Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Оренбургский государственный университет»

Н.А. Соловьев, Е.Н. Чернопрудова

СИСТЕМЫ АВТОМАТИЗАЦИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рекомендовано Ученым советом федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Оренбургский государственный университет» в качестве учебного пособия для студентов, обучающихся по программам высшего профессионального образования направления 230100.68 Информатика и вычислительная техника, 231000.68 Программная инженерия

Оренбург
2012

УДК 681.3(075.8)
ББК 32.973.26-018
С 60

Рецензент - доктор экономических наук, профессор, заведующий кафедрой управления и информатики в технических системах В.Н. Шепель

Соловьев, Н.А.
С 60 Системы автоматизации разработки программного обеспечения: учебное пособие / Н.А. Соловьев, Е.Н. Чернопрудова, Оренбургский гос. ун-т. – Оренбург: ОГУ, 2012. – 191 с.

В учебном пособии рассмотрены методологические основы построения систем автоматизации разработки программного обеспечения на основе универсального языка моделирования UML. Теоретический материал дополнен примерами автоматизированного проектирования программной системы с аналитическим приложением на основе методов теории статистических решений, вопросами для проверки усвоения материала.

Учебное пособие предназначено для студентов, обучающихся по программам высшего профессионального образования направления 230100.68 Информатика и вычислительная техника, 231000.68 Программная инженерия при изучении дисциплин «Автоматизация и проектирование информационно-телекоммуникационных ситсем» и «Системы автоматизированного проектирования программного обеспечения».

УДК 681.3(075.8)
ББК 32.973.26-018

ISBBN

Содержание

Введение

Создание автоматизированных информационных систем (АИС) – весьма сложная и трудоемкая задача в связи с тем, что современное программное обеспечение (ПО) данного класса составляет сотни тысяч операторов. Будущий специалист в области разработки ПО должен иметь представление о современных методах автоматизации анализа, проектирования, реализации и тестирования АИС, т.е. ориентироваться в современных подходах к технологиям программирования.

Теоретической основой построения систем автоматизации проектирования ПО (САПР ПО) являются методы технологии разработки ПО, автоматизация которых является предметом изучения настоящего учебного пособия. Изложение материала строится в соответствии с основными этапами жизненного цикла ПО.

Основой изложенного материала стал учебник «Технология программирования», разработанного в МГТУ им. Н. Баумана профессором Г.С. Ивановой, и допущенного Министерством образования и науки Российской Федерации (Минобрнауки РФ) для студентов ВУЗов, обучающихся по направлению 2301000 – Информатика и вычислительная техника. Материал доработан в процессе многолетней апробации на кафедре программного обеспечения вычислительной техники и автоматизированных систем «Оренбургский государственный университет».

1 Методология автоматизации разработки программного обеспечения

1.1 Актуальность автоматизации разработки программного обеспечения

Производство программного обеспечения сегодня – крупнейшая отрасль мировой экономики, в которой занято около 3-х млн. специалистов. Еще несколько млн. человек напрямую зависят от качества корпоративных автоматизированных информационных систем (АИС).

Поэтому состояние отрасли напрямую определяет благополучие специалистов-разработчиков программного обеспечения (ПО).

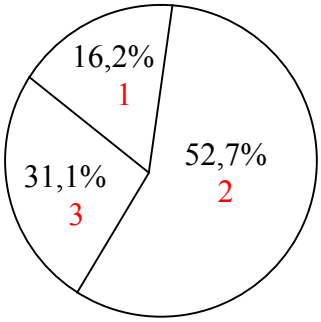
1.1.1 Кризис программной инженерии, его причины и пути преодоления

Проектирование корпоративных АИС – логически сложная, трудоемкая и длительная работа, требующая высокой квалификации участвующих в ней специалистов. Однако до настоящего времени проектирования АИС нередко осуществляется на интуитивном уровне неформализуемыми методами, включающими в себя элементы искусства, практический опыт и дорогостоящие экспериментальные проверки качества функционирования системы. Кроме того, в процессе создания и функционирования АИС информационные потребности пользователей постоянно изменяются или уточняются, что еще более осложняет разработку и сопровождение таких систем.

В конце XX-го века в программной инженерии сложилось критическая ситуация, неразрешенная до сих пор. Кризис выражается в том, что большие проекты ПО стали выполняться с отставанием графика и со значительным превышением расходов, а разработанный продукт не обладал требуемыми функциональными возможностями или производительностью, что не устраивает потребителей. Так, например,

в 1995 г. компания Standish Group проанализировала работу 364 американских корпораций по итогам выполнения более 23 000 проектов, связанных с разработкой ПО.

Результаты анализа, представленные на рисунке 1.1, оказались удручающими.



1 – доля проектов, завершившихся в срок без превышения бюджета;
2 – доля проектов, завершившихся с опозданием, превышением расходов, функционал которых не реализован в полном объеме;
3 – доля аннулированных проектов

Рисунок 1.1 – Результаты анализа проектов в области программной инженерии

Причины кризиса:

- нечеткая и неполная формулировка требований к ПО;
- недостаточное вовлечение пользователей в работу над проектом;
- отсутствие необходимых ресурсов и неудовлетворительное планирование;
- частое изменение требований спецификаций;
- новизна используемой технологии для организации;
- отсутствие грамотного управления проектом.

В конце 20-го века утвердилось понимание необходимости перехода от кустарных к индустриальным технологиям создания ПО, к созданию совокупности инженерных методов и средств разработки программных продуктов, объединенных общим названием «программная инженерия» (software engineering). Тогда же появилось первое издание, посвященное программной инженерии – IEEE Transaction on Software Engineering.

В основе программной инженерии лежит фундаментальная идея: *разработка ПО является формальным процессом и, следовательно, его можно автоматизировать.*

Таким образом, автоматизация разработки программного обеспечения является **актуальной** инженерной задачей в предметной области специалистов ПОВТАС.

1.1.2 Тенденции развития современных автоматизированных информационных систем

Предметной областью специалистов ПОВТАС являются АИС. Как отмечал Фредерик Брукс, руководитель проекта операционной системы OS/360, самым существенным свойством программных систем (ПС), к классу которых относится АИС, является их сложность. Благодаря уникальности и несхожести своих составных частей АИС принципиально отличается от технических систем, в которых преобладают повторяющиеся элементы.

Тенденциями развития АИС в современных условиях становятся:

– сложность описания (большое количество функций, процессов, элементов данных и сложные взаимосвязи между ними);

– наличие совокупности тесно взаимодействующих компонентов, имеющих локальные задачи и цели функционирования (например, традиционных приложений, связанных с обработкой транзакций, приложений аналитической обработки данных – поддержки принятия решений);

– отсутствие полных аналогов корпоративных АИС, ограничивающие возможность использования типовых проектных решений;

– необходимость интеграции существующих и вновь разрабатываемых приложений;

– функционирование в неоднородной среде на нескольких аппаратных платформах;

– разобщенность и разнородность групп разработчиков по уровню квалификации и сложившимся традициям использования тех или иных инструментальных средств;

– значительная временная протяженность проекта.

Нелинейность роста сложности ПС при увеличении размера системы становится причиной затруднений, возникающих в процессе общения между разработчиками, понимания ими всех возможных состояний программ, ведет к ошибкам в продукте.

В 80-90-х гг. прошлого века при разработке ПО применялись методы, базирующиеся на строго формализованных способах описания ПО и принимаемых технических решений. Однако широкое применение этих методов при разработке конкретных АИС сдерживалось отсутствием адекватных инструментальных средств. Неавтоматизированная разработка АИС сводила преимущества их форматизированного описания к нулю.

Таким образом, к концу XX-го века назрела **необходимость** разработки программно-технологических средств специального класса, реализующих CASE-технологии создания и сопровождения ПО АИС.

CASE-технология представляет собой совокупность методов проектирования ПО, а также набор инструментальных средств автоматизации, позволяющих в наглядной форме моделировать предметную область, анализировать модель на всех стадиях разработки и сопровождения ПО, а также разрабатывать приложения в соответствии с информационными потребностями пользователей.

1.1.3 Цели, задачи и структура учебного пособия

Цель учебного пособия – изложить современные методы и средства автоматизации разработки ПО АИС на основе CASE-технологии.

Структура учебного пособия представлена на рисунке 1.2.

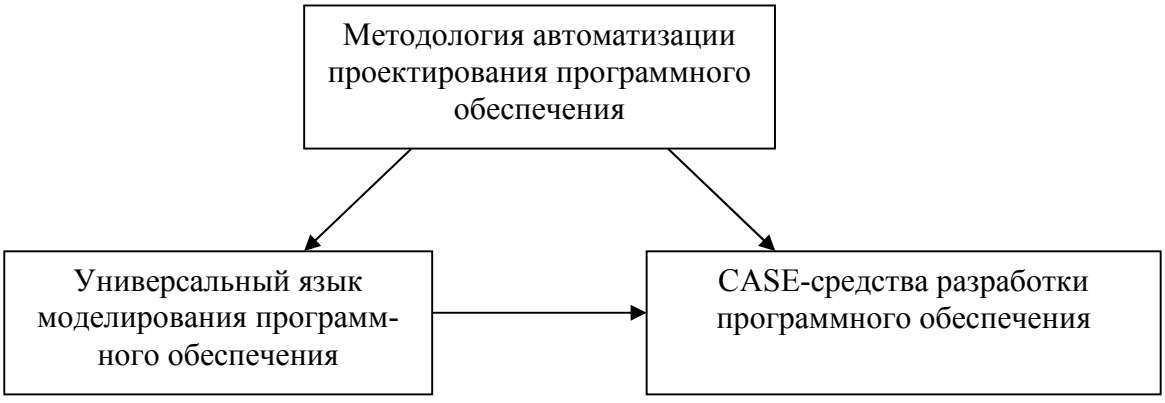


Рисунок 1.2 – Структура учебного пособия

Задачи учебного пособия

- осветить с системных позиций основные направления исследований, существующие в области программной инженерии;
- рассмотреть современное состояние развития CASE-средств и промышленных технологий разработки ПО;
- изучить унифицированный язык объектно-ориентированного моделирования UML и визуальный редактор на его основе – Rational Rose.

1.1.4 Вопросы и задания для самоконтроля

1 Перечислите причины кризиса программной инженерии.

2 Какая идея лежит в основе программной инженерии?

3 Каковы тенденции развития современных АИС?

4 Дополните определение: «CASE-технология представляет собой совокупность методов проектирования АИС, а также…».

5 Какие методы применялись в 80-90-х годах прошлого века при разработки программного обеспечения (ПО).

1.2 Методологические основы разработки программного обеспечения

Одним из базовых понятий методологии разработки АИС является понятие жизненного цикла (ЖЦ) ее программного обеспечения (ПО). ЖЦ – непрерывный процесс, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ISO/IEC 12207 (ISO – International Organization of Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике). Стан-

дарт определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

Одним из этапов ЖЦ ПО является проектирование – быстро развивающееся направление исследований в области программной инженерии. Опыт ведения реальных разработок и совершенствование имеющихся программно-аппаратных средств постоянно переосмысливается, в результате чего появляются новые технологии и методы их реализации, которые, в свою очередь, служат основой более современных средств разработки ПО.

1.2.1 Сущность технологии разработки программного обеспечения

Технологии и инструментальные средства разработки составляют основу проекта любой программной системы (ПС). Технологии реализуются через конкретные методы и поддерживающие их стандарты, методики и инструментальные средства, которые обеспечивают процессы реализации различных этапов ЖЦ ПО.

Спиральная модель жизненного цикла ПС, изображенная на рисунке 2.1, наиболее полно отвечает современным подходам к разработке ПО, т.к. предполагают, что технологические процессы выполняются итерационно.

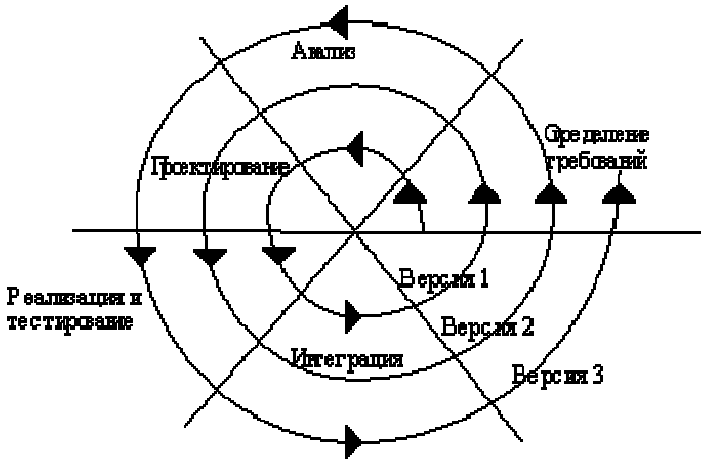


Рисунок 1.3 – Спиральная модель ЖЦ ПС

Основными этапами итерационного процесса являются: постановка задачи, анализ, проектирование, реализация и интегрирование. Наиболее сложным и трудо-

емким этапом является проектирование ПО.

В настоящее время определение технологии проектирования ПО не имеет устоявшейся формулировки. Учитывая то, что проектирование является одним из этапов ЖЦ ПО, можно остановиться на следующем определении: **технология проектирования ПО это совокупность методов и средств, используемых в процессе создания программных продуктов.**

Как и любая технология, технология программирования представляет собой набор технологических процессов, включающих:

- указание последовательности выполнения технологических операций, обобщенная схема которых представлена на рисунке 1.4;
- перечисление условий, при которых выполняется та или иная операция;
- описание самих операций, каждой из которых ставится в соответствие исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки.

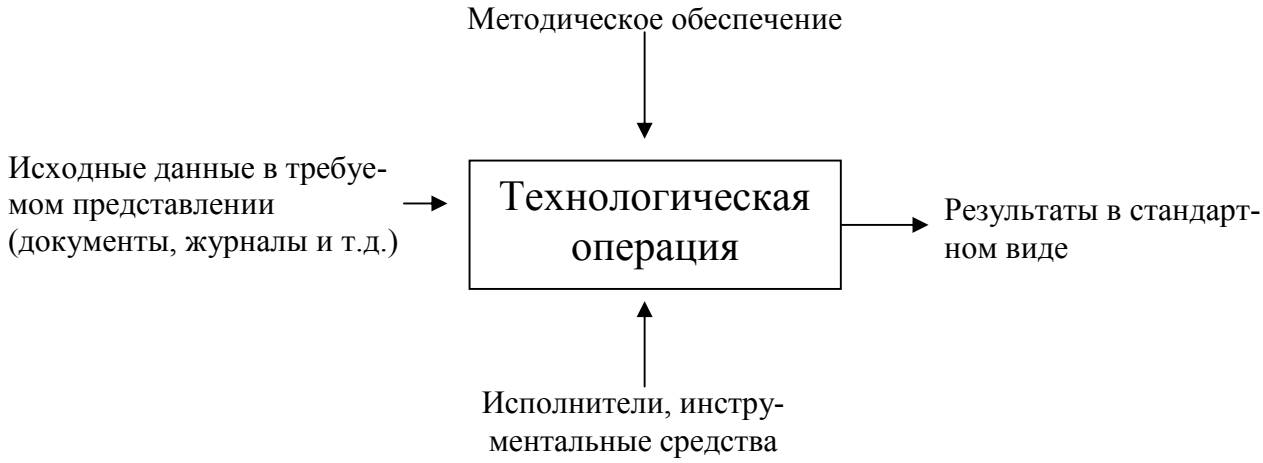


Рисунок 1.4 – Описание технологических операций

Кроме операций и их последовательности, технология определяет метод описания архитектуры проектируемой системы, т.е. модели, используемые на конкретном этапе разработки ПО.

Метод проектирования ПО представляет собой организованную совокупность информационных процессов создания ряда моделей, которые описывают раз-

личные аспекты разрабатываемой системы с использованием четко определенных технологических операций.

На формальном уровне метод определяется как совокупность составляющих языка моделирования:

- **концепций** (теоретических основ). В качестве таких основ выступают структурный или объектно-ориентированный подходы (парадигмы) программирования;
- **нотаций,** используемых для построения моделей спецификации статической структуры и динамики поведения проектирования АИС. В качестве таких нотаций обычно используются графические диаграммы (диаграммы потоков данных, диаграммы «сущность-связь», диаграммы вариантов использования (структурный подход), диаграммы классов (ООП));
- **руководства (правила),** определяющих практическое применение метода (последовательность и правила построения моделей, критерии, используемые для анализа результатов).

На рисунке 1.5 представлена структура языка моделирования, отражающего метод описания программного продукта.

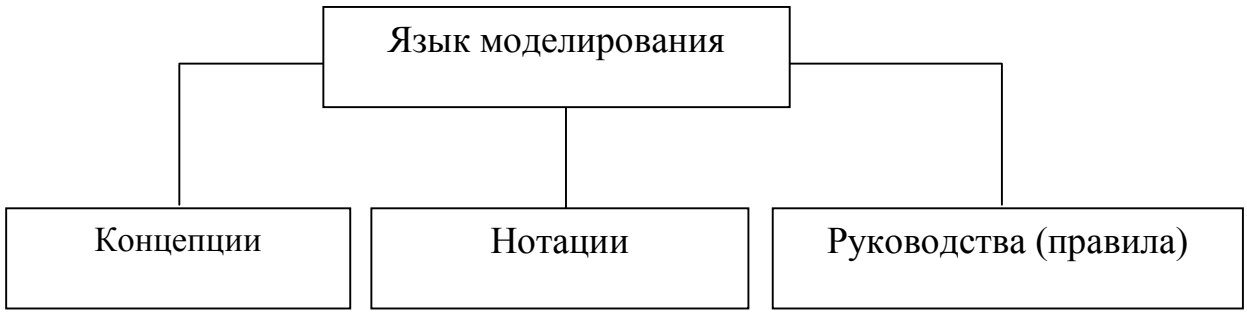


Рисунок 1.5 – Составляющие языка моделирования

К сожалению, в настоящее время не существует общепризнанного определения архитектуры ПО. Данное понятия определяется различными способами, например, «Программная архитектура есть абстрактная спецификация системы, состоящая из основных функциональных компонентов, описываемых в терминах их поведения, их интерфейсов и межкомпонентного взаимодействия» (Хэйес-Рос). «Архитектура есть структура компонентов программы-системы, их взаимосвязи, правила и

руководящие принципы организации ее проектирования и дальнейшей эволюции» (Галэн, Пэрри).

В основополагающем учебнике под архитектурой понимается совокупность базовых концепций (принципов) её построения, которые определяются сложностью решаемых задач, степенью универсальности ПО и числом пользователей, одновременно с ним работающих.

Несмотря на отличия, имеющиеся в определениях программной архитектуры, в каждом из них делается акцент на структурные аспекты организации ПО. Отсюда наиболее адекватным необходимо признать следующее определение – **программная архитектура** представляет собой совокупность моделей структурных элементов системы с видимыми извне свойствами и механизмами их взаимодействия.

Различают одно и многопользовательскую архитектуры.

Однопользовательские архитектуры реализуют в виде:

– программа или программное средство (адресованный компьютеру *набор инструкций*, точно описывающий последовательность действий, которые необходимо выполнить для решения конкретной задачи);

– пакета программ (*совокупность программ*, решающих задачи некоторой предметной области, например, библиотека программ);

– программной системы (организованная совокупность программ, позволяющих решать широкий класс задач из одной предметной области);

– программного комплекса (*совокупность программных систем*, обеспечивающих решение класса сложных задач предметной области).

Многопользовательские программные системы организуют сетевое взаимодействие отдельных компонентов ПО, построенные по принципам «файл-сервер», «клиент-сервер» и т.д.

Реальное применение любой технологии проектирования, разработки и сопровождения АИС в конкретной организации и конкретном проекте невозможно без ряда стандартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта. К таким стандартам относятся:

– стандарт проектирования;

– стандарт оформления проектной документации;

– стандарт пользовательского интерфейса.

Содержание стандартов рассматривается в курсе ТРПО.

Для успешной реализации проекта объект проектирования (АИС) должен быть прежде всего адекватно описан, т.е. должны быть построены полные и непротиворечивые модели архитектуры ПО.

Модели представляют собой средства для визуализации описания, проектирования и документирования архитектуры программной системы. По мнению одного из авторитетных специалистов в области программной инженерии Гради Буча, моделирование является центральным звеном всей деятельности по созданию ПО.

Модели строятся для того, чтобы понять и осмыслить архитектуру и поведение будущей ПС, облегчить управление процессом ее создания и документировать принимаемые проектные решения.

Таким образом, методы проектирования составляют центральную часть формализованной дисциплины выполнения проекта любого ПО и является основой совершенствования технологий.

1.2.2 Эволюция технологий проектирования программного обеспечения

Известная формула Вирта «алгоритмы + структура данных = программа» свидетельствует, что в недрах ПО существуют два начала, две противоположности, находящиеся, как водится, в диалектическом единстве и борьбе. Одно начало императивное, алгоритмическое, а другое – декларативное, непроцедурное, основанное на моделях.

Для уточнения содержания технологий и определения тендеции их развития, целесообразно рассмотрение технологий проектирования ПО в историческом аспекте. На рисунке 1.6 показана эволюция технологий проектирования ПО – объективный

процесс единства и борьбы названных противоположностей, движущей силой которого является увеличение сложности разрабатываемых программных продуктов.

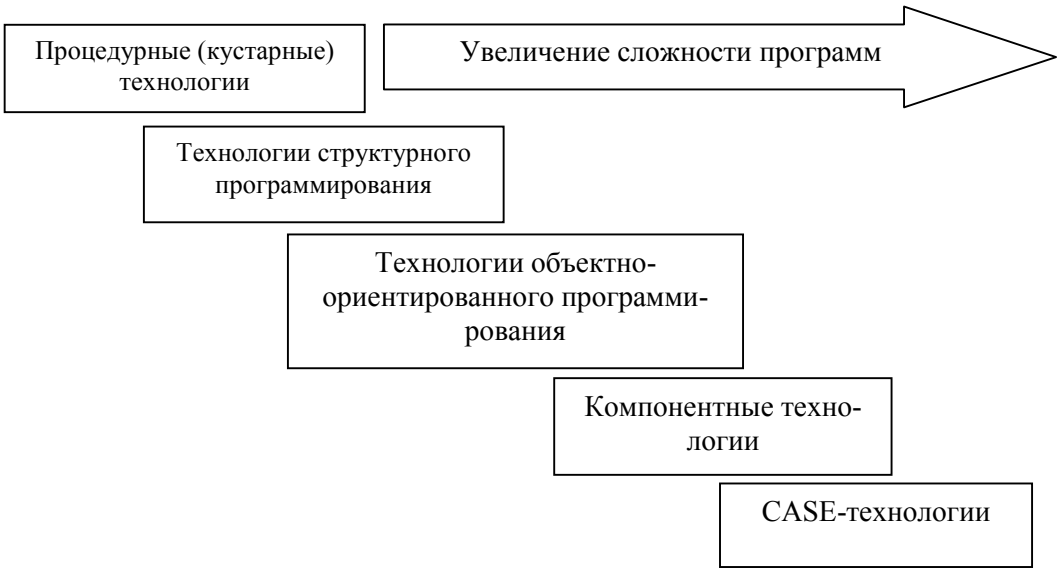


Рисунок 1.6 – Эволюция технологий проектирования

С теоретической точки зрения технологии различаются содержанием и последовательностью технологических операций, методами их описания и архитектурой разработанного ПО.

Технологии процедурного (стихийного) программирования. Заря информационных технологий была ознаменована полным, безраздельным торжеством алгоритмического начала. Это начало, чуждое стилю человеческого мышления, которое практически не использует алгоритмическую форму для изложения своих результатов, привело к возникновению новой профессии – программистов. В этот период (40..60 гг. XX-го века) программирование фактически являлось искусством. Программисты надолго оттеснили от компьютеров специалистов предметных областей знаний.

Последовательность технологических операций, характерная для технологий процедурного программирования, представлена на рисунке 1.7.

Первые программы имели простейшую архитектуру и состояли собственно из программ (процедур на машинном языке) и обрабатываемых данных.

Рисунок 1.7 – Последовательность операций технологии процедурного программирования и их исполнители

На верхнем уровне рисунка 1.8 изображена архитектура таких программ.

Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и место нахождения данных в физической памяти.

Появление АССЕМБЛЕРА и высокоуровневых языков FORTRAN, ALGOL позволило повысить сложность разрабатываемых ПО за счет использования подпрограмм. Однако слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части данных, которые не делились на глобальные и данные подпрограмм.

