. Добавить исходящий параметр типа *Step* в *AnalyzeStep*, направить объектный поток в новый входящий параметр *prevStep*, заключенный в отдельный набор контактов.

в. Поток управления к завершающему узлу деятельности перенаправить на новое действие *BuildDiagram*, на которое попадает входящий объектный поток из *Classes*, а результат передается в исходящий параметр деятельности типа *Diagram*. Добавить разделы деятельности («плавательные дорожки») *ProductOwner*, *Designer*, *Team*. Разместить *Elicit-Candidates*, *AnalyzeStep* в разделе *Team*, *SelectScenario* в разделе *ProductOwner*, *BuildDiagram* в разделе *Designer*.

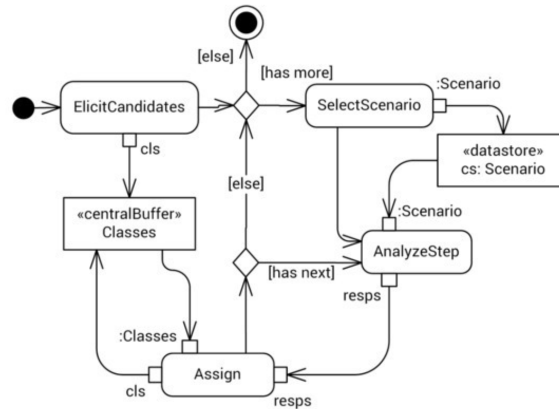


Рис. 160

г. В кооперации паттерна определены роли *mediator: Mediator* и с множественной кратностью *colleague: Colleague*. Кооперация CRC связывает роль *mediator* кооперации паттерна с *PM*, роль *colleague* с *ProductOwner*, *Team* и *Designer*. Класс *PM* реализует интерфейс *Mediator*, классы *ProductOwner*, *Team* и *Designer* реализуют интерфейс *Colleague*. Диаграмма последовательности приведена на рис. 161.

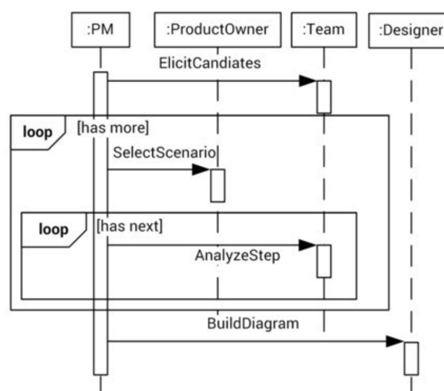


Рис. 161

ЛИТЕРАТУРА

[1] Фаулер. М. UML. Основы. Третье издание. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 192 с.

[2] Буч Г., Якобсон А., Рамбо Дж. UML Классика CS. 2-е Изд. / Пер. с англ. Под общей редакцией проф. С. Орлова – СПб.: Питер, 2006. – 736 с.; ил.

[3] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – Пер. с англ. – СПб.: Питер, 2007. – 366 с.

[4] OMG, UML v2.4.1 Superstructure, http://www.omg.org/ spec/UML/ [Электронный ресурс, получено 20.06.2012]

[5] Эванс, Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. – М.:Вильямс, 2017. – 448 с.

[6] Budgen, D., Software Design (2nd Edition) 2nd Ed. Addison-Wesley; 2003. – 400 p.

[7] Роберт С. Мартин, Мика Мартин. Принципы, паттерны и методики гибкой разработки на языке C#. – М.: Символ-Плюс, 2011. – 768 с.

[8] Ларман, К. Применение UML 2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ, проектирование и итеративную разработку. – М.: Вильямс, 2013. – 736 с.

[9] Bass, L., Clements, P., Kazman, R. Software Architecture in Practice 3rd ed. – Addison Wesley, 2012. – 640 p.

[10] Pressman, R., Maxim, B., Software Engineering: A Practitioner’s Approach 8th Ed. – McGraw-Hill Education, 2014. – 976 p.

[11] Mary Shaw and David Garlan. Software Architecture: Perspectives on an Emerging Discipline. – Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. – 242 p.