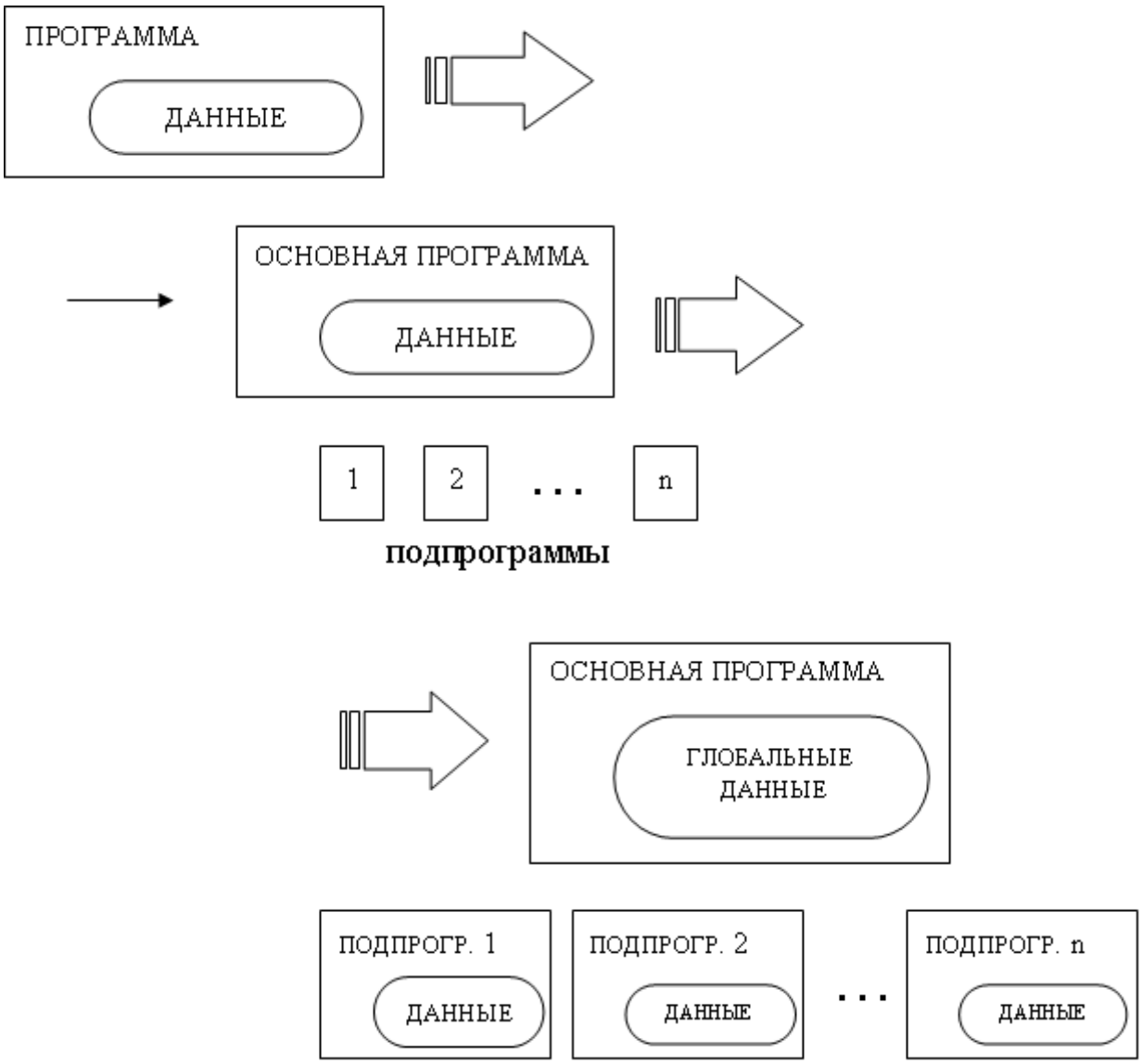


Рисунок 1.8 – Развитие архитектуры программ при технологиях процедурного и структурного программирования

Кроме того, разрабатываемое ПО с локальными данными по прежнему ограничивалось возможностью программиста отслеживать процессы их обработки. Это предопределило возникновение первого «кризиса программирования» (60-е годы XX-го века) – колоссальные успехи в области развития средств вычислительной техники пришли в противоречие с низкой производительностью труда программистов, отсюда проекты устаревали раньше, чем были готовы к внедрению. Появление операционных систем снизило остроту проблем. Однако оставалась разработка ПО «снизу-вверх» – подход при котором сначала разрабатывались сравнительно простые подпрограммы, из которых затем пытались построить сложную программу.

Отсутствие четких моделей описания подпрограмм и методов их проектирования, создание каждой подпрограммы превращалось в непростую задачу: интерфейсы подпрограмм получались сложными и при сборке программного продукта выявлялось большое количество ошибок согласования (80% времени разработки ПО уходило на тестирование).

Таким образом, эволюция технологий разработки ПО является объективной реальностью и определяется необходимостью преодоления проблем, связанных с ростом сложности ПО, отсутствием автоматизированных средств описания программных систем, потребностью в коллективной разработке и увеличению степени повторяемости программного кода.

1.2.3 Вопросы и задания для самоконтроля

1 Дайте определение технологии проектирования ПО?

2 Что понимают под архитектурой ПО?

3 Что представляют собой модели ПО?

4 В каких случаях строятся модели?

5 Что является центральным процессом моделирования? Что включает в себя язык моделирования?

6 Перечислите последовательность операций технологии процедурного программирования.

7 Какие объекты включает в себя технологические операции?

8 Дайте определение методу проектирования.

9 В чем заключается сущность стихийного программирования?

10 Перечислите и поясните последовательность операций технологий процедурного программирования и их исполнителей.

1.3 Базовые технологии разработки программного обеспечения

Программирование – сравнительно молодая и быстро развивающаяся отрасль науки и техники. Опыт ведения реальных разработок и совершенствование имеющихся программно-аппаратных средств постоянно переосмысливается, в результате чего появляются новые технологии и методы их реализации, которые, в свою очередь, служат основой более современных средств разработки ПО. Однако, в основе любой технологии лежат две базовые парадигмы: структурное и объектно-ориентированное программирование.

1.3.1 Технологии на основе парадигмы структурного программирования

Дальнейший рост сложности разрабатываемого ПО потребовал структурирования данных и, как следствие, в языках появилась возможность определения пользовательских типов данных. Кроме того, анализ причин возникновения большинства ошибок технологии процедурного программирования позволил сформулировать новый подход к программированию, который был назван «структурным».

Сущность структурного подхода к разработке АИС заключается в **декомпозиции** проектируемой системы на автоматизируемые функции: система разбивается на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и так далее. В отличии от используемого ранее процедурного подхода к декомпозиции, структурный подход предполагает представление задачи **в виде иерархии** подзадач простейшей структуры (40..50 операторов). Проектирование осуществляется «свреху-вниз» и подразумевает реализацию общей идеи за счет разработки интерфейсов подпрограмм, а также специальный метод проектирования алгоритмов – **метод пошаговой детализации**.

При этом последовательность технологических операций, характерная для

технологий структурного программирования, практически не изменилась (см. рисунок 1.8).

Все наиболее распространенные технологии структурного подхода базируются на ряде общих принципов:

– принцип "разделяй и властвуй" - принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;

– принцип иерархического упорядочивания - принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

– принцип абстрагирования - заключается в выделении существенных аспектов системы и отвлечения от несущественных;

– принцип структурирования данных - заключается в том, что данные должны быть структурированы и иерархически организованы.

Поддержка принципов структурного программирования заложено в основу так называемых структурных языков программирования – Pascal, C, PL/1.

В структурном анализе используются модели, иллюстрирующих функции, выполняемые системой, и модели, описывающие отношения между данными:

– SADT (Structured Analysis and Design Technique) – функциональные модели;

– DFD (Data Flow Diagrams) – диаграммы потоков данных;

– ERD (Entity-Relationship Diagrams) – диаграммы «сущность-связь».

Технология SADT разработана Дугласом Россом и на ее основе принят известный стандарт IDEF0 (Icam DEFinition), который является основной частью программы ICAM (Интеграция компьютерных и промышленных технологий). Технология возникла под влиянием PLEX, концепции клеточной модели человеко-ориентированных функций Хори, общей теории систем, технологий программирования и кибернетики.

SADT представляет собой совокупность концепций, нотаций и правил, предназначенных для построения функциональной модели объекта предметной области.

Элементы этой технологии основываются на концепции графического пред-

ставление блочного моделирования – SADT-диаграммы отображают функции в виде блоков, взаимодействующих друг с другом посредством интерфейсных дуг. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, показана с левой стороны блока, а результаты показаны с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу.

На рисунке 1.9 представлена основная **нотация** SADT-модели любой предметной области.

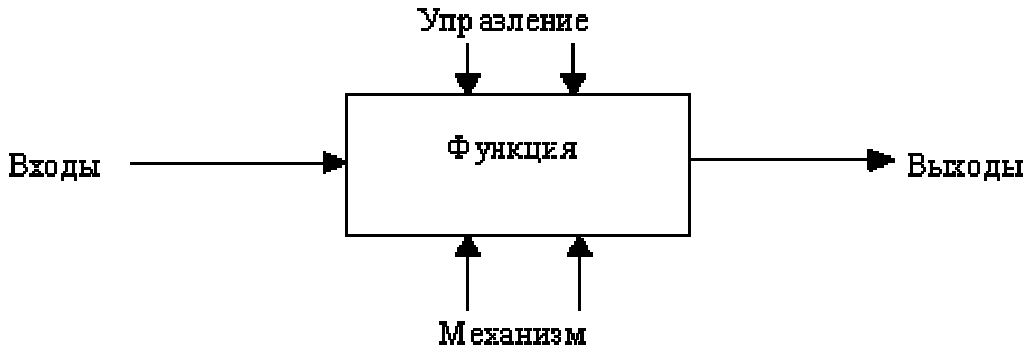


Рисунок 1.9 – Функциональный блок и интерфейсные дуги

Правила SADT включают:

– ограничение количества блоков на каждом уровне декомпозиции (правило 3-6 блоков);

– связность диаграмм (номера блоков), уникальность меток и наименований (отсутствие повторяющихся имен);

– разделение входов и управлений (правило определения роли данных) и отделение организации от функций, т.е. исключение влияния организационной структуры на функциональную модель.

Одной из наиболее важных особенностей технологии SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель. На рисунке 1.10 приведены четыре уровня диаграммы SADT-модели и их взаимосвязи.

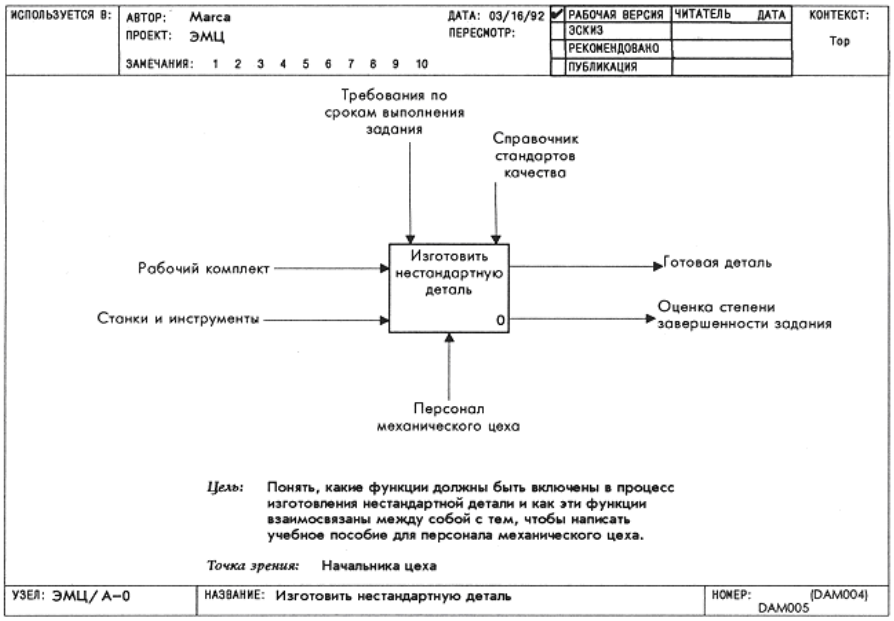


Рисунок 1.10 – Декомпозиция диаграмм SADT-модели

Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждая диаграмма иллюстрирует «внутреннее строение» блока на родительской диаграмме. Пример SADT-модели показан на рисунках 1.11 и 1.12, полученные в CASE-среде BPwin.

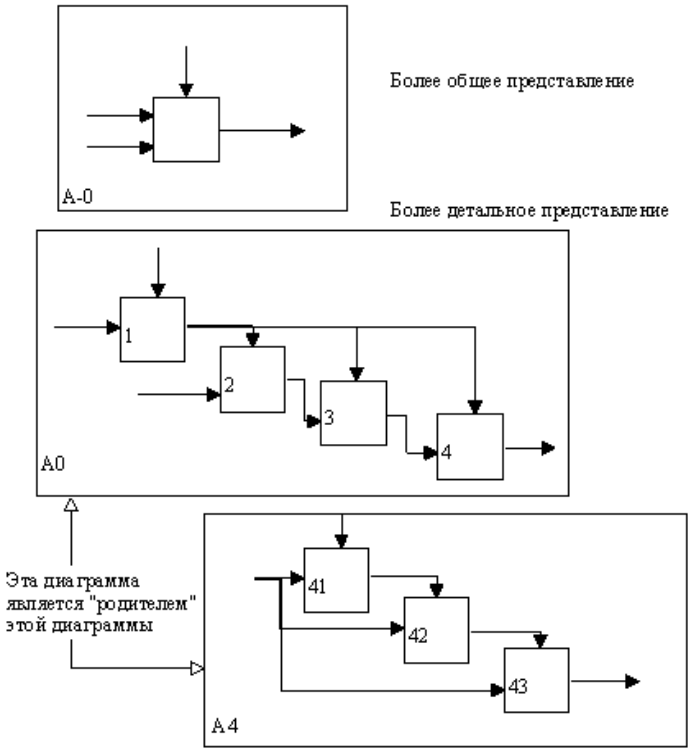


Рисунок 1.11.- Исходная диаграмма SADT-модели

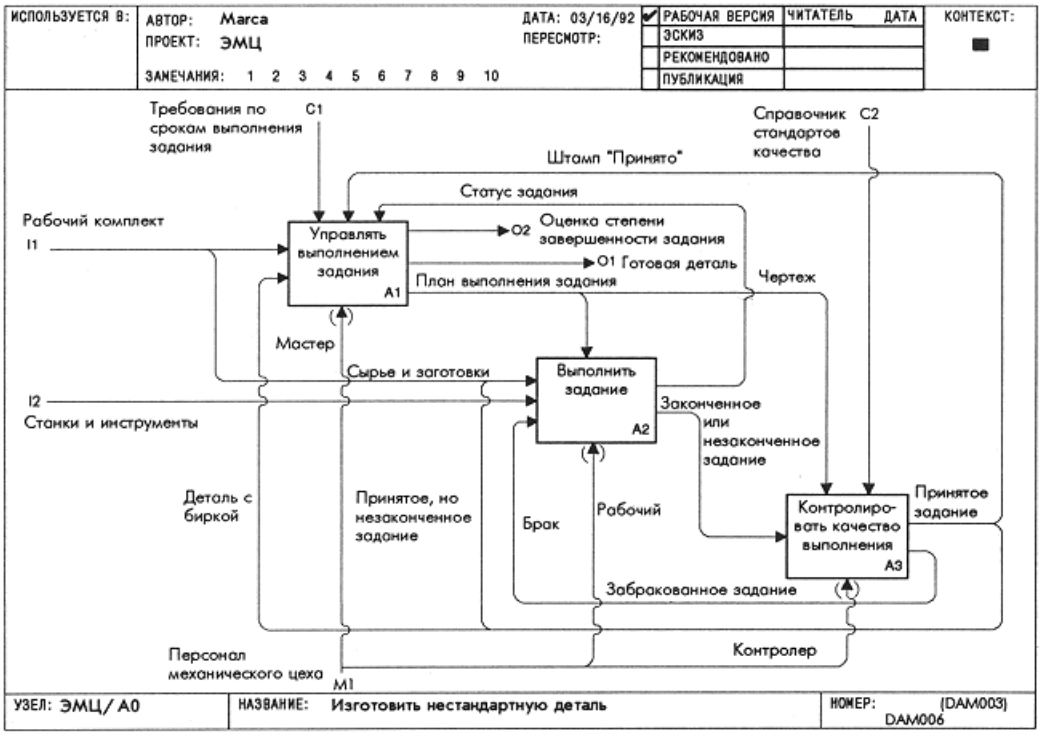


Рисунок 1.12 – Декомпозиция исходной диаграммы SADT-модели

Технология DFD. Графическая модель системы определяется как иерархия диаграмм потоков данных (ДПД), описывающих асинхронный процесс преобразования информации от ее ввода в систему до выдачи пользователю. Диаграммы верхних уровней иерархии (контекстные диаграммы) определяют основные процессы или подсистемы АИС с внешними входами и выходами. Они детализируются при помощи диаграмм нижнего уровня. Такая декомпозиция продолжается, создавая многоуровневую иерархию диаграмм, до тех пор, пока не будет достигнут такой уровень декомпозиции, на котором процессы становятся элементарными и детализировать их далее нецелесообразно.

Основными нотациями ДПД являются: внешние сущности; системы/подсистемы; процессы; накопители данных; потоки данных. Источники информации (внешние сущности) порождают информационные потоки (потоки данных), переносящие информацию к подсистемам или процессам. Те в свою очередь преобразуют информацию и порождают новые потоки, которые переносят информацию к другим процессам или подсистемам, накопителям данных или внешним сущностям

- потребителям информации.

Внешняя сущность представляет собой материальный предмет или физическое лицо, представляющее собой источник или приемник информации, например, как показано на рисунке 1.13, заказчики, поставщики, клиенты, склад.

Рисунок 1.13 – Внешняя сущность

Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что она находится за пределами границ анализируемой АИС.

На рисунке 1.14 изображена контекстная диаграмма **подсистемы** (системы). Номер подсистемы служит для ее идентификации. В поле имени вводится наименование подсистемы – предложения с подлежащим и соответствующими определениями и дополнениями.

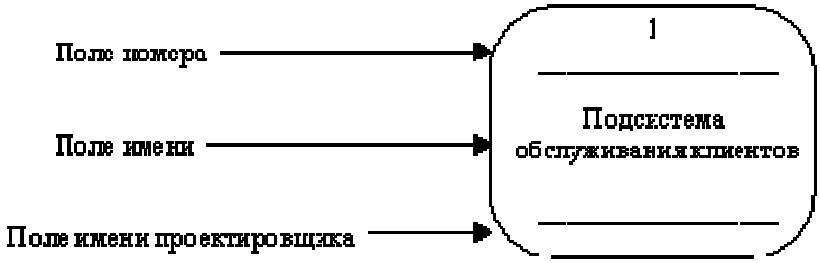


Рисунок 1.14 – Подсистема

Процесс представляет собой преобразование входных потоков данных в выходные на основе определенного алгоритма. Физически процесс может быть реализован различными способами: это может быть подразделение организации (отдел), выполняющее обработку входных документов и выпуск отчетов, программа, аппаратно реализованное логическое устройство и т.д. Процесс на диаграмме потоков данных изображается, как показано на рисунке 1.15.

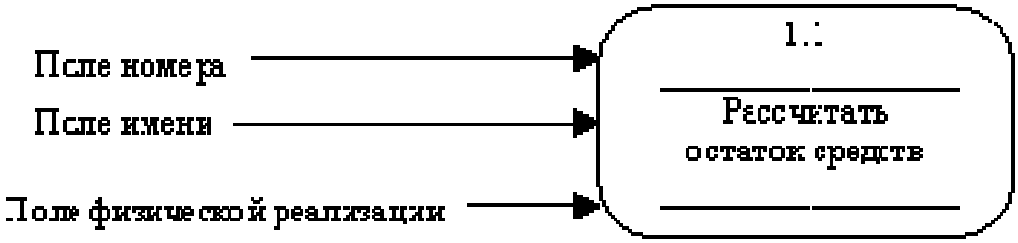


Рисунок 1.15 – Процесс

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с активным недвусмысленным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже. Информация в поле физической реализации показывает, какое подразделение организации, программа или аппаратное устройство выполняет данный процесс.

Накопитель данных представляет собой абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми. Накопитель данных может быть реализован физически в виде: таблицы в оперативной памяти, файла и т.д. Накопитель на диаграмме потоков данных изображается, как показано на рисунке 1.16

Рисунок 1.16 – Накопитель данных

Накопитель данных идентифицируется буквой "D" и произвольным числом. Накопитель данных в общем случае является прообразом будущей базы данных и описание хранящихся в нем данных должно быть увязано с информационной моделью.

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по кабелю между двумя устройствами, пересылаемыми по почте, переносимыми дискетами и т.д.

Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление потока (рисунок 1.17). Каждый поток данных имеет имя, отражающее его содержание. Средой, использующей DFD-модели, является BPwin, пример реализации которой показан на рисунке 1.17.

Рисунок 1.17 – Диаграмма потоков данных

Дальнейший рост сложности АИС потребовал разграничения доступа к глобальным данным программы. В результате технология структурного программирования получила развитие, отражением которого становится модульное программирования (70 гг. XX в.).

Технология модульного программирования предполагает выделение группы подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые модули (библиотеки подпрограмм).

Архитектура программы при технологии модульного программирования показана на рисунке 1.18.

Связи между модулями при использовании данной технологии осуществляются через специальный разрабатываемый интерфейс, в то время как доступ к реализации модуля (телу подпрограмм) запрещен. Использование модульного программирования существенно упростило разработку ПО несколькими

программистами. Кроме того, модули в дальнейшем без изменений можно было использовать в других проектах.

Эту технологию поддерживают современные версии высокоуровневых языков Turbo Pascal, C + и др.

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программные продукты, размер которых не превышает 100 000 операторов.

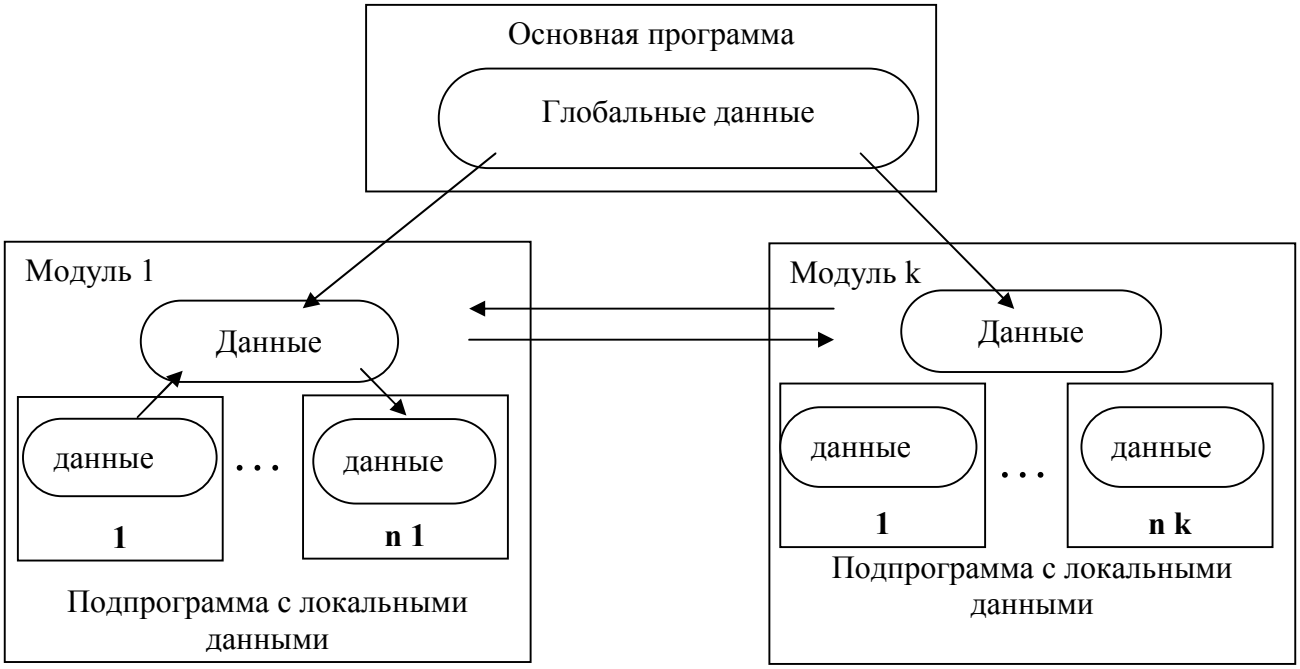


Рисунок 1.18 – Архитектура программы при технологии модульного программирования

Таким образом, узким местом технологии модульного программирования является то, что при увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и, с некоторого момента, предусмотреть взаимовлияние отдельных частей программы становиться практически невозможным.

1.3.2 Технологии на основе парадигмы объектно-ориентированного программирования

В 1980-90 гг. для проектирования ПО большого объема предложена к использованию технология объектно-ориентированная программирования (ООП). ООП определяется как технология, основанная на представлении программной архитектуры в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию объектов.

Такая технология требует переосмысления роли фундаментальных понятий прикладных информационных технологий – **модели и алгоритма** (рисунок 1.19).

Модель является базовым понятием для любых областей знаний, поскольку каждая попытка работать в точных терминах с реальным явлением должна начинаться с описания его формальной модели.

Именно модель представляет объект исследования и определяет характер формального аппарата, используемого для описания задачи и выполнения необходимых преобразований информации. Модель объекта вычислений определяет *ЧТО* надо вычислить, а алгоритм определяет *КАК* нужно вычислять. Простая истина - прежде, чем определить *КАК*, необходимо сформулировать *ЧТО* является объектом решения, т.е. построить модель, очевидна для всякой науки, использующей математику.

Рисунок 1.19 - Определения модели и алгоритма

Отсюда, особенностью последовательности технологических операций ООП,

изображенной на рисунке 1.19, является появление этапов моделирования и документирования, характерных для сложных программных проектов.

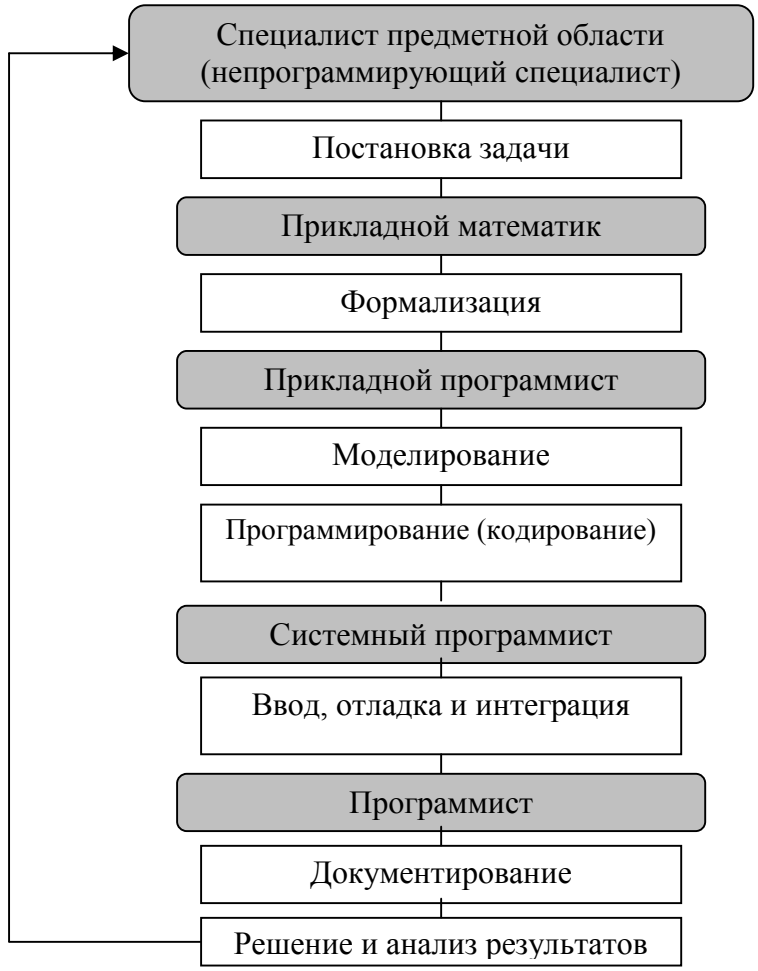


Рисунок 1.20 – Последовательность операций технологии ООП

Этап характеризуется появлением объектных языков программирования – Object Pascal, C++, в основе которых лежат следующие основные концепции:

– **класс** является описываемой на языке терминологии исходного кода моделью ещё не существующей сущности, т.е. объекта. Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области.

– **объект** является сущностью в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции исходного кода на выполнение).

Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений. Объект класса при этом обладает рядом характерных свойств (механизмов): абстрагирование, наследование, инкапсуляция, полиморфизм, существенно снижающая сложность проектирования ПО.

Абстрагирование – это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, абстракция – это набор таких характеристик.

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

В результате существенно увеличивается показатель повторяемости использования кода и появляется возможность создания библиотек классов для различных применений.

Другой характерной особенностью технологии ООП является архитектура программы, представленная на рисунке 1.20.

Реализацией технологии ООП в рамках спиральной модели ЖЦ является получившая в последнее время широкое распространение технология быстрой разработки приложений RAD (Rapid Application Development).

Основные принципы (концепции) технологии RAD:

- разработка приложений итерациями;
- необязательность полного завершения работ на каждом из этапов ЖЦ;
- обязательное вовлечение пользователей в процесс разработки АИС;
- необходимое применение CASE-средств, обеспечивающих целостность проекта;
- применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;

– использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности конечного пользователя;

– тестирование и развитие проекта одновременно с его разработкой;

– ведение разработки немногочисленной хорошо управляемой командой профессионалов;

– грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

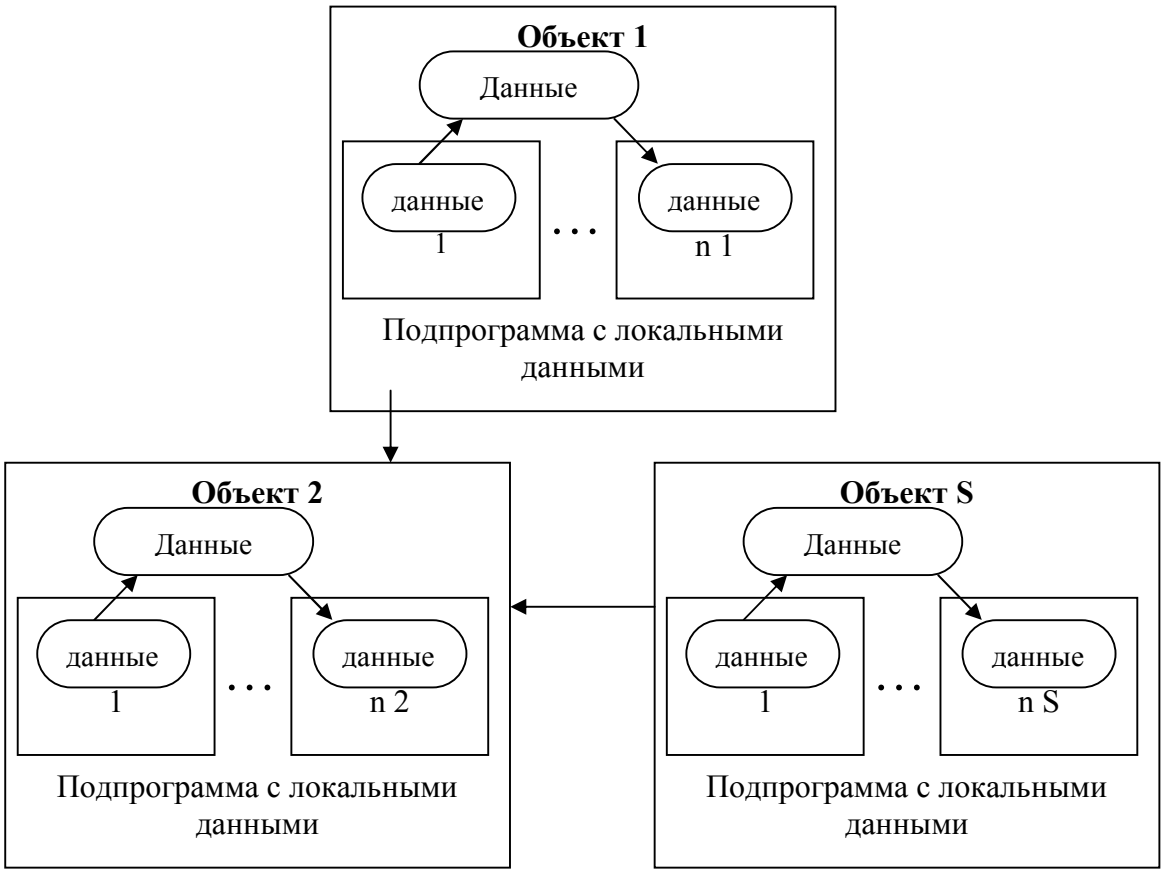


Рисунок 1.21 – Архитектура программы при технологии ООП

Процесс разработки программных систем по технологии RAD содержит следующие требования:

– небольшую команду программистов (от 2 до 10 человек);

– короткий производственный график (от 2 до 6 мес.);

– повторяющийся цикл, при котором разработчики, по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования,

полученные через взаимодействие с заказчиком.

Этапы спиральной модели ЖЦ программных систем, выполняемых в соответствии с технологией RAD, представлены на рисунке 1.22.

Рисунок 1.22 – ЖЦ АИС по технологии RAD

На этапе анализа и планирования пользователи системы определяют функции и требования АИС, выделяют наиболее приоритетные функции, описывают информационные потоки. Определение требований выполняется в основном силами пользователей под руководством специалистов-разработчиков. Ограничивается масштаб проекта, определяются временные рамки для каждого из последующих этапов. Результатом данного этапа являются техническое задание на разработку АИС.

На этапе проектирования пользователи принимают участие в техническом проектировании системы под руководством специалистов-разработчиков. CASE-средства используются для быстрого получения работающих прототипов приложений. Пользователи, непосредственно взаимодействуя с ними, уточняют и дополняют требования к системе. Более подробно рассматриваются процессы системы. Анализируется и, при необходимости, корректируется функциональная схема (модель). Каждая функция рассматривается детально. При необходимости для каждого элементарного процесса создается частичный прототип: экран, диалог, отчет, устраняющий неясности или неоднозначности. Определяются требования разграничения доступа к данным. На этом этапе формируется список необходимой документации.

После детального определения состава процессов оценивается количество функциональных элементов разрабатываемой системы и принимается решение о разделении АИС на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое для RAD-проектов время (60 - 90 дней). Проект распределяется между различными командами (делится функциональная модель).

Результаты этапа:

- общая информационная модель системы;
- функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;
- точно определенные интерфейсы между автономно разрабатываемыми подсистемами;
- прототипы экранов, отчетов, диалогов.

На этапе реализации выполняется непосредственно быстрая разработка приложения – разработчики производят итеративное построение реальной системы на основе полученных на предыдущем этапе моделей. Программный код частично формируется при помощи автоматических генераторов, получающих информацию непосредственно из репозитория CASE-средств. Конечные пользователи оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется непосредственно в процессе разработки.

После окончания работ каждой отдельной команды разработчиков производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование совместной работы данной части приложения с остальными, а затем тестирование системы в целом. Результатом этапа является готовая система, удовлетворяющая всем согласованным требованиям.

На этапе внедрения производится обучение пользователей, организационные изменения и опытная эксплуатация новой системы.

Оценка размера приложений производится на основе так называемых функциональных точек (экраны, сообщения, отчеты, файлы и т.п.) Подобная метрика не зависит от языка программирования, на котором ведется разработка. Размер приложений, разработанных по технологии RAD, для хорошо отлаженной среды разработки АИС с максимальным повторным использованием программных компонентов представлен в таблице 1.1

Таблица 1.1 – Характеристика приложений, реализуемых по технологии RAD

Количество Функциональных точек	Характеристика группы Разработчиков
< 1000	один человек
1000-4000	одна команда разработчиков
> 4000	4000 функциональных точек на одну команду разработчиков

Технология RAD, соответствующая парадигме ООП, наряду с неоспоримыми преимуществами, обладает рядом существенных недостатков:

– отсутствие стандартов компоновки двоичных результатов компиляции объектов в единое целое даже в рамках одного языка программирования;

– взаимодействия между объектами требует разработки интерфейса, а, следовательно, дополнительных затрат времени и возникновение возможности ошибки в коде;

– изменение реализации одного объекта требует перекомпиляции всего программного продукта.

Таким образом, технология RAD эффективна для программных проектов средней сложности под конкретного заказчика. Разработка сложных программных систем (операционные системы, системы реального масштаба времени), т.е. программы с большим процентом уникального кода, требуют более высокого уровня планирования и жесткой дисциплины проектирования.

Для преодоления указанных недостатков ООП получил развитие компонентно-ориентированная парадигма программирования.

1.3.3 Вопросы и задания для самоконтроля

1 Что послужило формированию нового дохода к программированию который был назван «структурным».

2 В чем заключается сущность структурного подхода?

3 Охарактеризуйте технологию SADT. Перечислите правила SADT.

4 Охарактеризуйте технологию DFD. Дайте определение внешней сущности.

5 В чем заключается технология модульного программирования? Поясните архитектуру при технологии модульного программирования.

6 Поясните архитектуру программы при объектно – ориентированной технологии.

7 Дайте определение понятиям *модель* и *алгоритм.*

8 Перечислите последовательность операций технологии ООП.

9 Перечислите этапы спиральной модели ЖЦ АИС по технологии RAD. Охарактеризуйте каждый этап ЖЦ.

10 Перечислите недостатки характерные технологии RAD.

1.4 Современные технологии разработки программного обеспечения

Разработка программного обеспечения является молодой и быстро развивающейся отраслью инженерной науки, которая подвержена постоянным и быстрым изменениям. Так, лишь в начале 90-х годов Британское сообщество вычислительной техники (British Computer Society) начало присваивать разработчикам программ квалификацию инженера (Chartered Engineer), а в Соединенных Штатах (в штате Техас) только в 1998 году стало возможным зарегистрироваться в качестве профессионального инженера программного обеспечения. Но по-прежнему, даже в начале 21-го века, ***общепризнанным остается тот факт, что разработке программного обеспечения не достает развитой научной базы.*** По некоторым оценкам, 75 % организаций программной индустрии занимаются разработкой программ на интуитивном уровне. С другой стороны, в этой области сформировалось немало интересных идей и знакомство с ними является содержанием настоящей лекции.

1.4.1 Технологии компонентно-ориентированного программирования

Технологии компонентно-ориентированного программирования (КОП) определяет **стандартный механизм**, с помощью которого одна часть ПО предоставляет свои услуги другой части. Организация предоставления услуг в библиотеках, приложениях, системном и сетевом программном обеспечении позволяет изменить технологию создания программ.

Наиболее известные технологии КОП представлены на рисунке 1.23.

COM (Component Object Model) – компонентная модель объектов	CORBA (Common Object Request Broker Architecture) - общая архитектура с посредником обработки запросов объектов

Рисунок 1.23 – Технологии компонентно-ориентированного программирования

Концепция технологии COM для семейства операционных систем Windows заключается в построении программ из компонент, которые состоят из объектов, представляющих собой непосредственно **исполняемый двоичный код**. Важнейший признак «компонентности» - исполняемую программу можно собирать из отдельных частей **без операций сборки** (модуля).

COM устанавливает **понятия и правила**, необходимые для определения **объектов и интерфейсов**; кроме того, в ее состав входят программы, реализующие ключевые функции. В COM любая часть ПО реализует свои услуги с помощью объектов COM. Каждый объект COM поддерживает несколько интерфейсов. Клиенты могут получить доступ к услугам объекта COM только через вызовы операций его интерфейсов — нет непосредственного доступа к телу объекта.

Отличие COM от привычных объектов в стиле ООП состоит в том, что объекты ООП известны только компилятору. Это абстракции, в которых мыслит программист и которые компилятор превращает в двоичные структуры «данные + код». Технология COM есть технология, которая переносит все преимущества ООП, доступные программисту на уровне исходного текста, на **двоичный уровень**. Если в исходном тексте технологии ООП программист волен использовать любые объекты, но теряет всяческий контроль над тем, что сделано, как только исходный текст скомпилирован, то при

использовании COM эти возможности сохраняются на протяжении всего жизненного цикла программы. Кроме того добавляются возможности разделения проекта на отдельные, **повторно используемые, двоичные компоненты.**

Хронология развития технологии COM показана на рисунке 1.24.

Рисунок 1.24 – Хронология развития технологии COM

Использованные сокращения: Dynamic Link Libraries (DLL) – динамически подключаемые библиотеки, Open DataBase Connectivity (ODBC) – открытый интерфейс доступа к базам данных, встроенный в Windows и Windows NT, Dynamic Data Exchange (DDE) – динамический обмен данными, Object Linking and Embedding (OLE1.0) – внедрение и связывание объектов, **OLE2.0 –** OLE **на базе COM**, Distributed COM (DCOM) – распределенная модель компонентных объектов, COM+ – новейшая технология COM.

Главная идея технологии **ODBC** заключается в создании **промежуточного программного слоя**, который определяет стандартный интерфейс для приложений. На уровне вызовов этот интерфейс использует язык SQL, а реализация взаимодействия с БД обеспечивается драйверами, поставляемые в форме DLL. Таким образом, ODBC располагается между приложением и источниками данных различных форматов.

Технологию динамического обмена данными **DDE** можно рассматривать как попытку **стандартизации обмена данными** между приложениями. Концепция Windows основана на обработке сообщений (messages). Отсюда приложения могут обмениваться друг с другом сообщениями, используя общую очередь сообщений. Проблема состоит в том, что каждое из приложений должно знать протокол обмена данными, т.е. **формат сообщений**. Технология DDE как раз и предложила такой **стандартный протокол**, реализованный во многих приложениях. Недостатками DDE является сложность программирования, невысокая надежность и то обстоятельство, что приложения должны знать формат передаваемых данных.

Технологию внедрения и связывания объектов **OLE1.0** фирма Microsoft представила в 1991 г. как попытку реализации **объектно-ориентированного механизма взаимодействия приложений**. Главной идеей OLE является **концепция составного документа**, который может содержать объекты других приложений. До OLE приложения могли обмениваться **статическими снимками данных** через буфер обмена Windows. Однако редактирование таких данных должно выполняться тем приложением, которое их породило, а после редактирования они вновь должны быть вставлены в другой документ. Если изменяются исходные данные, то, очевидно, должны изменяться данные и в составном документе. Однако, системный буфер обмена не имеет никаких средств поддержания таких связей. Более того, проблемы возникают и при перемещении исходных данных в новое место. Внедренный с помощью OLE1.0 объект содержит статические данные и данные, необходимые для его редактирования. Для редактирования объекта пользователю приложения-контейнера необходимо щелкнуть по объекту, вследствие чего в отдельном окне запускается исходное приложение, породившее эти данные. По окончании редактирования пользователь может сохранить данные, которые будут обновлены и в приложении-контейнере.

Недостатками технологии OLE1.0 являются:

– базовый механизм OLE1.0 – DDE по своей природе асинхронен, т.е. возврат управления при вызове любой функции происходит немедленно, но после завершения операции;

– для передачи данных между приложениями используется разделяемая глобальная память, т.е. данные сначала копируются в нее, а затем могут быть вытолкнуты Windows в **файл подкачки**, вследствие чего замедляется работа приложения;

– связи OLE1.0 легко разрываются при перемещении файлов;

– пользователю неудобно редактировать данные в отдельном окне.

Архитектура программных компонентов, разработанных по технологии COM, и взаимодействие COM-объектов показана на рисунке 1.25.

По технологии COM приложение представляет собой службы (функции) использующие специальные объекты – объекты COM, которые являются экземпляром

класса COM. Объект COM включает **поля и методы**, но может реализовать **несколько интерфейсов**, обеспечивающих доступ к его полям и функциям (достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса). При этом интерфейс объединяет несколько однотипных функций. Кроме того, классы COM поддерживают **наследование интерфейсов**, но не поддерживают наследование реализации, т.е. **не наследуют код методов**, хотя при необходимости объект класса-потомка может вызвать метод родителя.

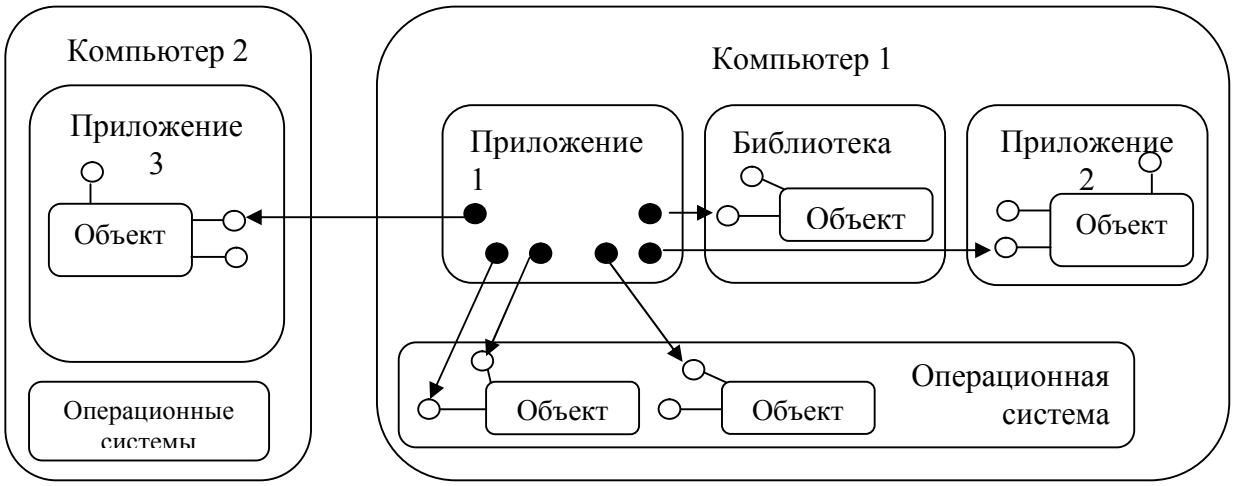


Рисунок 1.25 – Архитектура приложений на основе программных компонентов

Объекты COM всегда функционируют в составе сервера – динамической библиотеки и исполняемого файла. Различают три типа серверов, представленных на рисунке 1.26.

Внутренний сервер	**Локальный сервер**	**Удаленный сервер**
Реализуется динамическими библиотеками, которые подключаются к приложению - клиента и работают в одном адресном пространстве.	Реализуется отдельным процессом (модулем .exe) который работает на одном компьютере с клиентом.	Реализуется процессом, который работает на другом компьютере.

Рисунок 1.26 – Реализация технологии COM

Например, Microsoft Word является локальным сервером, включающим множество объектов, которые могут использоваться другими приложениями. Для обра-

щения к службам клиент должен получить указатель на соответствующий интерфейс. Перед первым обращением клиент посылает запрос к библиотеке COM, хранящей информацию обо всех зарегистрированных в системе классах COM объектов, и передает ей имя класса, идентификатор интерфейса и тип сервера. Библиотека запускает необходимый сервер, создает требуемые объекты и возвращает указатели на объекты и интерфейс. Получив указатели, клиент может вызывать необходимые функции объекта.

Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM. При использовании удаленных серверов в адресном пространстве клиента создается proxy-объект (заместитель объекта COM), а в адресном пространстве сервера COM – заглушка, соответствующая клиенту. Получив задание от клиента, заместитель упаковывает его параметры и, используя службы ОС, передает вызов заглушке. Заглушка распаковывает задание и передает его объекту COM. Результат возвращается объекту в обратном порядке.

Технология CORBA, разработанная группой компании OMG (группа внедрения объектной технологии программирования), реализует подход, аналогичный COM, но на базе объектов и интерфейсов CORBA. Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ и потому эту технологию можно использовать для создания распределенного ПО **в гетерогенной (разнородной) вычислительной среде**. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью специального посредника, названного VisiBroker.

Последовательность операций технологии КОП представлена на рисунке 1.27

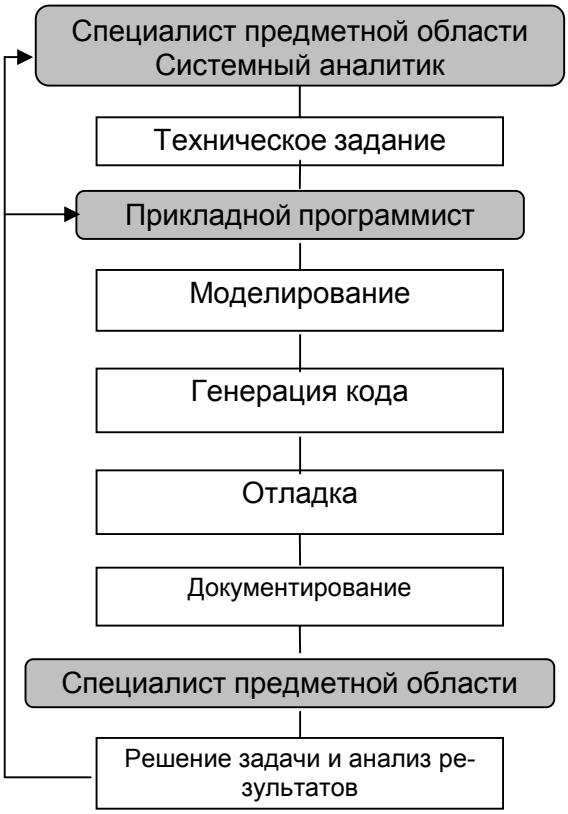


Рисунок 1.27 – Последовательность технологических операций КОП

Таким образом, технология КОП существенно упрощает разработку, позволяет сократить сроки проектирования за счет повышения повторяемости элементов программных системы и автоматизации отдельных этапов разработки, т.е. реально перейти к индустриальному методу разработки программного обеспечения АИС

1.4.2 Case-технологии проектирования программного обеспечения

Современные методологии и реализующие их технологии поставляются в электронном виде вместе с CASE-средствами и включают библиотеки процессов, шаблонов, методов, моделей и других компонент, предназначенных для построения ПО того класса систем, на который ориентирована методология. CASE-средства поддерживают внедрение автоматизированных технологий на всех этапах жизненного цикла ПО – **CASE-технологии** (Computer-Aided Software/System Engineering –

разработка программного обеспечения/программных систем с использованием компьютерной поддержки).

Существующие CASE-средства поддерживают как структурный, так и объектный подходы (в том числе и компонентный) к программированию и являются предметом изучения настоящей дисциплины.

На рисунке 1.28 представлены базовые CASE-средства современных технологий разработки ПО.

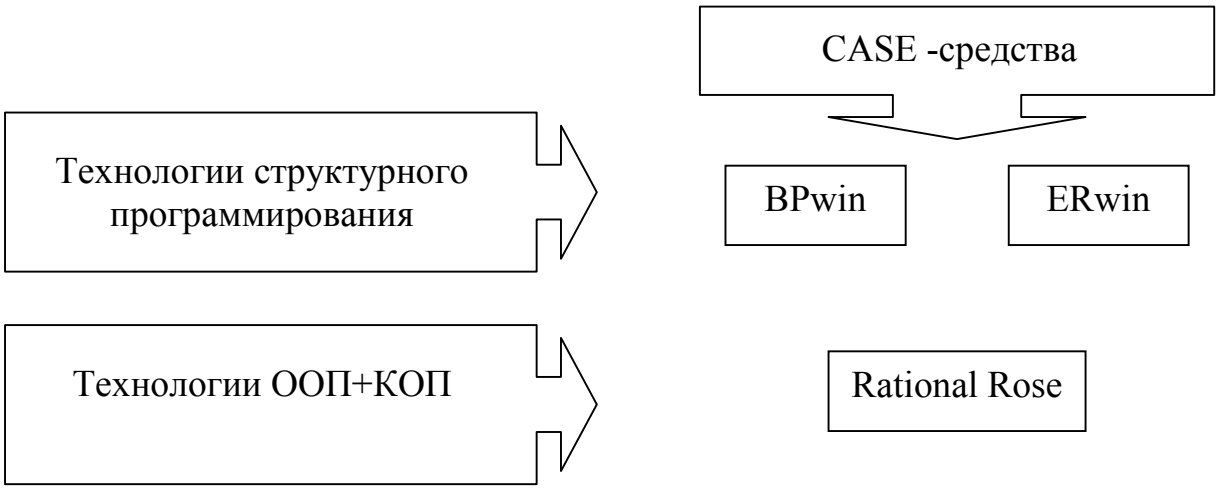


Рисунок 1.28 – Базовые CASE-средства современных технологий разработки ПО

BPwin – мощный инструмент моделирования программных систем, который используется для анализа, документирования и реорганизации сложных бизнес-процессов. Модель, созданная средствами BPwin, позволяет четко документировать различные аспекты деятельности – действия, которые необходимо предпринять, способы их осуществления, требующиеся для этого ресурсы и др. При этом формируется целостная картина деятельности предприятия – от моделей организации работы в отделах до сложных иерархических структур. Модели бизнес-процессов служат прекрасным средством документирования потребностей, помогая обеспечить высокую эффективность инвестиций в сферу IT. В руках же системных аналитиков и разработчиков BPwin – мощное средство моделирования процессов при создании корпоративных информационных систем (КИС).

BPwin совмещает в одном инструменте средства моделирования функций (IDEF0), потоков данных (DFD) и потоков работ (IDEF3), координируя эти три основных аспекта бизнеса для соответствия потребностям бизнес-аналитиков и системных аналитиков.

ERwin (AllFusion ERwin Data Modeler 4.1) предназначен для проектирования, внедрения и поддержки баз данных, хранилищ данных и моделей данных масштаба предприятия.

Интеграция с другими продуктами:

ModelMart – среда для совместной работы группы проектировщиков BPwin и/или ERwin над одним проектом. Позволяет управлять проектом моделирования, повышает скорость и эффективность работы.

ADvantage – линейка продуктов для поддержки всех стадий разработки программного обеспечения (аналог Rational Suite) В ADvantage, в частности, входит линейка CASE-средств ERwin Modeling Suite (ERwin, BPwin, ModelMart, Paradigm Plus, ERwin Examiner) и средства управления проектами. Совместное применение этих продуктов обеспечивает прочный фундамент для построения, развертывания и управления приложениями.

Paradigm Plus – средство проектирования компонентов ПО и кодогенерации. Двусторонняя связь между продуктами позволяет импортировать смоделированные бизнес-процессы в Paradigm Plus как use cases и экспортировать их в BPwin.

Model Navigator – продукт для просмотра моделей ERwin и BPwin с возможностью генерации отчетов.

Arena – система имитационного моделирования. Интеграция позволяет использовать готовые модели для изучения изменяющегося во времени взаимодействия бизнес-процессов.

Rational Rose – средство визуального моделирования объектно-ориентированных информационных систем компании Rational Software Corp. Работа продукта основана на универсальном языке моделирования UML

(Universal Modeling Language). Благодаря уникальному языку моделирования Rational Rose способен решать практически любые задачи в проектировании информационных систем: от анализа бизнес процессов до кодогенерации на определенном языке программирования. Только Rose позволяет разрабатывать как высокоуровневые, так и низкоуровневые модели, осуществляя тем самым либо абстрактное проектирование, либо логическое, кроме того осуществляет такие подходы, как прямое и обратное проектирование,

Известны следующие продукты компании:

Rational Software Architect — средство моделирования (дальнейшее развитие Rational Rose на платформе Eclipse).

Rational PurifyPlus — набор программ для вылавливания утечек памяти, анализа области перекрытия кода и производительности кода.

Rational ClearCase — система управления версиями.

Rational RequisitePro — система управления требованиями.

Rational ClearQuest — система управления изменениями.

SoDA — система автоматизированного документирования.

Rational Robot и Rational Functional Tester — средство автоматизированного тестирования

Rational Performance Tester — средство автоматизированного нагрузочного тестирования.

Rational Process Advisor — инструмент интеграции процесса разработки ПО при помощи инструментов разработки и тестирования.

Использование указанных средств автоматизации разработки ПО позволяет перейти к последовательности технологических операций, представленных на рисунке 1.29.

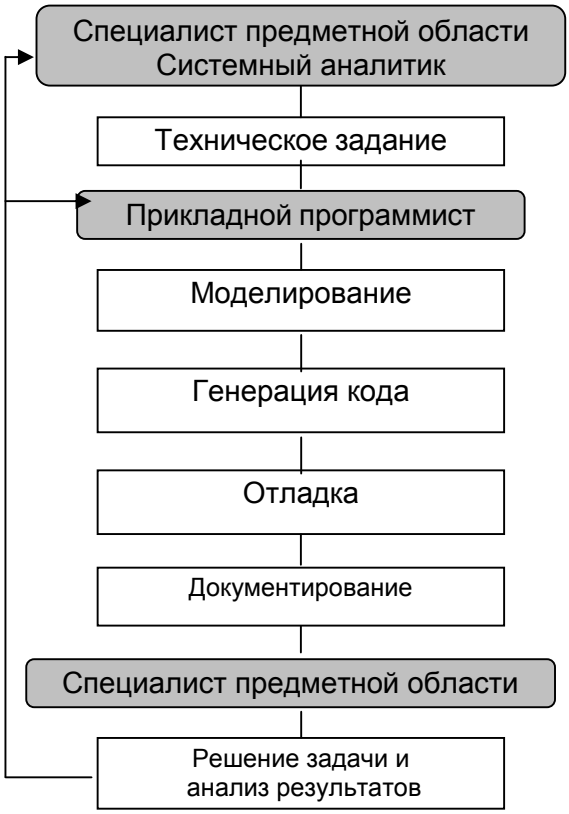


Рисунок 1.29 – Последовательность технологических операций на основе CASE-средств

Таким образом, CASE-технологи позволяют автоматизировать большинство технологических операций, т.е. реально перейти к индустриальному методу разработки программного обеспечения АИС.

1.4.3 Вопросы и задания для самоконтроля

1 В чем заключается технология компонентно-ориетированного программирования.

2 В чем заключается концепция технологии COM.

3 В чем заключается основная идея технологии ODBC?

4 Поясните сущность технологии CORBA.

5 Перечислите последовательность технологических операций технологий КОП.

6 Перечислите базовые Case – средства современных технологий разработки ПО. Дайте основные характеристики функциональным возможностям Case – средства BP Win.

7 Дайте основные характеристики средству визуального моделирования объектно ориентированных систем.

8 Перечислите последовательность технологических операций на основе Case-средств.

9 Дайте основные характеристики функциональным возможностям Case – средства Rational Rose.

10 Поясните архитектуру приложений на основе программных компонентов.

2 Автоматизация разработки программного обеспечения на основе UML

2.1 Спецификация программного обеспечения при использовании UML

В основе индустриального подхода к программной инженерии лежит фундаментальная идея: ***проектирование ПО является формальным процессом, который можно изучать и совершенствовать.***

В настоящее время фактически стандартным средством описания проектов ПО, создаваемого с использованием объектно-ориентированного подхода, признан унифицированный язык моделирования – UML (Unified Modeling Language), первой версии которого появилась в I995 г. Его создателями являются ведущие специалисты в области программной инженерии: Гради Буч, Ивар Якобсон и Джеймс Рамбо, которые использовали в этом языке все лучшее, что появилось в подходах этих авторов во время «войны методов».

2.1.1 Основы модельного языка описания программного обеспечения

В основе стандартного языка описания технологии проектирования ПО лежит унифицированный язык моделирования UML, использующий объектную декомпозицию, т. е. представление разрабатываемого ПО в виде совокупности объектов, в процессе взаимодействия которых через передачу сообщений и происходит выполнение требуемых функций.

Основные принципы построения моделей в UML:

– абстрагирование (предписывает включать в модель только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций или целевого назначения;

– многомодельность (утверждение, что никакая единственная модель не может с достаточной степенью адекватности описать различные аспекты сложной системы);

– иерархичность (предполагает рассматривать процесс построения модели на разных уровнях абстрагирования или детализации в рамках фиксированных представлений).

Спецификация разрабатываемого ПО при использовании UML объединяет несколько моделей, представленных на рисунке 2.1.

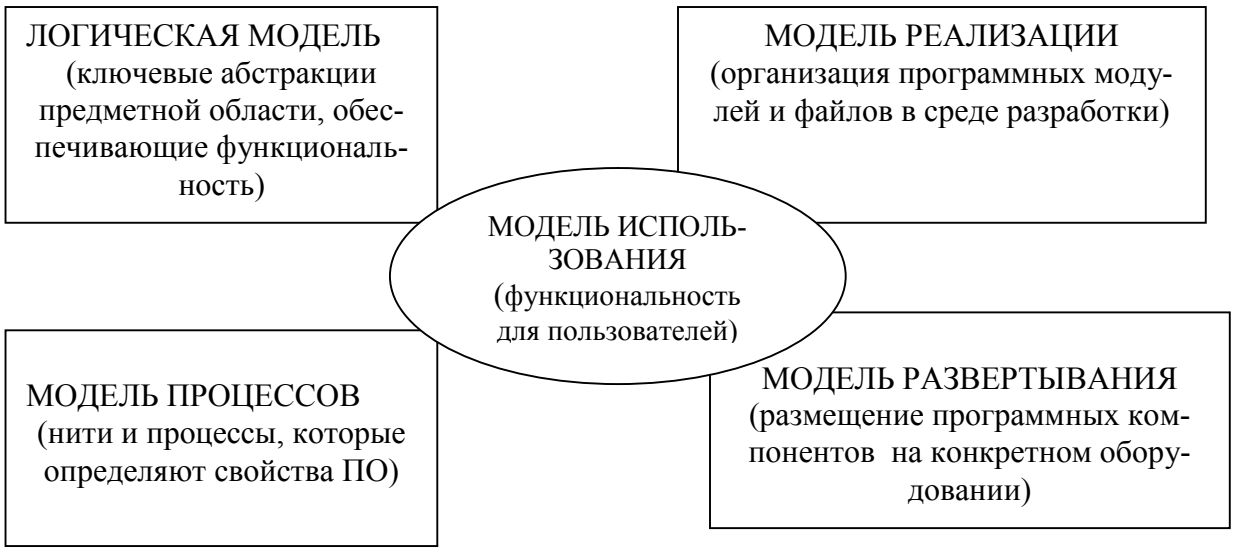


Рисунок 2.1 – Полная спецификация разрабатываемого ПО при использовании UML

Модель использования представляет собой описание функциональности ПО с точки зрения пользователя.

Логическая модель описывает ключевые абстракции предметной области ПО (классы, интерфейсы и т.д.), т.е. средства, обеспечивающие требуемую функциональность.

Модель реализации определяет организацию программных модулей в среде разработки.

Модель процессов отображает организацию вычислений и оперирует понятиями «процессы» и «нити», которые позволяют оценить производительность, масштабируемость и надежность программного обеспечения.

Модель развертывания показывает особенности размещения программных компонентов на конкретном оборудовании.

Таким образом, каждая из указанных моделей характеризует определенный аспект проектируемой системы, а все они вместе составляют относительно полную модель разрабатываемого программного обеспечения.

Для реализации каждой из указанных моделей в UML используется девять дополняющих друг друга диаграмм (могут входить в различные модели):

– **диаграммы вариантов использования** (use case diagrams) – для моделирования бизнес-процессов организации и требований к создаваемой системе;

– **диаграммы классов** (class diagrams) – для моделирования статической структуры системы;

– диаграммы поведения системы:

–**диаграммы последовательности** (sequence diagrams) и **диаграммы кооперации** (collaboration diagrams) – для моделирования процесса обмена сообщениями между объектами;

–**диаграммы деятельностей** (activity diagrams) – для моделирования поведения системы в рамках различных вариантов использования;

–**диаграммы состояний** (state chard diagrams) – для моделирования поведения объектов системы при переходе из одного состояния в другое;

– диаграммы реализации:

–**диаграммы компонентов** (component diagrams) – для моделирования иерархии компонентов (подсистем);

–**диаграммы размещения** (deployment diagrams) – для моделирования физической архитектуры системы.

Помимо указанных диаграмм, как и при структурном подходе, спецификация обязательно включает словарь терминов, а также различного рода описания и тек-

стовые спецификации. Конкретный набор документации определяется разработчиком. До настоящего времени не существует единой, устоявшейся терминологии в области объектно-ориентированного проектирования. В таблице 2.1 приведены соответствия между основными терминами, используемыми наиболее известными авторами в этой области.

Таблица 2.1- Соответствия между терминами и соответствующими нотациями

Нотации	Термины			
UML	Класс	Ассоциация	Обобщение	Агрегация
Буч	Класс	Использование	Наследование	Включение
Якобсон	Объект родства	Ассоциация	Наследование	Состоит из
Одел	Тип объекта	Связь	Подтип	Композиция
Рамбо	Класс	Ассоциация	Обобщение	Агрегация

UML и предлагаемая методика Rational Unified Process поддерживаются пакетом **Rational Rose** фирмы Rational Software Corporation. Ряд диаграмм UML можно построить также средствами программы Microsoft Visual Modeler и других CASE-средств.

По данным «USA Today» в настоящее время 49 из 50-ти ведущих компьютерных компаний используют UML при проектировании ПО с использованием объектного подхода, что позволяет говорить о том, что сегодня UML фактически стал стандартом описания сложных программных систем.

2.1.2 Спецификация разрабатываемого программного обеспечения на этапе постановки задачи

Постановка задачи на разработку ПО формулируется в виде технического задания (ТЗ). В качестве примера формирования ТЗ рассматривается постановка

задачи на разработку автоматизированной информационной системы управления запасами товара заданного наименования на складе.

Предметная область. Отдел маркетинга предприятия занимается реализацией товара. Предприятие имеет несколько складов, в которых находится товар для реализации. Товар завозят на склад в соответствии с приходной накладной, которая имеет дату, перечень (наименование) товара, количество единиц каждого товара. Также указывается ФИО и должность сотрудника склада, принявшего его. Расход товара со склада осуществляется по расходной накладной, которая имеет ту же структуру, что и приходная, только учитывает расход товара со склада. Поступление товара на склад отражается в карточке складского учета, заводимой для каждого наименования товара. В карточке учитываются все приходы и расходы. В накладной также указывается отпускная цена на текущую дату, количество отпущенного товара.

Аналитическое приложение должно обеспечивать автоматизацию информационных процессов менеджмента отдела маркетинга – прогнозирование запаса товара на складе торгового предприятия**.**

На основании описания предметной области формулируется техническое задание.

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

на разработку автоматизированной информационной системы менеджера торгового предприятия

Введение

Настоящее техническое задание распространяется на разработку приложения, предназначенного для автоматизации информационных процессов менеджмента торгового предприятия**.**

Широкий круг задач управления бизнесом относится к классу задач теории статистических решений, методы которых, как правило, имеют экспоненциальную вычислительную сложность и реализованы в дорогостоящих универсальных лицензионных программных пакетах, сложно интегрируемых с автоматизированными системами управления бизнес – процессами.

Поэтому создание системы, в рамках которой были бы реализованы наиболее часто используемые методы и алгоритмы теории статистических решений, является **необходимым** условием повышения эффективности бизнес-процессов и, кроме того, позволит как оценивать и исследовать отдельные методы и алгоритмы, так и сравнивать их с точки зрения затрат вычислительных ресурсов и точности получаемых решений.

Анализ известных аналогов средств автоматизации бизнес-процессов свидетельствует об **актуальности** разработки программной системы, интегрируемой с существующими базами данных корпоративных информационных систем.

Менеджеры предприятия работают с потоками данных о товаре, что требует большого количества времени и трудозатрат, чтобы вести учет прихода и расхода товара, а также спланировать будущее поступление товара определенного наименования на склад.

1 Основание для разработки

Система разрабатывается на основании учебного плана направления 230100.65 – Информатика и вычислительная техника по дисциплине «Системы автоматизации проектирования программного обеспечения» студентов набора 2008 г.

2 Назначение

Первая версия системы предназначена для автоматизации информационного процесса прогнозирования количества товара заданного наименования на складе предприятия. В следующих версиях предполагается увеличение количества решаемых задач.

Пользователями данного средства являются менеджеры организации. Кроме того, программное средство могут использовать специалисты других предметных областей, которым приходится решать подобные бизнес-задачи.

3 Требования к программе или программному изделию

3.1Требования к функциональным характеристикам

3.1.1 Система должна представлять совокупность методических и программных средств решения следующих задач:

– учет поступления товара на склад предприятия;

– учет расхода товара на складе;

– выборка необходимых данных из базы данных;

– прогнозирование закупки товара определенного наименования.

3.1.2 Для реализации этих задач должен быть реализован метод корреляционного и регрессионного анализа.

3.1.3 Методическое обеспечение должно быть реализовано в пользовательском интерфейсе системы, который должен предполагать выбор необходимой задачи; ввод данных и сохранение исходных данных, промежуточных и окончательных результатов в базе данных для последующего анализа.

3.2 Требования к надежности

3.2.1 Предусмотреть контроль вводимой информации и блокировку некорректных действий пользователя при работе с системой.

3.2.2 Обеспечить корректное завершение вычислений с соответствующей диагностикой при превышении имеющихся вычислительных ресурсов.

3.2.3 Возможность автоматизации создания отчетной документации по результатам обработки информации.

3.2.4 Обеспечить целостность информации, хранящейся в базе данных.

3.3 Требования к составу и параметрам технических средств.

3.3.1 Система должна работать на IBM совместимых персональных компьютерах.

3.3.2 Минимальная конфигурация:

Тип процессора……Pentium-IV;

Место, занимаемое на диске……...106,63 Мб;

Наличие локальной сети для связи с сервером, мышь, клавиатура.

3.4 Требования к информационной и программной совместимости

Система должна работать под управлением операционной системы Windows'95 и выше.

4 Требования к программной документации

4.1 Система должна включать справочную информацию о разработчике, последовательность работ и подсказки пользователю.

4.2 В состав сопровождающей документации должны входить:

- руководство системного программиста;
- руководство оператора – менеджера торгового отдела.

5 Этапы разработки

№	Название этапа	Срок	Работы
1	Разработка ядра системы	15.02.2010-01.03.2010	Описание внутренних форматов, интерфейса и форматов данных базы. Реализация системы на уровне интерфейса.
2	Разработка методов, алгоритмов и их реализация для поставленной задачи	01.03.2010-19.03.2010	Описание методов и алгоритмов. Программные модули, реализующие методы.
3	Тестирование программной системы и разработка документации	19.03.2010-01.04.2010	Тесты. Документация. Программная система.

После утверждения ТЗ организация-разработчик непосредственно приступает к созданию ПО. Однако переход к следующему этапу разработки – этапу уточнения спецификаций требует принятия еще некоторых принципиальных решений, от которых во многом зависят как характеристики и возможности разрабатываемого ПО, так и особенности его разработки, начиная с выбора моделей этапа уточнения спецификаций

Разработку спецификаций ПО начинают с **анализа** требований к функциональности, указанных в техническом задании.

В процессе анализа уточняются внешние пользователи разрабатываемого ПО и перечень отдельных аспектов его поведения в процессе взаимодействия с конкретными пользователями. Аспекты поведения ПО называются «**вариантами использования**» или «**прецедентами**» (use cases).

Вариант использования представляет собой характерную процедуру применения разрабатываемой системы конкретным действующим лицом, в качестве которого могут выступать не только люди, но и другие программные системы или устройства.

Не следует путать вариант использования с конкретными операциями будущей системы. Каждый вариант использования связан с некоторой целью, имеющей самостоятельное значение, например, для текстового редактора *Формирование оглавления* – это вариант использования, а *Связывание заголовков* со специальными стилями – операция, которую необходимо выполнить, чтобы стало возможно автоматическое построение оглавления.

В зависимости от цели выполнения конкретной процедуры различают следующие варианты использования:

– основные (обеспечивают требуемую функциональность разрабатываемого ПО):

– вспомогательные (обеспечивают выполнение необходимых настроек системы и ее обслуживание, например, архивирование информации);

– дополнительные (обеспечивают удобства для пользователя, как правило, реализуются в том случае, если не требуют серьезных затрат каких-либо ресурсов при разработке или эксплуатации).

Вариант использования можно описать кратко или подробно.

Краткая форма описания содержит: название варианта использования, его цель, действующих лиц, тип варианта использования (основная, второстепенная или дополнительная) и его краткое описание. Краткое описание варианта использования *Прогнозирование запаса товара на складе* можно представить в виде таблицы 2.2.

Таблица 2.2 – Краткое описание (спецификация) варианта использования *Прогнозирование запаса товара на складе*

Название варианта	*Прогнозирование запаса товара на складе*
Цель	Получение результатов прогнозирования
Действующие лица	Пользователь (менеджер)
Краткое описание	Решение задачи предполагает выбор задачи прогнозирования, формирования необходимых данных и получение результатов прогноза.
Тип варианта	Основной

Основные варианты использования обычно описывают подробно, стараясь отразить особенности предметной области разрабатываемого ПО. Подробная форма, кроме указанной выше информации, включает описание типичного хода событий и возможных альтернатив.

Типичный ход событий представляют в виде диалога между пользователями и системой, последовательно нумеруя события. Если пользователь может выбирать варианты, то их описывают в отдельных таблицах. Также отдельно приводят альтернативы, связанные с нарушением типичного хода событий.

В таблицах 2.3, 2.4, 2.5 представлено подробное описание варианта использования *Инициализация процесса прогнозирования запаса товара на складе* в условиях типичного хода событий.

Альтернативы:

9 Если время выполнения программы с точки зрения пользователя велико, то он прерывает процесс выполнения.

10 Система прерывает расчеты, предлагает список алгоритмов решении и возвращается на шаг 7.

Дополнительная информация

1 Необходимо обеспечить произвольную последовательность выбора типа задачи, данных и алгоритма.

2 Необходимо обеспечить возможность выхода из варианта на любом этапе.

Таблица 2.3 – Вариант использования *Прогнозирование запаса товара*

Действия исполнителя	Отклик системы
1 Пользователь инициирует процесс прогнозирования запасов 3 Пользователь выбирает способ задания данных: а) Если выбран способ формирования пользователем данных из базы (см. *Ввод данных*). б) Если выбран ввод из базы данных (см. *Выбор данных из базы*).	2 Система регистрирует задание и предлагает список способов задания данных 4 Система регистрирует данные и предлагает список алгоритмов решения.
5 Пользователь выбирает алгоритм 7Пользователь инициирует процесс прогнозирования 9 Пользователь ожидает 11 Пользователь анализирует результаты и выбирает, сохранять их в базе или нет	6 Система регистрирует алгоритм и предлагает начать решение. 8 Система проверяет полноту определения задания и запускает модуль прогнозирования. 10 Система демонстрирует пользователю результаты и предлагает сохранить их в базе данных. 12 Если выбрано сохранение данных, то система выполняет запись данных прогноза в базу. 13 Система переходит в состояние ожидания.

Таблица 2.4 – Вариант использования *Ввод данных*

Действия исполнителя.	Отклик системы
1 Пользователь выбрал Ввод данных	2 Система последовательно запрашивает ввод данных
3 Пользователь вводит данные	4 Система проверяет данные и запрашивает, сохранять ли данные в базе
5 Пользователь отвечает на запрос	6 Если выбран вариант сохранения данных, то система выполняет запись данных в базу и регистрирует их в текущем задании

Альтернатива

4 Если обнаружены некорректные данные, то система выдает сообщение об ошибке и предлагает их исправить, возвращаясь на предыдущий шаг.

Таблица 2.5 – Вариант использования *Выбор данных из базы*

Действия исполнителя	Отклик системы
1 Пользователь выбрал *Выбор данных* из базы	2 Система демонстрирует список в базе.
3 Пользователь выбирает данные.	4 Система читает данные и регистрирует их в текущем задании

Таким образом, спецификацию разрабатываемого ПО на этапе постановки задачи можно считать завершенной и перейти к построению первой модели будущей системы – диаграммы вариантов использования.

2.1.3 Вопросы и задания для самоконтроля

1 Какие модели включает спецификация разрабатываемого ПО при использовании UML?

2 Какая модель описывает ключевые абстракции ПО?

3 Какие диаграммы используются для построения моделей в UML?

4 Перечислите диаграммы реализации системы. Каково назначение каждой из них?

5 Каким образом происходит разработка спецификаций ПО?

6 Дайте определение варианта использования.

7 Какой тип диаграмм позволяет наглядно представить ожидаемое поведение системы?

8 Какие различают варианты использования

9 Какие существуют формы описания варианта использования? Какие разделы включает краткая форма описания варианта использования?

10 Какие разделы включает полное описание варианта использования?

2.2 Спецификация программного обеспечения при использовании UML на этапе анализа

Модели разрабатываемого ПО при объектном подходе основаны на предметах и явлениях реального мира. В основе этих моделей лежит описание требуемого поведения разрабатываемого ПО, т. е. его функциональности, но это поведение связывается с состояниями элементов (объектов) конкретной предметной области.

Отсюда, на этапе анализа ставятся две задачи:

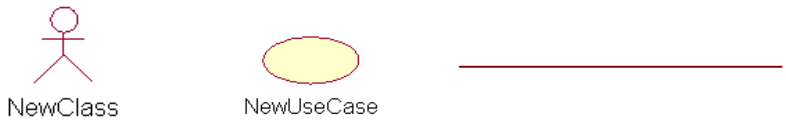
- уточнить требуемое поведение ПО (функциональность);
- разработать концептуальную модель предметной области с точки зрения поставленных целей.

2.2.1 Методика построение диаграммы вариантов использования

Первая задача решается путем построение диаграммы вариантов использования, которая отражает цели программной системы.

Диаграммы вариантов использования (ДВИ) – визуальная модель, позволяющая наглядно представить ожидаемое поведение системы.

Основными нотациями ДВИ (см. рисунок 2.2) являются: действующее лицо (а), вариант использования (б), связь (в).



а б в

а – действующее лицо; б – вариант использования; в – связь

Рисунок 2.2 – Основные нотации ДВИ

Действующее лицо – внешняя по отношению к разрабатываемому ПО сущность, которая взаимодействует с ним с целью получения или предоставления какой-либо информации. Действующими лицами могут быть пользователи, другое ПО или какие-либо технические средства, взаимодействующие с разрабатываемой системой.

Вариант использования – некоторая, очевидная для действующего лица процедура, решающая конкретную функцию. Все ДВИ, так или иначе, связаны с требованиями к функциональности разрабатываемой системы и могут отличаться по объему выполняемой работы.

Связь – взаимодействие действующих лиц и соответствующих вариантов использования.

Варианты использования также могут быть связаны между собой. При этом фиксируются связи использования и расширения.

Использование подразумевает, что существует некоторый фрагмент поведения разрабатываемого ПО, который повторяется в нескольких вариантах использования. Этот фрагмент оформляют, как отдельный вариант использования и указывают связь с ним типа «использование».

Расширение применяют, если имеется два подобных варианта использования, различающиеся наличием в одном из них некоторых дополнительных действий. В

этом случае дополнительные действия определяют как отдельный вариант использования, который связан с основным вариантом связью типа «расширение».

Пример. Построить ДВИ для автоматизированной информационной системы прогнозирования товара заданного наименования на складе.

Действующее лицо данной системы – *Менеджер торговой организации,* отвечающий за своевременное пополнение товара.

Диаграмма вариантов использования показывающая взаимодействие менеджера с разрабатываемым приложением представлена на рисунке 2.3.

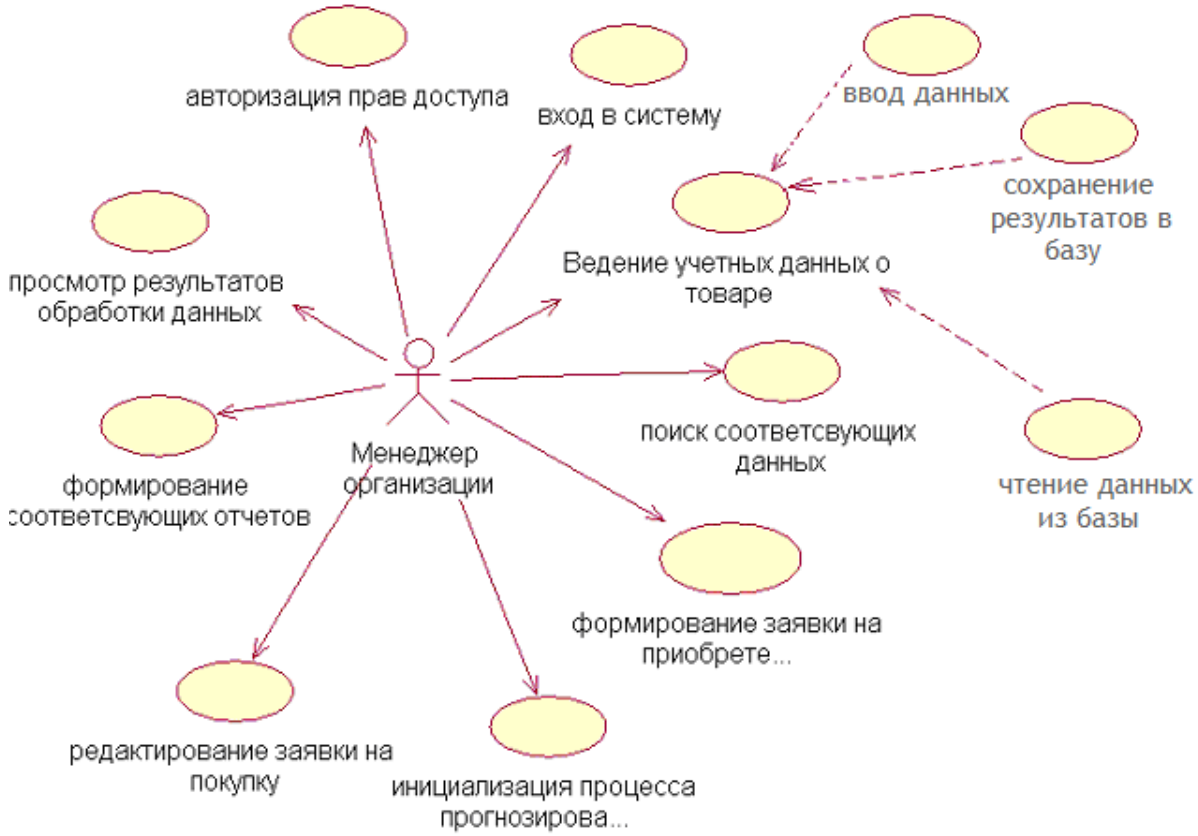


Рисунок 2.3 – Диаграмма вариантов использования взаимодействия пользователя с приложением

Помимо основных вариантов использования, система должна также предусматривать вспомогательные процедуры для удаления лишних данных и результатов прогноза из базы.

Полученная диаграмма вариантов использования отражает типичное взаимодействие пользователя с разрабатываемым ПО. Ее необходимо обсудить с заказчиком для определения как можно большего числа основных вариантов использования и

проанализировать на полноту обслуживания системы.

Естественно, все варианты использования определить, как правило, не удастся - новые варианты фиксируют постоянно, даже в процессе эксплуатации. Но, чем больше вариантов выявлено в процессе уточнения спецификаций, тем лучше, так как при этом получают более точную модель предметной области, что уменьшает вероятность ее пересмотра при добавлении функций.

Таким образом, ДВИ отражает функциональность будущего программного продукта и позволяет наглядно представить ожидаемое поведение системы.

2.2.2 Методика построение концептуальной модели предметной области

Диаграмма классов – центральное звено объектно-ориентированных методов проектирования ПО. В отличие от ранее существовавших методов, UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

– концептуальный уровень, на котором диаграммы классов, называемые в этом случае контекстными, демонстрируют связи между основными понятиями предметной области;

– уровень спецификаций, на котором диаграммы классов отображают интерфейсы классов предметной области, т. е. связи объектов этих классов;

– уровень реализации, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

Практически это три разных модели, связь между которыми неоднозначна. Так, если концептуальная модель определяет некоторое понятие предметной области как класс, то это не означает, что для реализации этого понятия будет использован отдельный класс. Однако во всех трех моделях представляют интерес типы объектов (классы) и их статические отношения, что позволяет использовать единые нотации.

Каждая из моделей используется на конкретном этапе разработки ПО:

концептуальная модель – на этапе анализа;

диаграммы классов уровня спецификации – на этапе проектирования;

диаграммы классов уровня реализации – на этапе реализации.

Концептуальная модель – визуальная модель, оперирующая понятиями предметной области, атрибутами этих понятий и отношениями меду ними. Понятию в предметной области разрабатываемого ПО могут соответствовать как материальные предметы, так и абстракции, которые применяют специалисты предметной области.

Основным понятиям в концептуальной модели ставятся в соответствие классы. Под **классом** при этом традиционно понимают совокупность общих признаков заданной группы объектов предметной области. В соответствии с этим определением на диаграмме классов каждому классу соответствует группа объектов, общие признаки которых фиксирует класс.

На диаграммах класс изображается в виде прямоугольника, внутри которого указано имя класса (рисунок 2.4, б). При необходимости допускается указывать характеристики класса, например, атрибуты, используя специальные секции условного обозначения (рисунок 2.4, а).

а – с уточнением атрибутов; б – без уточнения.

Рисунок 2.4 – Обозначение класса на концептуальной диаграмме классов

В качестве **атрибутов** представляют некоторые, существенные с точки зрения решаемой задачи, характеристики объектов, например идентифицирующие значения (имя, номер). Для конкретного объекта атрибут всегда имеет определенное значение. На диаграмме классов атрибуты обычно показывают в секции атрибутов.

Под отношением классов понимают статическую, т. е. не зависящую от времени, связь между классами. Различают два основных вида отношений: ассоциация и обобщение.

Отношение ассоциации означает наличие связи между экземплярами классов или объектами.

Связь между экземплярами классов подразумевает некоторые **роли**, которые соответствующие объекты играют по отношению друг к другу. Роль связана с направлением ассоциации. Если роль собственного имени не имеет, то можно считать, что ее имя совпадает с именем класса, по отношению к которому определяется эта роль.

Роль обладает характеристикой множественности, которая показывает, сколько объектов может участвовать в одной связи с каждой стороны, допускается указывать множественность:

* – от 0 до бесконечности;

<целое>.. * – от заданного числа до бесконечности;

<целое> – точно определенное количество объектов;

<целое1>, <целое2> – несколько вариантов точного количества объектов;

<целое1>..<целое2> – диапазон объектов.

С теоретической точки зрения атрибут тоже класс, экземпляры которого жестко ассоциированы с рассматриваемым классом. В концептуальной модели для отображения соответствующих отношений могут использоваться как ассоциации, так и обобщения.

Чтобы избежать излишних нагромождений рекомендуется следовать правилу: *если некоторый объект X в реальном мире не является числом или текстом, то это скорее всего понятие. В противном случае это атрибут.*

Обобщением называют такое отношение между классами, при котором любой объект одного класса (подтипа) обязательно является также объектом другого класса, называемого в данном контексте супертипом.

Следовательно, все, что известно об объектах супертипа (ассоциации, атрибуты, операции), касается и объектов подтипа. На диаграмме классов обобщение обозначают линией с треугольной стрелкой на конце, подходящей к супертипу.

На практике определение основных понятий предметной области, которые должны представляться на контекстной диаграмме в виде классов, является нетривиальной задачей.

Обычно используют следующий способ:

– формируют множество понятий-кандидатов из существительных, характеризующих предметную область в описании вариантов использования;

– исключают понятия, не существенные для данного варианта использования.

Пример. Построить концептуальную модель для АИС менеджера торговой организации. Исходные данные для построения модели берутся из диаграммы вариантов использования.

Множество понятий-кандидатов для данной разработки включает следующие словосочетания (под *задачей* будем понимать информационный процесс прогнозирования):

Задание, ввод данных, выбор данных из базы, алгоритм решения задачи, список конкретных алгоритмов решения задачи, полнота описания задания, результаты, данные, база данных..

Цель основного варианта использования системы – выполнение поставленной задачи – *прогнозирование запаса товара определенного наименования на складе предприятия.*

Полное описание поставленной задачи включает: данные и указание на алгоритм. С ним же будут связаны и полученные результаты. Данные могут сохраняться в базе или вводиться оператором - менеджером. Описание задания и все, что с ним связано, может сохраняться в базе.

Возможные обобщения:

– способ задания данных: ввод данных, выбор данных из базы;

– алгоритм: алгоритм решения задачи: конкретный алгоритм решения задачи.

Отсюда возможен переход к построению концептуальной модели, представленный на рисунке 2.5.

Основным классом-понятием, исходя из модели ДВИ, является *Задание* т.е.

перечень возможных аналитических приложений АИС, в том числе и *процесс прогнозирования товара соответствующего наименования (управление запасами)*. С ним связаны классы-понятия *Данные, Алгоритм* и *Результаты*.

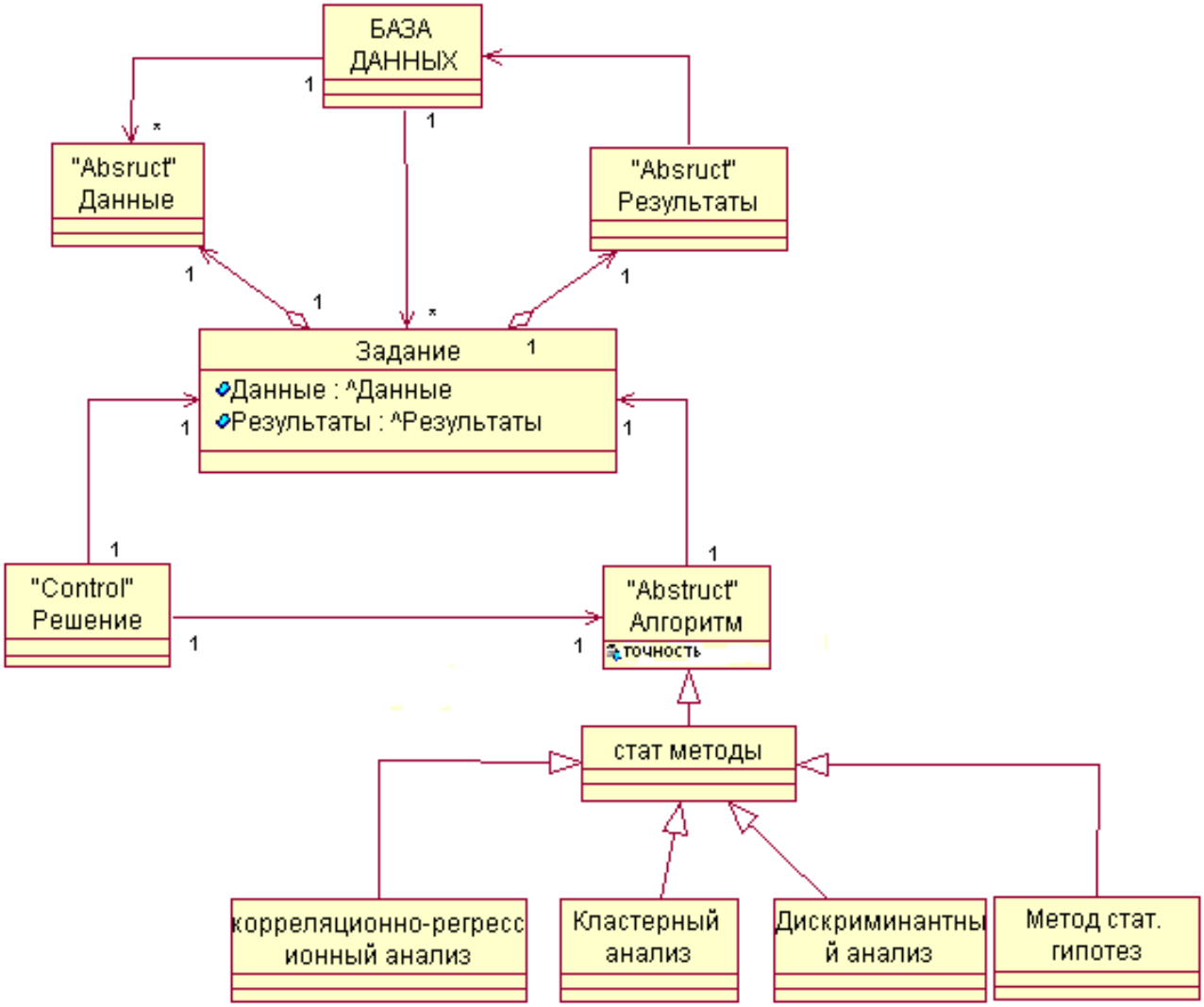


Рисунок 2.5 – Концептуальная модель АИС

В разрабатываемой системе планируется реализовать регрессионный анализ, как модель прогнозирования количества товара заданного наименования на складе торговой организации. Следовательно, класс-понятие *Алгоритм* является супертипом для классов *Корреляционно-регрессионный анализ, Кластерный анализ, Дискриминантный анализ, статистические методы*, от которых, в свою очередь, будут наследоваться *Алгоритмы*, реализующие конкретные методы. *Алгоритм* также связан с *Данными* и *Результатами*.

Данные и *Задания* должны храниться в *Базе данных*, что показывается ассоциациями соответствующих классов. Способ задания данных для понимания основной концепции проектируемой системы пока не очень существенен.

Вид задачи скорее атрибут класса *Задание*, чем самостоятельный класс, так как в реальном мире – это имя, которое позволяет уточнить группу возможных алгоритмов решения, а также структуры исходных данных и получаемых результатов. Для алгоритма существенной характеристикой является его точность, соответственно добавляется атрибут *Точность*. Другие атрибуты пока не проявились.

Таким образом, концептуальная модель предметной области построена. Следует отметить, что концептуальная модель отражает статику будущей системы.

2.2.3 Вопросы и задания для самоконтроля

1 Какие основные задачи ставятся на этапе анализа ?

2 Какие варианты использования обеспечивают выполнение необходимых настроек системы и её обслуживания ?

3 Действующее лицо это…

4 Что такое вариант использования?

5 Связь это … .

6 Отношение между подобными вариантами использования, различающимися наличием в одном из них некоторых дополнительных действий называют….

7 На каком этапе разработки программного обеспечения используют концептуальную модель?

8 Что понимают под концептуальной моделью?

9 На каких уровнях строятся диаграммы классов?

10 На каких этапах разработки используют концептуальную модель, модель уровня спецификации, модель уровня реализации.

2.3 Спецификация описания поведения программного обеспечения при использовании UML

Диаграмма классов представляет собой логическую модель статического представления моделируемой системы. Однако, для большинства физических систем, кроме самых простых, этого представления недостаточно, т.к. необходимо представить поведение системы более детально на логическом уровне. Для этого в языке UML могут использоваться сразу несколько канонических диаграмм: состояний, деятельности, последовательности и кооперации, каждая из которых фиксирует внимание на отдельном аспекте функционирования системы.

2.3.1 Методика построения диаграммы последовательностей

Диаграмма последовательностей (SEQUENCE DIAGRAM) – визуальная модель, которая для определенного сценария варианта использования показывает генерируемые действующими лицами события и их порядок во времени.

Временной аспект поведения имеет существенное значение при моделировании синхронных процессов, описывающих взаимодействия объектов. Именно для этой цели и используются диаграммы последовательности (ДП), в которых ключевым моментом является динамика взаимодействия объектов во времени.

ДП имеет два измерения:

– слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии;

– вертикальная временная ось, направленная сверху вниз, на которой начальному моменту времени соответствует самая верхняя часть диаграммы.

Нотации ДП представлены на рисунке 2.6.

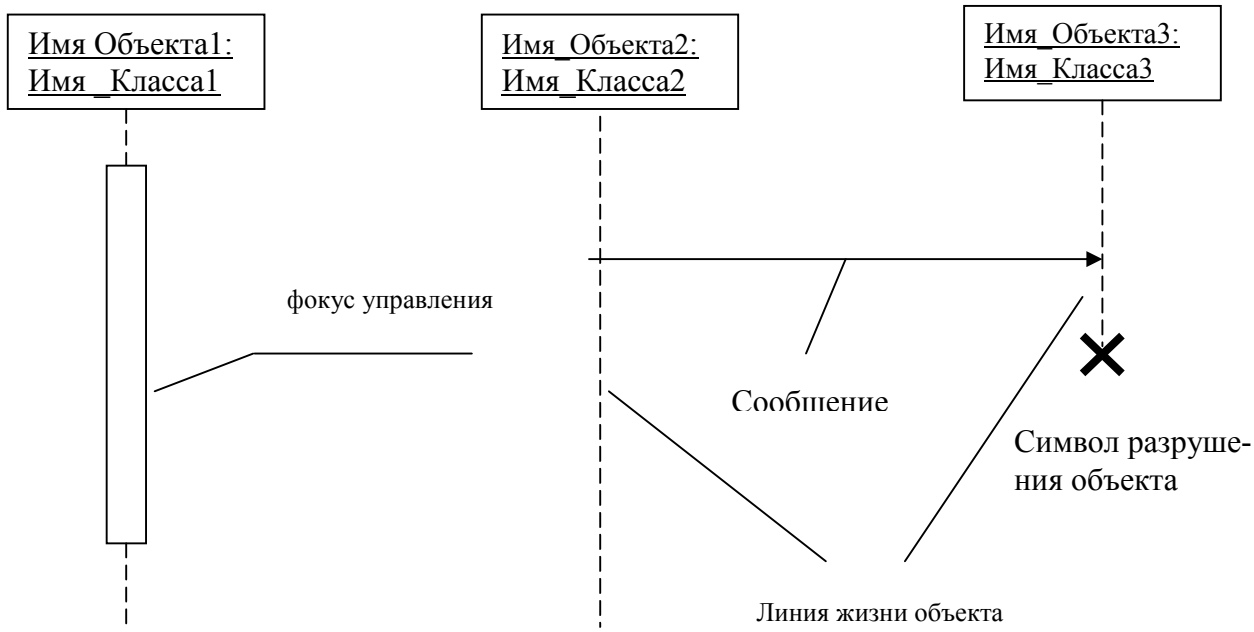


Рисунок 2.6 – Нотации диаграммы последовательностей

Объекты. Все объекты на диаграмме последовательности образуют некоторый порядок, определяемый степенью активности этих объектов при взаимодействии друг с другом. Крайним слева на диаграмме изображается **объект**, который является **инициатором** взаимодействия, правее изображается другой объект, который непосредственно взаимодействует с первым. При этом взаимодействие объектов реализуется посредством **сообщений**, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и образуют порядок по времени своего возникновения, т.е., сообщения, расположенные на диаграмме последовательности выше, инициируются раньше тех, которые расположены ниже. Масштаб на оси времени не моделируется, т.к. диаграмма моделирует лишь временную упорядоченность взаимодействий типа «раньше – позже».

Линия жизни объекта служит для обозначения периода времени, в течение которого объект существует в системе и может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то его линия жизни должна продолжаться по всей плоскости диаграммы от верхней части до самой

нижней. Отдельные объекты, выполнив свою роль, могут быть уничтожены (разрушены), чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML используется специальный символ (крест).

Отдельные объекты в системе могут создаваться по мере необходимости, существенно экономя ресурсы и повышая ее производительность. Символ такого объекта изображается на диаграмме последовательности в той ее части, которая соответствует моменту создания объекта и по оси времени совпадает с моментом его возникновения в системе.

Фокус управления. В процессе функционирования системы одни объекты ее могут находиться в активном состоянии, непосредственно выполняя определенные действия или находиться в состоянии пассивного ожидания сообщений от других объектов. Активность объекта в языке UML на ДП соответствует **фокусу управления.** Фокус управления может получить только существующий объект, у которого в этот момент «существует» линия жизни. Если же некоторый объект уничтожен, то вновь возникнуть в системе он уже не может, а вместо него может быть создан другой экземпляр этого же класса, который будет являться другим объектом. Верхняя часть изображения фокуса управления соответствует началу получения фокуса, а нижняя часть - окончание фокуса управления.

Периоды активности объекта могут чередоваться с периодами его пассивности, а также, если объект на протяжении своей линии жизни является активным, то фокус управления может заменить его линию жизни.

Сообщения. Каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой. Сообщение - законченный фрагмент информации, который отправляется одним объектом другому. Прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено, т.е. сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением. На ДП все сообщения упорядочены по времени своего возникновения в моделируе-

мой системе. Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают от принимающего объекта выполнения ожидаемых действий. Сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением. На диаграмме последовательности все сообщения упорядочены по времени своего возникновения в моделируемой системе.

Существуют следующие виды сообщений:

- простое (simple);
- синхронное (synchronus);
- сообщение с отказом (balking);
- сообщение с лимитированным временем ожидания (timeout);
- асинхронное (asynchronus).

Простое сообщение используется по умолчанию, оно показывает, что все сообщения выполняются в одном потоке управления. Условное обозначение такого сообщения представлено на рисунке 2.7

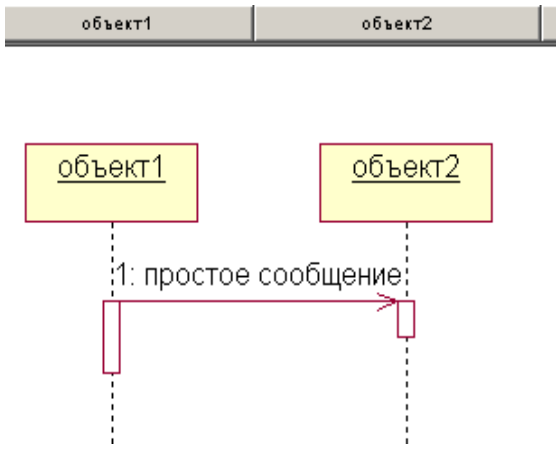


Рисунок 2.7 – Условные обозначения простой передачи сообщения

Синхронное (Synchronous) сообщение применяется, когда *Объект1* посылает сообщение и ждет ответа пользователя (*Объект2*). Условное обозначение такого сообщения представлен на рисунке 2.8

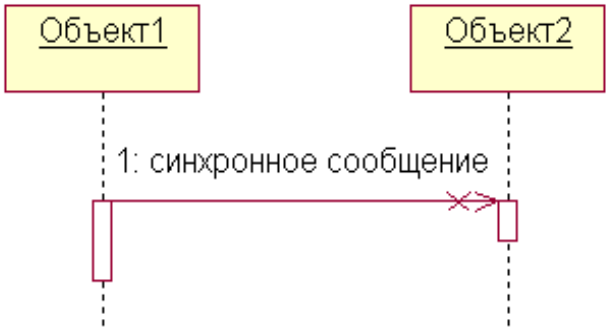


Рисунок 2.8 – Условные обозначения синхронной передачи сообщения

Сообщение с отказом становиться в очередь (Balking). *Объект1* посылает сообщение *Объекту2*. Если *Объект2* не может немедленно принять сообщение, оно отменяется.

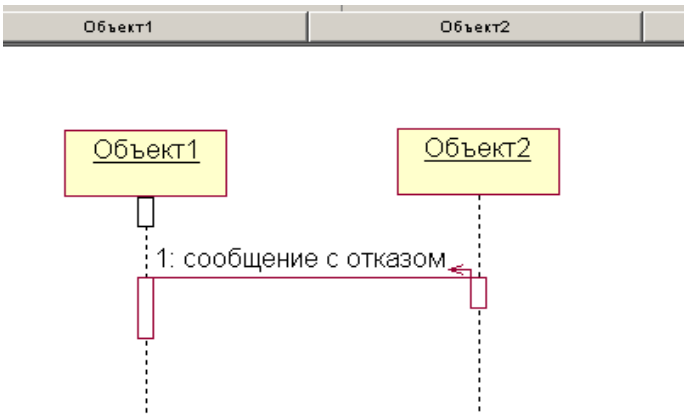


Рисунок 2.9 – Условные обозначения передачи сообщения с отказом

При сообщении с лимитированным временем ожидания (Timeout) *Объект1* посылает сообщение *Объекту2*, а затем ждет указанное время. Если в течение этого времени *Объект2* не принимает сообщение, оно отменяется.

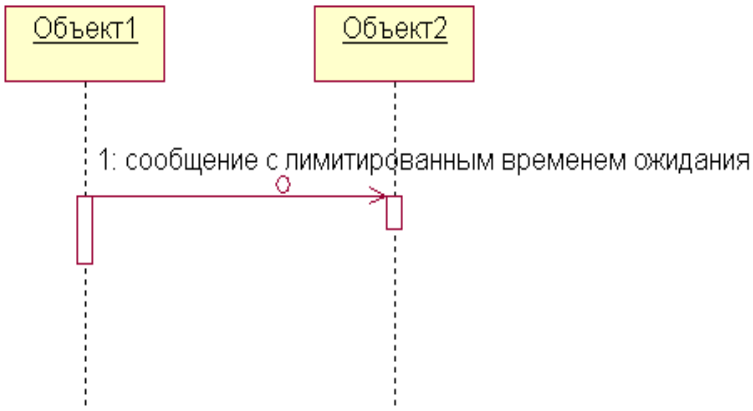


Рисунок 2.10 – Условные обозначения сообщения с лимитированным временем ожидания

Асинхронное сообщение (Asynchronous) это сообщение при котором *Объект1* посылает сообщение *Объекту2* и продолжает свою работу, не ожидая подтверждения о получении

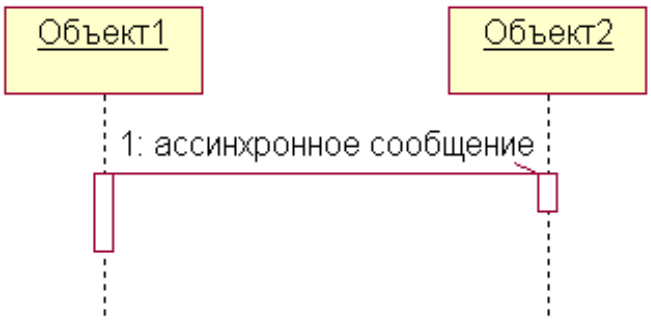


Рисунок 2.11 – Условные обозначения асинхронного сообщения

Ветвление потока управления. Для изображения ветвления рисуются две и более стрелки, выходящие из одной точки фокуса управления объекта. При этом соответствующие условия должны быть явно указаны рядом с каждой из стрелок в форме сторожевого условия.

Стереотипы сообщений. В языке UML предусмотрены некоторые стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Они могут быть указаны на ДП в форме стереотипа рядом с сообщением, к которому они относятся. В этом случае они записываются в кавычках. Используются сле-

дующие обозначения для моделирования действий:

– ***call*** (вызвать) - сообщение, требующее вызова операции или процедуры принимающего объекта;

– ***return*** (возвратить) - сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту;

– ***create*** (создать) - сообщение, требующее создания другого объекта для выполнения определенных действий;

– ***destroy*** (уничтожить) - сообщение с явным требованием уничтожить соответствующий объект;

– ***send*** (послать) - обозначает посылку другому объекту некоторого сигнала, который асинхронно инициируется одним объектом и принимается другим (отличается от сообщения тем, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу).

Системные события и операции. В отличие от внутренних событий, события, которые генерируются для системы действующими лицами, называют системными. Каждую системную операцию называют по имени соответствующего сообщения. При этом система рассматривается как единое целое.

Множество всех системных операций определяют, идентифицируя системные события всех вариантов использования. Для наглядности системные операции изображают в виде операций абстрактного класса (типа) *System*. Если необходимо разделить множество операций на подмножества, инициируемые разными пользователями, то используют несколько абстрактных классов: *System1, System2* и т. д.

Пример Разработать диаграмму последовательностей системы для варианта использования *Инициализация процесса прогнозирования товара*.

Анализ ДВИ позволяет определить, что действующее лицо должно инициировать системные события, включая загрузку задания из базы, которая логически следует из операции сохранения. На рисунке 2.11 изображена ДП системных событий.

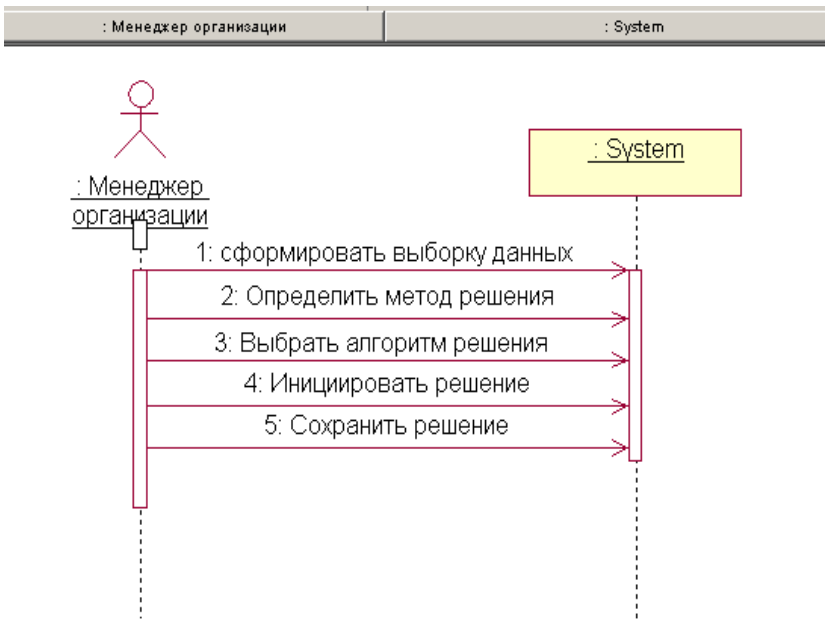


Рисунок 2.12 – Диаграмма последовательностей системы

Следовательно, система должна обеспечивать выполнение указанных операций, которые приписываются классу System (рисунок 2.13), а в скобках указываются параметры, которые должны формировать эти события.

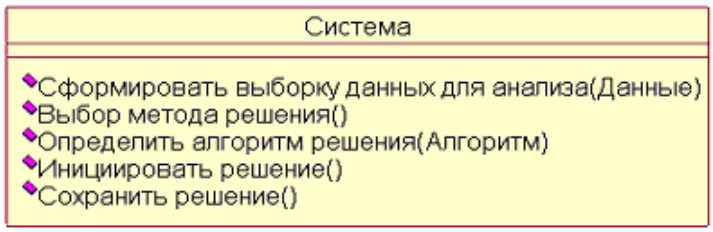


Рисунок 2.13 – Системные операции

Каждую системную операцию необходимо описать. Обычно описание системной операции содержит: имя операции и ее параметры, описание обязанности, указание типа, названия вариантов использования, в которых она используется, примечания для разработчиков алгоритмов, описание обработки возможных исключений, описание вывода не интерфейсных сообщений, предположение о состоянии системы до выполнения операции (предусловие), описание изменения состояния системы после выполнения операции (постусловие).

В таблице 2.6 описаны операции *Инициировать процесс прогнозирования.*

Таблица 2.6 – Описание операции *Инициировать процесс прогнозирования*

Глава	Описание
Имя	Инициировать процесс прогнозирования
Обязанности	Выполнить решение(построить модель) и вывести результаты пользователю
Тип	Системная
Ссылки	Вариант использования
Примечания	Предусмотреть возможность прерывания процесса пользователем
Исключения	1 Если в задании указаны не все исходные данные, то вывести сообщение об ошибке. 2 Если при указанных исходных данных решение задачи указанным методом невозможно, то вывести сообщение об ошибке
Вывод	----------
Предусловие	Предполагает наличие всех исходных данных задания
Постусловие	Получен результат (построена модель прогнозирования)

В зависимости от степени детализации ДП так же, как диаграммы классов, используют на разных этапах проектирования, детализируя поведение ПО предыдущего этапа.

Таким образом, диаграмма последовательностей позволят моделировать поведение ПО на различных этапах проектирования.

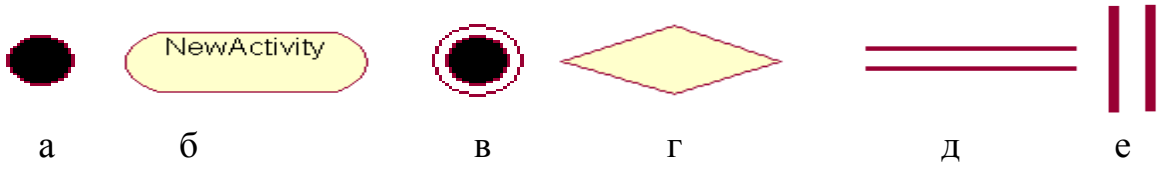
2.3.2 Методика построения диаграммы деятельности

При моделировании поведения проектируемой системы возникает необходимость детализировать особенности алгоритмической и логической реализации выполняемых системой операций. Для моделирования процесса выполнения операций в языке UML используются так называемые диаграммы деятельности (ДД). Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, переход в следующее состояние срабатывает только при завершении этой операции. Графически ДД представляется в форме графа, вершинами

которого являются состояния действия, а дугами - переходы от одного состояния действия к другому. Поэтому ДД (ACTIVITY DIAGRAM) – частный случай диаграмм состояний. Основная **цель** использования таких диаграмм – визуализация особенностей реализации **операций классов**, когда необходимо представить алгоритмы их выполнения.

Диаграмма деятельности – визуальная модель, являющаяся обобщенным представлением алгоритма, реализующего анализируемый вариант использования.

На ДД деятельность обозначается прямоугольником с закругленными углами (рисунок 2.14 б).



а - начало; б - деятельность; в - конец; г- выбор; д - горизонтальные линейки синхронизации; е - вертикальные линейки синхронизации

Рисунок 2.14 - Условные обозначения диаграммы деятельностей

ДД позволяют описывать альтернативные и параллельные процессы. Для обозначения альтернативных процессов используют ромб (рисунок 2.13, г), условие указывают над ним слева или справа, а альтернативы «да», «нет» – рядом с соответствующими выходами. С помощью этого же блока можно построить циклический процесс.

Множественность активации деятельности обозначают символом «*», помещенным рядом со стрелкой активации, и при необходимости уточняют надписью вида «для каждой строки».

Для обозначения параллельных процессов используют линейки синхронизации (рисунок 2.15, д, е), причем условие синхронизации можно уточнить, указав его на диаграмме.

Таким образом, ДД на этапе анализа можно использовать вместо описания ва-

риантов использования или как дополнение к ним.

Пример. Построить ДД этапа анализа, уточняющую вариант использования *инициировать процесс прогнозирования товара на складе,* автоматизированной информационной системы менеджера торгового предприятия.

Учитывая описание предметной области в виде контекстной диаграммы классов (концептуальная модель), и анализируя описание выбранного варианта использования (декомпозиция процессов на отдельные операции), получена диаграмма деятельностей, представленная на рисунке 2.15.

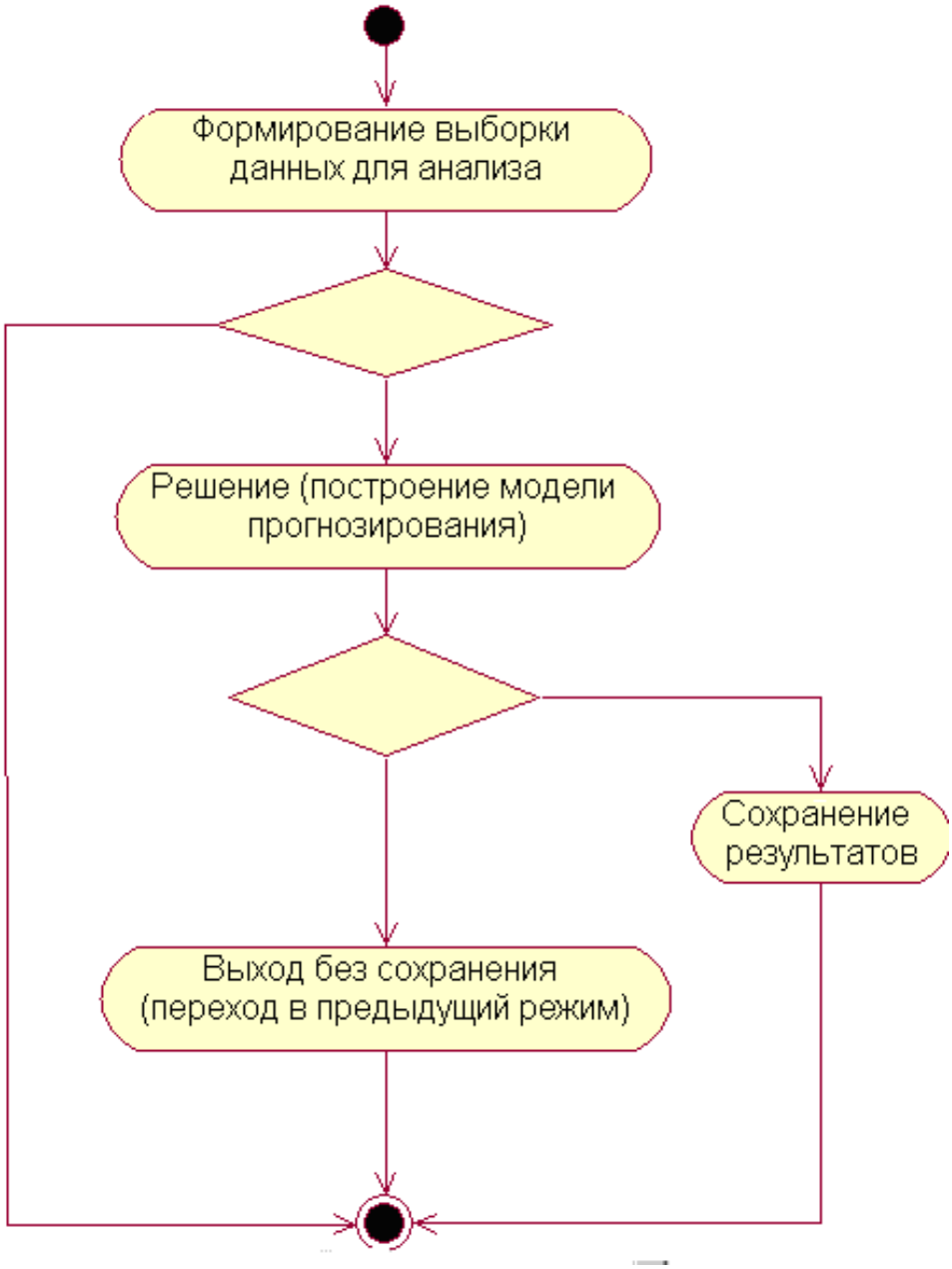


Рисунок 2.15 – ДД варианта использования *инициировать процесс прогнозирования*

Иногда возникает необходимость представить на ДД некоторое сложное действие, которое, в свою очередь, состоит из нескольких более простых действий. В этом случае можно использовать специальное обозначение состояния поддеятельности (subactivity state). Эта конструкция может применяться к любому элементу языка UML, который поддерживает «вложенность» своей структуры. Примером такой ситуация является деятельность *решения задачи прогнозирования,* которое в свою очередь включает в себя последовательность «простых» действий метода корреляционно-регрессионного анализа (рисунок 2.16).

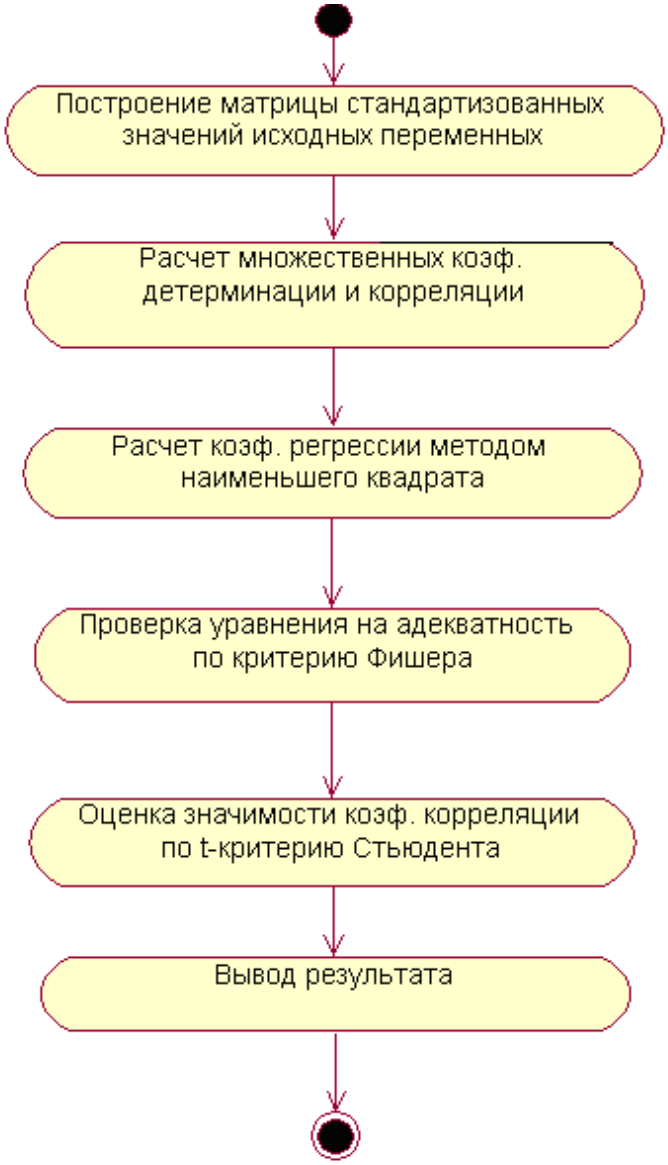


Рисунок 2.16 – Диаграмма деятельностей, уточняющая действие *инициировать процесс прогнозирования*

2.3.3 Вопросы и задания для самоконтроля

1 Что представляет собой диаграмма последовательностей?

2 Какие диаграммы UML применяют для описания поведения разрабатываемого ПО?

3 Что необходимо для построения диаграммы последовательностей ?

4 Какие события называют системными?

5 Что понимается под системными операциями?

6 Какие разделы содержит описание системных операций?

7 На этапе анализа требований и уточнения спецификаций диаграммы деятельностей позволяют конкретизировать основные функции разрабатываемого ПО. Что понимают под деятельностью в данном случае?

8 Какие процессы позволяют описывать диаграммы деятельности?

9 Как на диаграмме деятельности обозначаются альтернативные процессы? Поясните специфику такого обозначения.

10 Для обозначения каких процессов используются линейки синхронизации?

2.4 Спецификация программного обеспечения при использовании UML на этапе проектирования

Основной задачей логического проектирования при объектном подходе является разработка классов для реализации объектов, полученных при объектной декомпозиции, что предполагает полное описание полей и методов основных классов.

2.4.1 Структура программного обеспечения при объектном подходе

Для автоматизированных систем количество классов-кандидатов и других ресурсов велико, поэтому их объединяют в группы, получивших наименование пакетов. **Пакетом** при объектном подходе называют совокупность описаний классов и других программных ресурсов. При этом в один пакет обычно собирают классы и другие ресурсы единого назначения.

Диаграмма пакетов является визуальной моделью ПО, которая показывает из каких частей состоит структура проектируемой программной системы и как эти части связаны друг с другом.

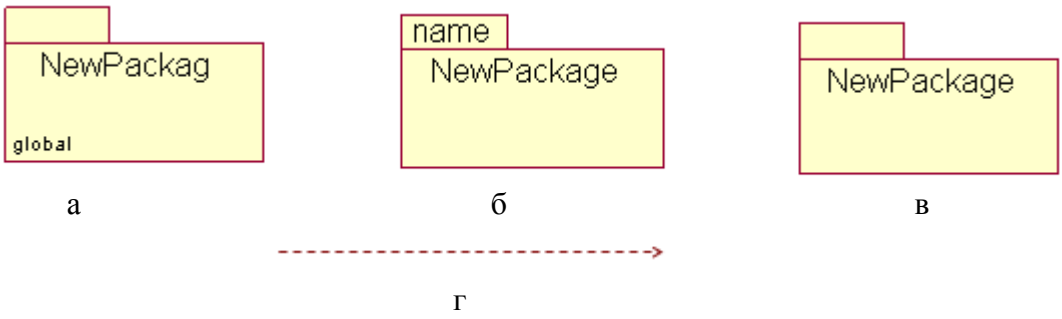
Связь между пакетами фиксируют, если изменения в одном пакете могут повлечь за собой изменения в другом. Возможны различные виды зависимости классов, например:

- объекты одного класса посылают сообщения объектам другого класса;
- объекты одного класса обращаются к компонентам объектов другого;
- объекты одного класса используют объекты другого в списке параметров методов и т. п.

Самыми хорошими технологическими характеристиками отличается вариант, при котором каждый пакет включает интерфейс, содержащий описание всех ресурсов данного пакета, и взаимодействие пакетов осуществляется только через этот интерфейс. Изменения реализации ресурсов пакета в этом случае не затрагивает других пакетов и только изменения в интерфейсе могут потребовать изменения пакетов, использующих ресурсы данного пакета.

Пакеты, с которыми связаны все пакеты программной системы, называются глобальными.

На рисунке 2.17 приведены нотации UML, которые используются на диаграммах пакетов.



а - глобальный пакет; б - пакет с обозначением содержимого; в - пакет; г - зависимость классов (стрелка указывает направление вызовов)

Рисунок 2.17 – Нотации диаграммы пакетов

Кроме указанных обозначений на диаграммах пакетов допустимо показывать обобщения, что, как правило, подразумевает наличие единого интерфейса нескольких пакетов. В этом случае фиксируется связь от пакета-подтипа к пакету-супертипу.

Пример. Разработать диаграмму пакетов автоматизированной системы менеджера торговой организации.

Анализ концептуальной модели и ДВИ позволяют выделить следующие группы классов или пакетов:

– *Пользовательский интерфейс* (классы, реализующие объекты интерфейса с пользователем);

– *Библиотека интерфейсных компонентов* (классы, реализующие интерфейсные компоненты: окна, кнопки, метки и т. п.);

– *Объекты управления* (классы, реализующие сценарии вариантов использования);

– *Объекты задачи* (классы, реализующие объекты предметной области системы);

– *Интерфейс базы данных* (классы, реализующие интерфейс с базой данных);

– *База данных*;

– *Базовые структуры данных* (классы, реализующие внутренние структуры данных, такие, как деревья, n-связные списки и т. п.);

– *Обработка ошибок* (классы исключений, реализующие обработку нештатных ситуаций).

Последние два пакета являются глобальными, так как их элементы могут использовать классы всех пакетов.

Диаграмма пакетов АИС менеджера примет вид, представленный на рисунке 2.18

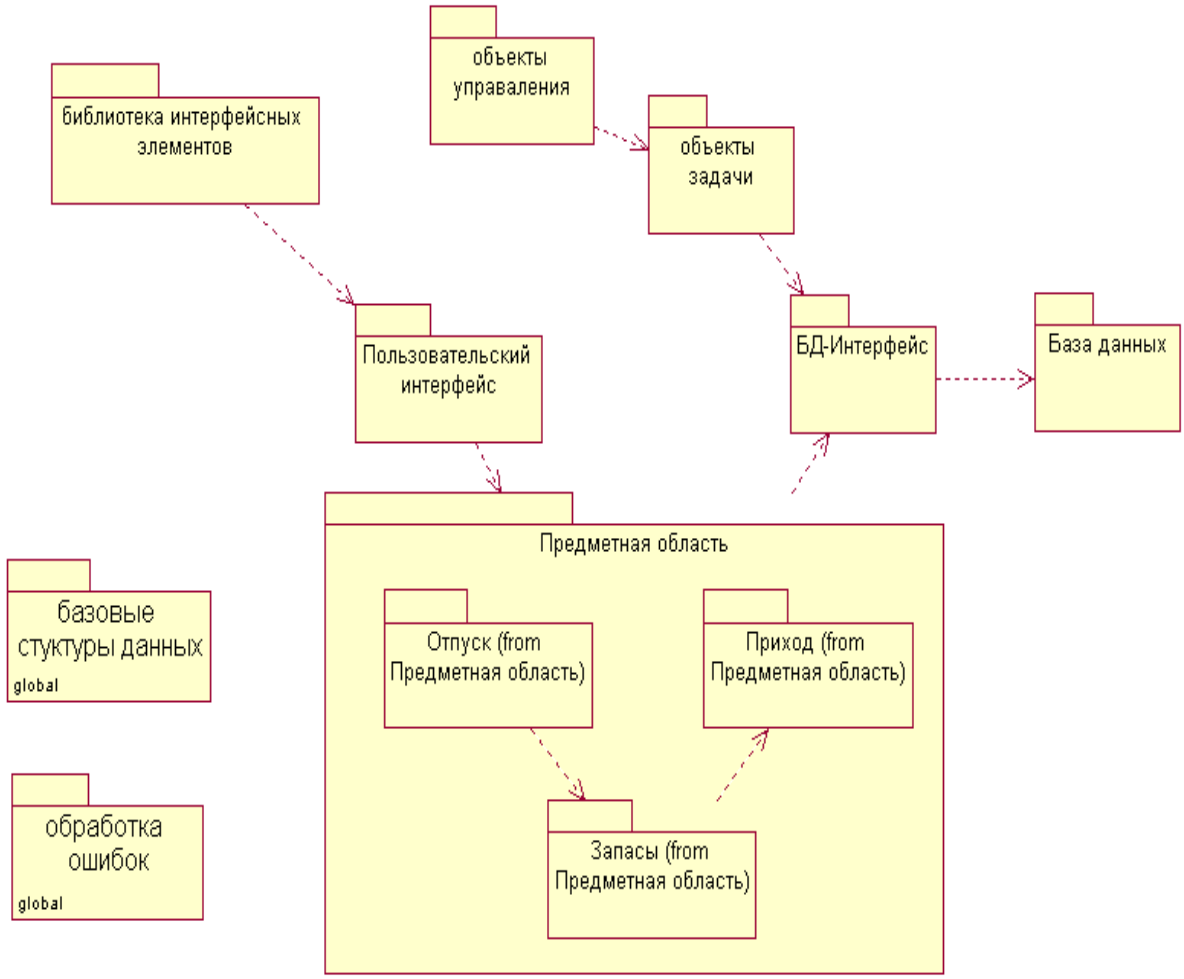


Рисунок 2.18 – Диаграмма пакетов АИС менеджера торгового предприятия

Таким образом, диаграмма пакетов является дополнительным инструментом визуального моделирования и анализа сложных программных систем, к которым относятся автоматизированные информационные системы.

2.4.2 Модели поведения программных систем этапа проектирования

После определения основных пакетов разрабатываемого ПО переходят к детальному проектированию классов, входящих в каждый пакет. Классы-кандидаты, которые предположительно должны войти в конкретный пакет, показывают на диаграмме классов этапа проектирования и уточняют отношения между объектами указанных классов.

Пример. Определить классы – кандидаты пакета *Объекты задачи* приложения автоматизированной информационной системы.

Анализ концептуальной модели предметной области и описание основного варианта использования *Прогнозирование запаса товара* и его диаграммы деятельностей позволил сформировать список классов-кандидатов:

– класс *Задание* (объекты данного класса должны создаваться каждый раз, когда пользователь инициирует новое задание);

– семейство классов с базовым классом *Алгоритм* (объекты данного класса должны создаваться, когда определен алгоритм решения задачи);

– класс *Данные* (объекты данного класса должны создаваться при определении данных – вводе или выборе из базы);

– класс *Результаты* (объекты класса должны создаваться при решении конкретной задачи конкретным алгоритмом с использованием конкретных данных).

Основой для проектирования классов является уточнение взаимодействия объектов этих классов в процессе реализации вариантов использования. При этом применяют диаграммы последовательностей и диаграммы кооперации этапа проектирования. Если необходимо описать взаимодействие объектов при обработке конкретного сообщения, удобны именно диаграммы последовательности.

Диаграмма последовательности этапа проектирования показывают внутренние объекты, а также последовательность сообщений, которыми обмениваются объекты в процессе реализации фрагмента варианта использования, называемого **сценарием**.

Каждое взаимодействие объектов описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой. В этом смысле сообщение (message) представляет собой законченный фрагмент информации, который отправляется одним объектом другому. При этом прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено.

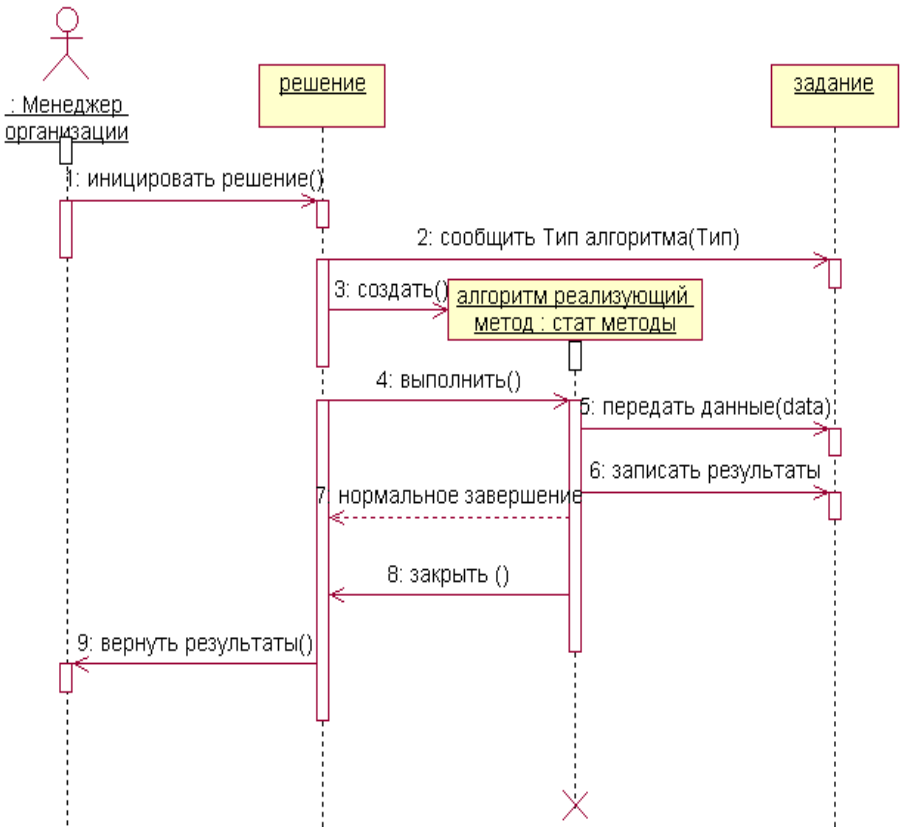
Пример. Разработать диаграмму последовательностей для сценария *Решение задачи* (сценарий варианта использования *прогнозирование запаса товара соответствующего наименования* от момента инициализации пользователем процесса решения до его завершения).

Анализ описания варианта использования показывает, что необходимо рассмотреть три варианта последовательности действий: а) нормальный процесс; б) прерывание процесса пользователем; в) возникновение исключения при выполнении алгоритма.

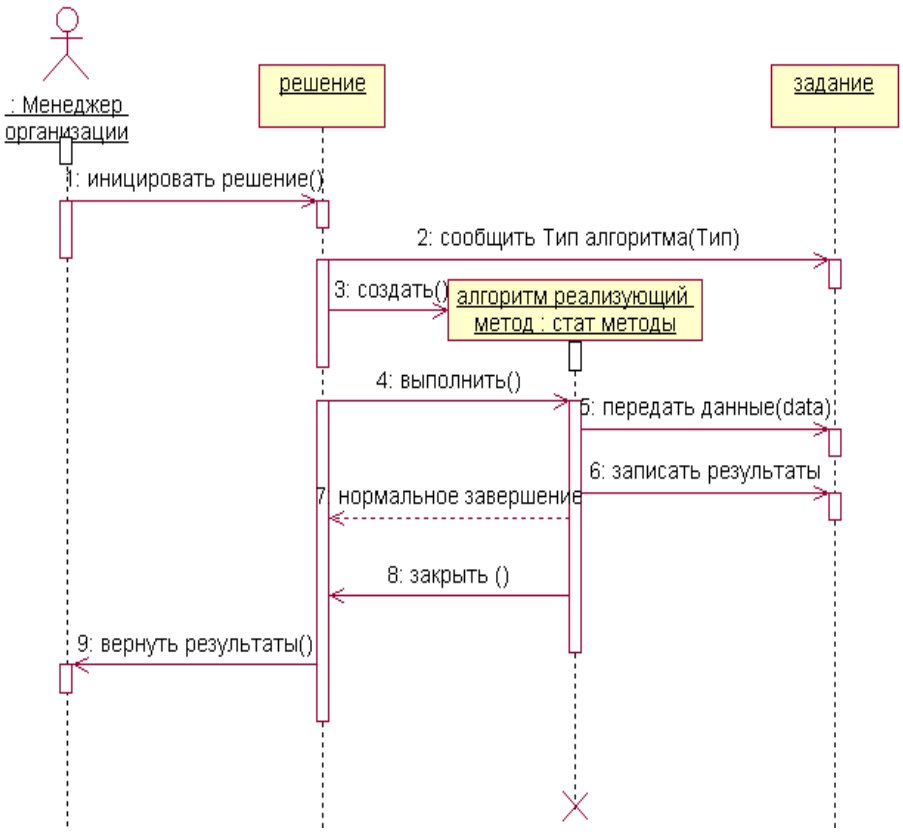
Нормальный процесс (рисунок 2.19а) предполагает, что при выдаче команды *Создать* создается объект *Решение*, управляющий данным сценарием. Следующее сообщение *Начать* активизирует этот объект. Объект *Решение* запрашивает у объекта класса *Задание* тип объекта *Алгоритм*, создает объект требуемого класса и активизирует его, сохраняя способность получать и обрабатывать сообщения (параллельный процесс).

Объект класса *Алгоритм*, реализующий метод, запрашивает у объекта класса *Задание* данные и начинает обработку, используя вспомогательные объекты. Нормально завершив обработку, объект класса *Алгоритм*, реализующий метод, передает объекту класса *Задание* результаты и возвращает объекту *Решение* признак нормального завершения. Объект *Решение* уничтожает объект класса *Алгоритм*, реализующий метод, и возвращает вызвавшему его объекту признак нормального завершения решения.

В случае прерывании процесса объект *Решение* прерывает процесс решения, уничтожает объект *Алгоритм* и возвращает признак прерванного выполнения (рисунок. 2.19 б).



а- нормальный процесс



б- прерывание процесса пользователем

Рисунок 2.19 – Диаграмма последовательностей сценария *Решение задачи*

Обрабатывая исключение, объект класса *Решение*, генерирует соответствующее сообщение пользователю, уничтожает объект класса *Алгоритм*, реализующий метод и возвращает признак завершения выполнения с ошибкой (рисунок 2.20).

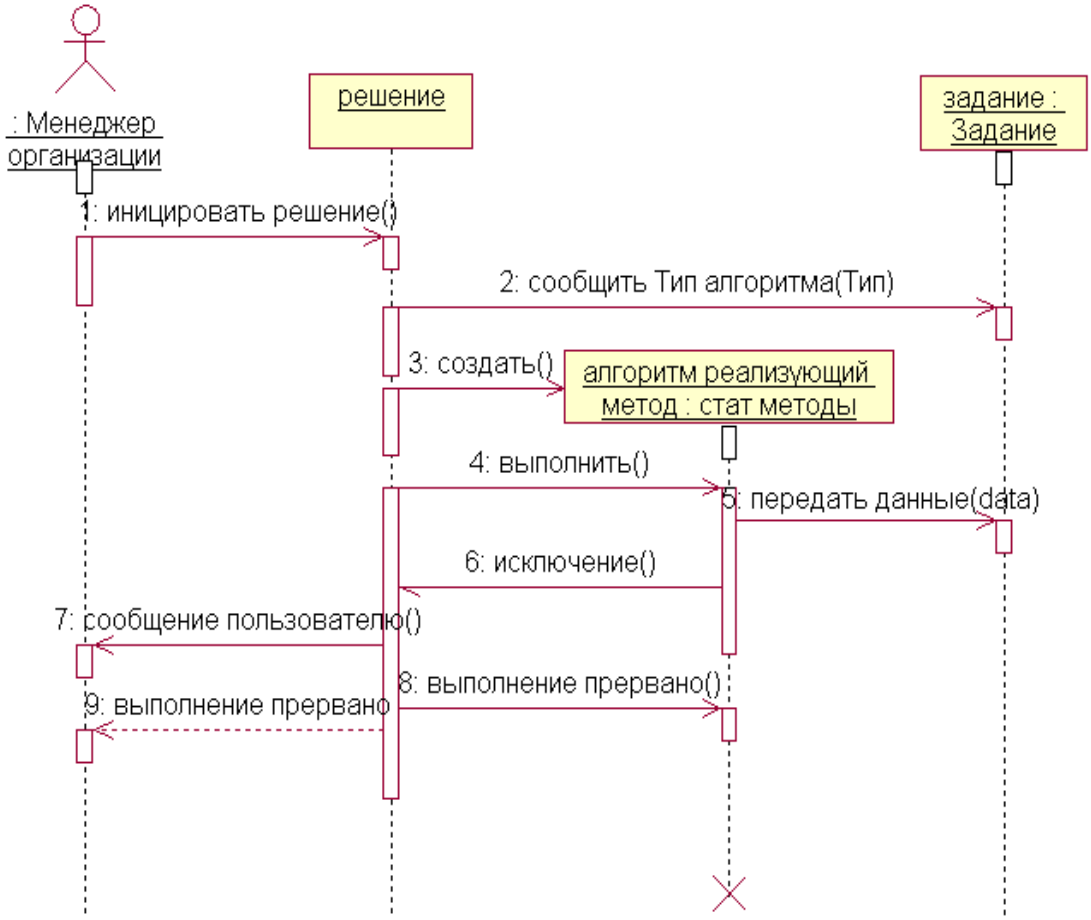


Рисунок 2.20 – Диаграмма последовательности сценария *Решение задачи* для случая возникновения исключения

При разработке диаграмм последовательностей часто применяется двухэтапный подход. На первом этапе отображается информация высокого уровня, которая нужна конечным пользователям проектируемой системы. Сообщения еще не соотносятся с операциями и объекты могут быть не соотнесены с классами. Эти диаграммы позволяют аналитикам, пользователям и всем заинтересованным в бизнес-процессах лицам увидеть, как будут развиваться события в системе. Полученная на первом этапе диаграмма последовательности, уточняющая вариант использования *добавление учетных данных в базу* примет вид, представленный на рисунке 2.21

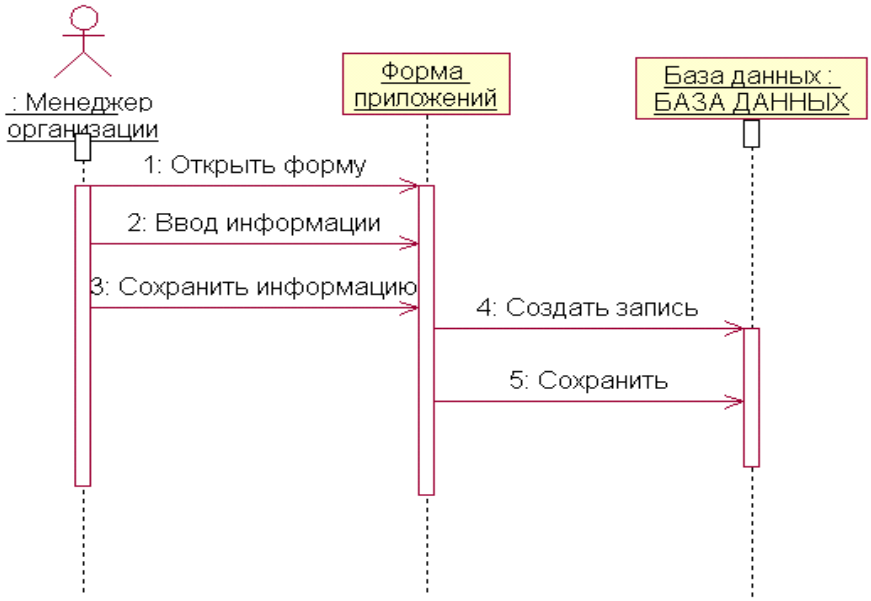


Рисунок 2.21 – Диаграмма последовательности ВИ *Добавление учетных данных в базу данных* (общий вид)

На втором этапе, после того как пользователи придут к согласию по поводу полученной диаграммы, можно уточнить детали. При этом диаграмма может утратить свою полезность для конечных пользователей, но станет важна для разработчиков, тестировщиков и остальных участников команды проекта.

В начале второго этапа в диаграмму могут быть добавлены некоторые новые объекты. Обычно в диаграмму помещают управляющий объект, отвечающий за управление последовательностью событий сценария. Такие объекты называют объектами-менеджерами (manager object). Кроме того, в диаграммы можно поместить и другие объекты, отвечающие за безопасность или обработку ошибок.

Это только одна из диаграмм, необходимых для моделирования варианта использования *формирование заявки на приобретение нового товара*. Она соответствует успешному варианту хода событий. Для описания того, что случится, если возникнет ошибка, придется разработать дополнительные диаграммы. Каждый альтернативный поток варианта использования может быть промоделирован с помощью собственных диаграмм последовательностей.

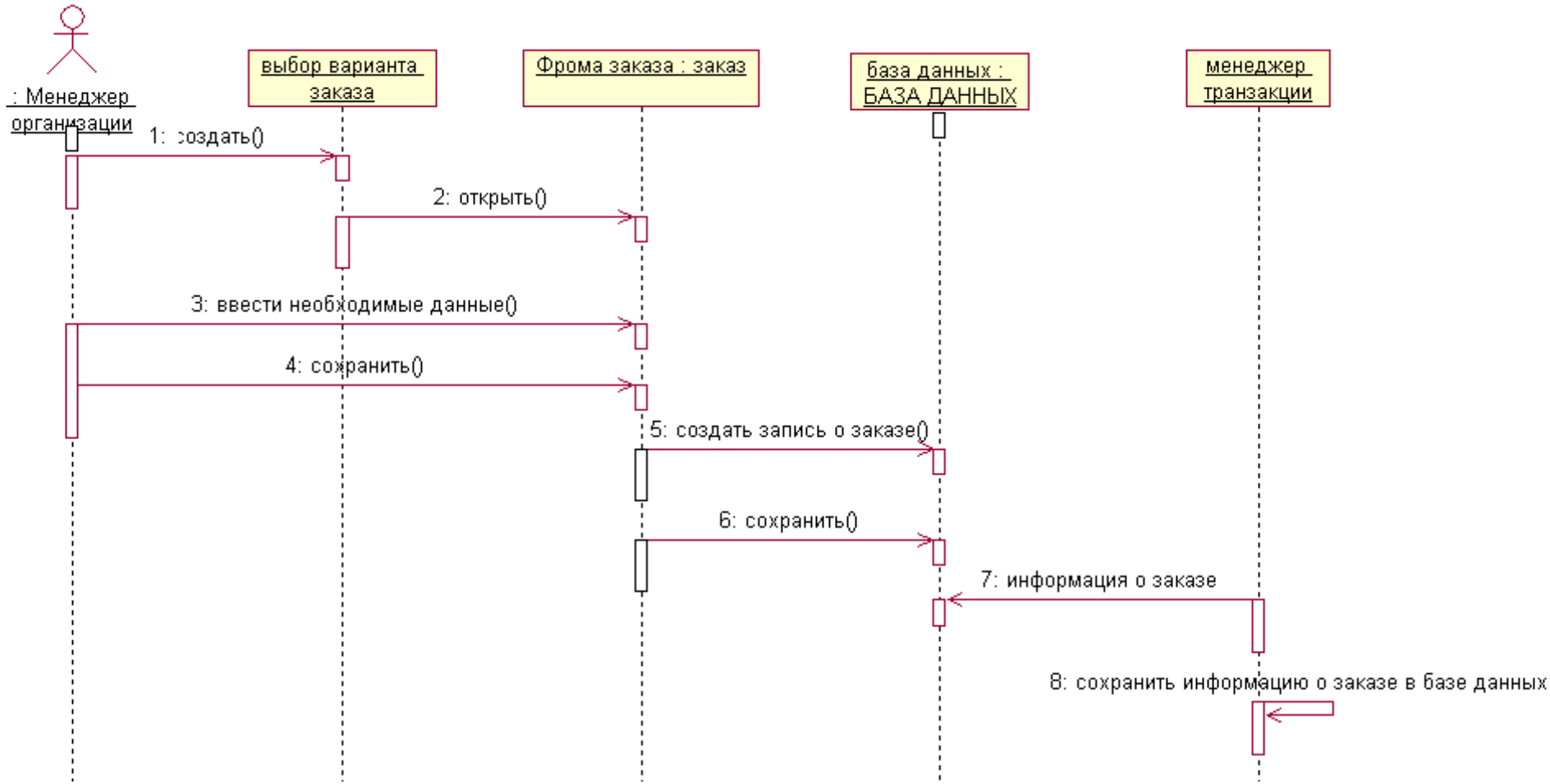


Рисунок 2.22 – Диаграмма последовательности ВИ *Формирование заявки на приобретение товара*

Диаграмма кооперации – это визуальная модель, которая показывает потоки данных между объектами классов, что позволяет уточнить связи между ними. Это альтернативный способ представления взаимодействия объектов в процессе реализации сценария.

Пример. Разработать диаграмму кооперации для сценария *Решение задачи.*

На рисунке 2.23 изображены на одной диаграмме три возможных случая реализации сценария, нумеруя сообщения в порядке их возможной генерации.

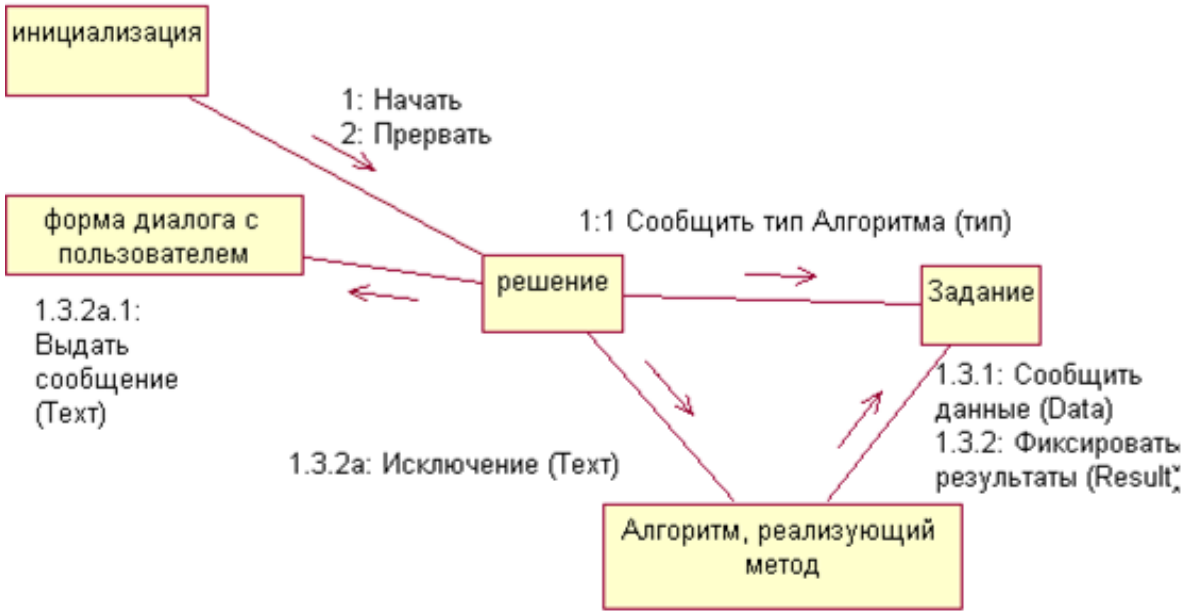


Рисунок 2.23 – Диаграмма кооперации сценария *Процесс решения*

Такое представление позволяет описать потоки данных, передаваемых между объектами классов *Решение*, *Задание* и *Алгоритм*, реализующий метод, для сценария *Решение задачи*.

2.4.3 Методика проектирования классов при использовании UML

Большинство классов при проектировании автоматизированной системы можно отнести к определенному типу.

Классы-сущности используют для представления сущностей реального мира

или внутренних элементов системы, например, структур данных. Как правило, они не зависят от окружения и используются в различных приложениях. Для выявления классов-сущностей изучают описания вариантов использования, концептуальную модель и диаграммы деятельностей. Полученный таким образом список классов-кандидатов фильтруют, удаляя слова, не относящиеся к предметной области, языковые выражения и т.п. Среди оставшихся отбирают классы-кандидаты, объекты которых обладают как состоянием, так и поведением.

а б в г

а - класс-сущность; б - граничный класс; в – управляющий класс;
г – явное указание стереотипа

Рисунок 2.24 – Условное обозначение стереотипов классов

Граничные классы обеспечивают взаимодействие между действующими лицами и внутренними элементами системы. К этому типу относят как классы, реализующие пользовательские интерфейсы, так и классы, обеспечивающие интерфейс с аппаратными средствами или программными системами. Для обнаружения граничных классов изучают пары «действующее лицо – вариант использования».

Управляющие классы служат для моделирования последовательного поведения, заложенного в один или несколько вариантов использования.

Построение диаграммы классов этапа проектирования начинается с уточнения отношений. При этом между классами помимо ассоциации и обобщения различают еще два типа отношений: агрегацию и композицию.

Агрегацией называют ассоциацию между целым и его частью или частями, если отношение «целое-часть» в конкретном случае существенно. Например, если колесо представляет собой часть автомобиля, то между соответствующими класса-

ми целесообразно указать отношение агрегации, а если колесо - товар, также как и автомобиль, то связь целое-часть не существенна.

Композиция - более сильная разновидность агрегации, которая подразумевает, что объект-часть может принадлежать только единственному целому. Объект-часть при этом создается и уничтожается только вместе со своим целым.

Уточненные отношения между классами фиксируют на диаграмме классов. Для этого используют специальные уловные обозначения, представленные на рисунке 2.25.

а- композиция б - агрегация

Рисунок 2.25 - Условные обозначения специальных видов ассоциации

Поскольку отношение ассоциации и его подвиды (агрегация и композиция) означают наличие обмена сообщениями между объектами классов целесообразно уточнить направление передачи сообщений.

Навигацию (направление ассоциации) показывают стрелкой на конце линии ассоциации. Если стрелки указаны с обеих сторон, то это означает *двунаправленную* ассоциацию.

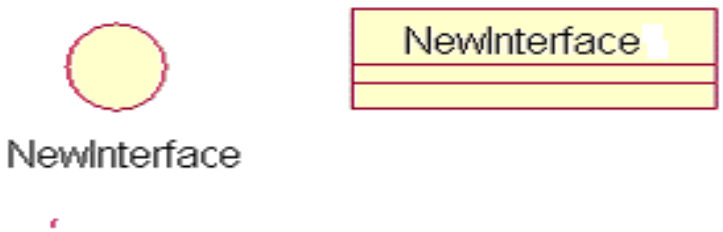
Специальное обозначение на диаграмме классов этапа проектирования используют для указания абстрактных классов и методов: на диаграмме классов их имена выделяют курсивом, либо перед именем класса указывают стереотип «abstract».

Диаграммы классов позволяют также отобразить *ограничения,* которые невозможно показать, используя только понятия, рассмотренные выше (ассоциации, обобщения, атрибуты, операции). Например, показать, что средний балл студентов должен определяться по соответствующей формуле. Подобную информацию на диаграмме классов можно представить в виде записи на естественном языке или в виде математической формулы, поместив их в фигурные скобки.

Особое место в процессе проектирования классов занимает проектирование интерфейсов. **Интерфейсом** в UML называют класс, содержащий только объявление операций. Отдельное описание интерфейсов улучшает технологические качества проектируемого ПО. Интерфейсы широко применяют при разработке сетевого ПО, которое должно идентично функционировать в гетерогенных средах, а также для организации взаимодействия с системами управления базами данных и т. п., так как механизм полиморфного наследования позволяет создавать различные реализации одного и того же интерфейса.

С точки зрения теории объектно-ориентированного программирования интерфейс представляет собой особый вид абстрактного класса, отличающийся тем, что он не содержит методов, реализующих указанные операции, и объявлений полей. Другими словами, абстрактные классы позволяют определить реализацию некоторых методов, а интерфейсы требуют отложить определение всех методов.

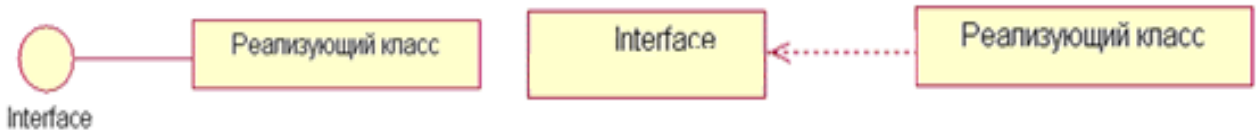
На диаграмме классов интерфейс можно показать двумя способами: с помощью специального условного обозначения (рисунок 2.26, а) или, объявив для класса стереотип «Interface» (рисунок 2.26, б).



а – специальное, б- с указанием стереотипа

Рисунок 2.26 – Нотации интерфейса в UML

Реализацию интерфейса также можно показать двумя способами: сокращенно (рисунок 2.27, а) или, используя отношение реализации (рисунок 2.27, б).



а– сжатая форма б – с указанием отношения реализаци

Рисунок 2.27 – Условные обозначения реализации интерфейсов:

Для остальных классов, ассоциированных с интерфейсом, следует уточнить ассоциацию, показав отношение зависимости. Это отношение в данном случае означает, что класс использует указанный интерфейс (рисунок 2.28), т. е. обращается к описанным в интерфейсе функциям.

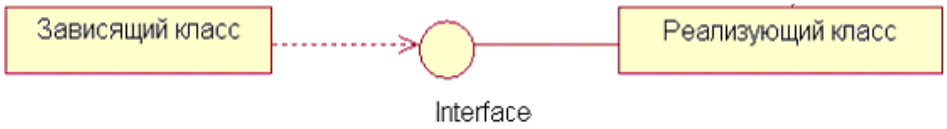


Рисунок 2.28 – Обозначение зависимости класса от интерфейса

Одновременно с уточнением отношений классов в пакете следует продумать и отношения классов, включенных в различные пакеты, между собой.

Пример. Уточнить отношения классов пакета *Объекты задачи* между собой и с классом *Решение* из пакета *Объекты управления*, используя результаты детализации отношений между объектами рассматриваемых классов.

Анализ диаграммы кооперации, представленной на рисунке 2.23, показывает, что:

– класс *Задание* по сути дела представляет собой таблицу, в которой фиксируется вся информация о конкретной задаче: вид задачи, алгоритм решения, данные и результат, причем результат связан с заданием неразрывно, так как теряет смысл вне контекста задания (отношение композиции), а данные имеют смысл сами по себе (отношение агрегации);

– класс *Алгоритм* целесообразно разрабатывать как абстрактный; этот класс будет описывать интерфейс между объектом класса *Решение* и конкретным алгоритмом, а также между объектом класса *Задание* и опять же конкретным алгоритмом;

– отношение между классами Задание и Алгоритм, Решение и Алгоритм, а

также Задание и Решение – ассоциации, направленные к классу Задание.

Кроме того, анализ структур исходных данных и результатов решаемых задач показывает их существенное различие, следовательно, классы *Данные* и *Результаты* также необходимо реализовать как абстрактные и наследовать от них классы, уточняющие структуры данных и результатов для каждого случая. При дальнейшем анализе следует выяснить, будут ли классы *Данные* и *Результаты* описывать какие-либо поля или они только определят интерфейсы, через которые будет осуществляться доступ к данным и результатам конкретных заданий.

На рисунке 2.29 изображена уточненная диаграмма классов автоматизированной информационной системы.

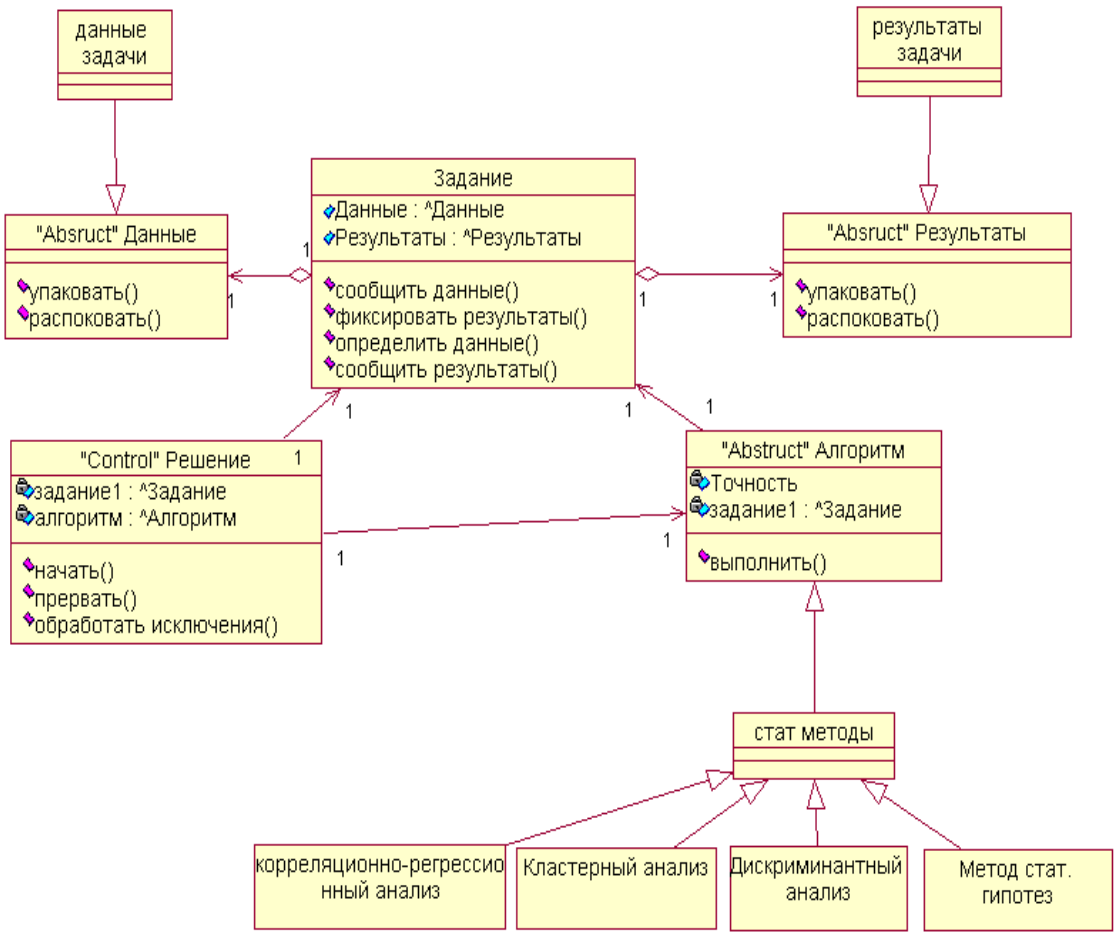


Рисунок 2.30 – Уточненная диаграмма классов пакета *Объекты задачи* и класса *Решение* из пакета *Управляющие объекты*

На диаграмме классов целесообразно также указать множественность объектов. Поскольку каждый раз решается одна задача с единственными данными, ис-

пользуя конкретный алгоритм, и в результате получают единственное решение, все перечисленные выше ассоциации связывают объекты «один к одному».

Уточнение отношений между классами позволяет перейти к собственно построению диаграммы классов этапа реализации.

2.4.4 Вопросы и задания для самоконтроля

1 Для чего используют классы-сущности?

2 Каким образом можно выявить классы-сущности?

3 Какой вид классов обеспечивает взаимодействие между действующими лицами и внутренними элементами системы?

4 Какой вид классов служит для моделирования последовательного поведения, заложенного в один или несколько вариантов использования.

5 Пакетом при объектном подходе называют…

6 Что показывает диаграмма пакетов? Для чего используют диаграммы пакетов?

7 В каком случае фиксируют связь между пакетами?

8 Какие пакеты называются глобальными?

9 Что показывает диаграмма последовательностей этапа проектирования?

10 Какой тип диаграмм показывает альтернативный способ представления взаимодействия объектов в процессе реализации сценария?

2.5 Спецификация программного обеспечения при использовании UML на этапе реализации

Основной задачей этапа реализации является разработка объектов классов, полученных при объектной декомпозиции, что предполагает полное описание полей и методов каждого класса.

2.5.1 Реализация методов классов

Информацию о действиях, которые должны выполняться методами класса, получают, анализируя диаграммы последовательностей этапа проектирования. Однако алгоритмы сложных методов необходимо проработать детально. При этом можно использовать как уже известные нотации (схемы алгоритмов и псевдокоды), так и диаграммы деятельностей.

Следует помнить, что в соответствии с общими правилами процедурной декомпозиции *любую деятельность можно декомпозировать и изобразить в виде диаграммы деятельности более низкого уровня.*

Пример. Построить диаграмму деятельности для операции *Начать()* класса *Решение*. Анализ диаграммы классов (рисунок 2.30) показывает, что данная деятельность затрагивает три объекта уже детализированных классов *Решение, Алгоритм* и *Задание*.

Объект класса *Решения* организует обработку, то есть инициализирует переменные (в том числе определяет тип *Алгоритма*), создает объект класса *Алгоритм* требуемого типа, активизируют обработку, а затем уничтожают объект класса *Алгоритм*.

Объект класса *Задание* должен в ответ на запрос сообщить тип *Алгоритма*, предоставить данные и запомнить результаты.

Объект класса *Алгоритм* отвечает за решение задачи.

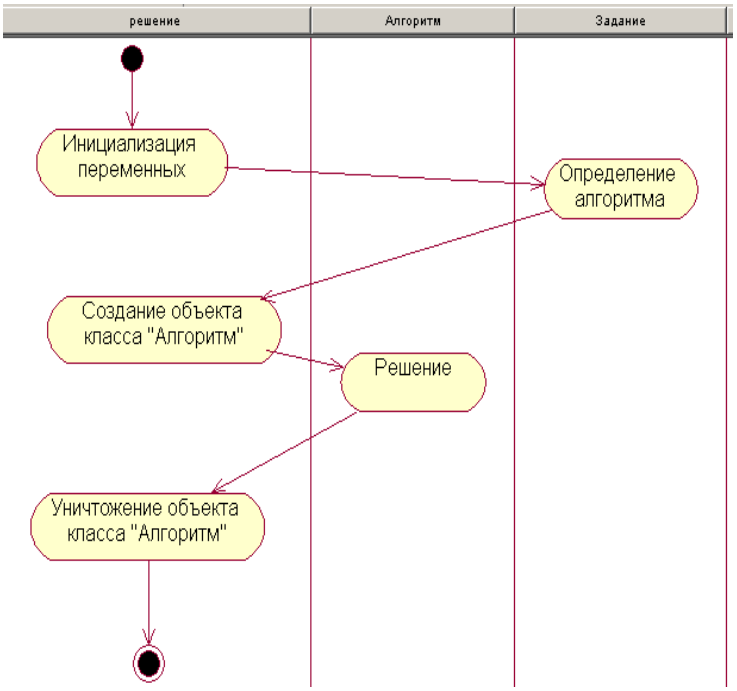


Рисунок 2.31 – Диаграмма деятельности объекта *Решение* для операции *Начать()*

Пример. Реализовать диаграмму деятельности корреляционно-регрессионного анализа функции управления запасами торговой организации.

Если объекты проектируемого класса должны реализовывать сложное поведение, для них целесообразно разрабатывать диаграммы состояний (ДС).

Диаграмма состояний (STATECHART DIAGRAM) – это визуальная модель, описывающая процесс изменения состояний одного экземпляра определенного класса, т.е. моделирующая все возможные изменения в состоянии конкретного объекта.

ДС по существу является графом специального вида, который представляет некоторый автомат. Вершинами этого графа являются **состояния**. Дуги графа служат для обозначения **переходов** из состояния в состояние. ДС могут быть вложены друг в друга, образуя вложенные диаграммы более детального представления отдельных элементов модели.

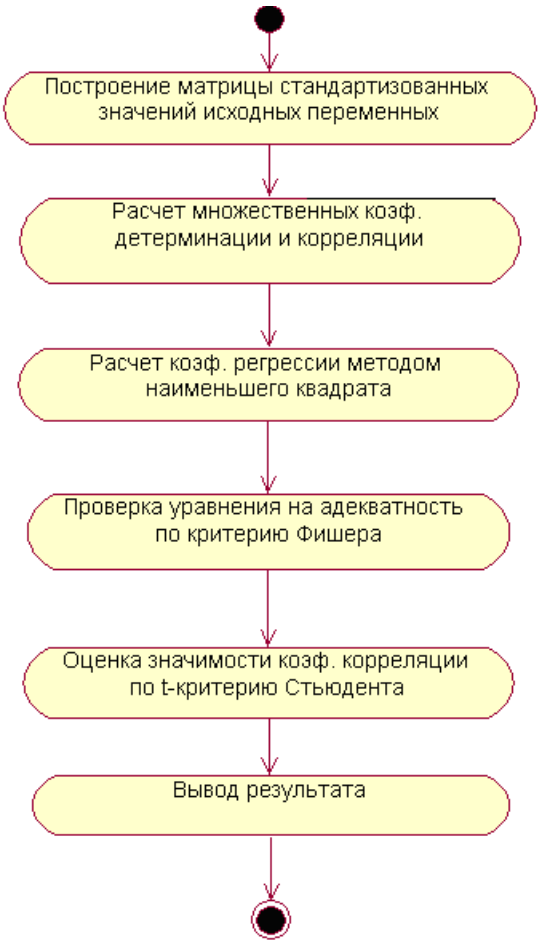


Рисунок 2.32 – Диаграмма деятельностей уточняющая действие *инициировать процесс прогнозирования товара на складе*

Автомат в языке UML представляет собой некоторый формализм для моделирования поведения элементов модели и системы в целом. В метамодели UML автомат является пакетом, в котором определено множество понятий, необходимых для представления поведения моделируемой сущности в виде дискретного пространства с конечным числом состояний и переходов.

Формализм автомата основан на выполнении следующих обязательных условий:

– автомат не запоминает историю перемещения из состояния в состояние;

– в каждый момент времени автомат может находиться в одном и только в одном из своих состояний;

– процесс изменения состояний автомата происходит во времени, однако яв-

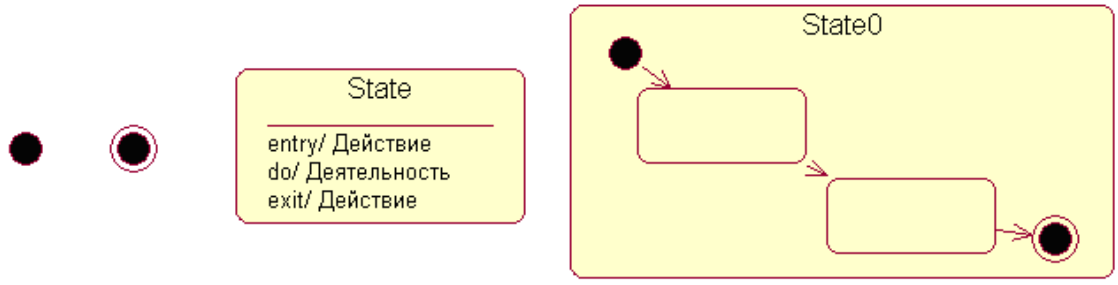
но концепция времени не входит в формализм автомата (переход объекта из состояния в состояние происходит мгновенно);

– количество состояний автомата должно быть обязательно конечным;

– граф автомата не может содержать изолированных состояний и переходов, т.е. для каждого состояния, кроме начального, должно быть определено предшествующее состояние (допускается переход из состояния в себя, такой переход называют «петлей»);

– автомат не должен содержать конфликтующих переходов, когда объект одновременно может перейти в два и более последующих состояния (исключая случай параллельных подавтоматов).

Нотации диаграммы состояний приведены на рисунке 2.33.



а (б) – начальное (конечное) состояние; в – промежуточное состояние; г – суперсостояние.

Рисунок 2.33 – Нотации диаграммы состояний объекта

Основными понятиями, входящими в формализм автомата, являются СОСТОЯНИЕ и ПЕРЕХОД.

Под **состоянием** объекта применительно к ДС понимают ситуацию в жизненном цикле объекта, во время которой он: удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает некоторого события. Состояние может быть задано в виде набора конкретных значений атрибутов класса или объекта, при этом изменение их отдельных значений будет отражать изменение состояний моделируемого класса или объекта.

Имя состояния представляет собой строку текста, которая раскрывает содер-

жательный смысл данного состояния. В качестве имени используются глаголы в настоящем времени или соответствующие причастия (передано, получено).

Каждое из внутренних действий записывается в виде отдельной строки и имеет формат: **< метка действия / выражение действия >**

Метка действия указывает на условия, при которых будет выполняться деятельность, определенная выражением действия. Значения меток действия в языке UML (не могут использоваться в качестве имен событий):

– ***entry*** указывает на действие, которое выполняется в момент входа в данное состояние;

– ***exit*** указывает на действие, которое выполняется в момент выхода из данного состояния;

– ***do*** специфицирует выполняющуюся деятельность (пока объект находится в данном состоянии);

– ***include*** используется для обращения к подавтомату, при этом следующее за ней выражение действия содержит имя этого подавтомата.

Действие, указанное после слова *Вход*, выполняется при входе в состояние, а действие, указанное после слова *Выход* – при выходе из него. Деятельность связывается с нахождением в состоянии. Во всех остальных случаях метка действия идентифицирует событие, которое запускает соответствующее выражение действия.

Изменение состояния, связанное с нарушением условия, или завершением деятельности, или наступлением события называют **переходом**. Отсюда ДС показывают состояния объекта, возможные переходы, а также события или сообщения, вызывающие каждый переход.

Переход обозначается линией со стрелкой и может быть помечен меткой, состоящей из частей, каждую из которых можно опустить: **< имя события >(< список параметров >)[< сторожевое условие>]< выражение действия >.**

Событие является самостоятельным элементом языка UML и представляет собой спецификацию некоторого факта, имеющего место в пространстве и во времени. В качестве событий можно рассматривать сигналы, вызовы, окончание фик-

сированных промежутков времени или моменты окончания выполнения определенных действий. **Имя события** идентифицирует каждый отдельный переход на ДС и может содержать строку текста, начинающуюся со строчной буквы.

Если событие не указано, то это означает, что переход выполняется по завершению деятельности, связанной с данным состоянием. Если же оно указано – то при наступлении события.

Сторожевое условие записывается в виде логического выражения. Переход происходит, если результат выражения – «истина». Объект не может одновременно перейти в два разных состояния, поэтому условия перехода для любого события должны быть взаимоисключающими.

Выражение действия выполняется в том и только в том случае, когда переход срабатывает. Представляет собой атомарную операцию (достаточно простое вычисление), выполняемую сразу после срабатывания соответствующего перехода до начала каких бы то ни было действий в целевом состоянии. Атомарность означает, что действие не может быть прервано никаким другим действием до тех пор, пока не закончится его выполнение.

При необходимости можно определять суперсостояния (рисунок 2.33, г), которые объединяют несколько состояний в одно. Этот механизм обычно используют, чтобы показать переход из нескольких состояний в одно и то же состояние, например, при отмене каких-либо действий.

ДС объекта строится, анализируя соответствующие диаграммы последовательностей. При этом необходимо уточнить, в какой момент разрешить прерывание процесса извне, чтобы показать, что прерывание процесса возможно еще во время его инициализации.

Пример. Разработать ДС фрагмента варианта использования *Инициализация процесса прогнозирования товара* от момента инициализации пользователем процесса решения до его завершения для объекта класса *Решение.*

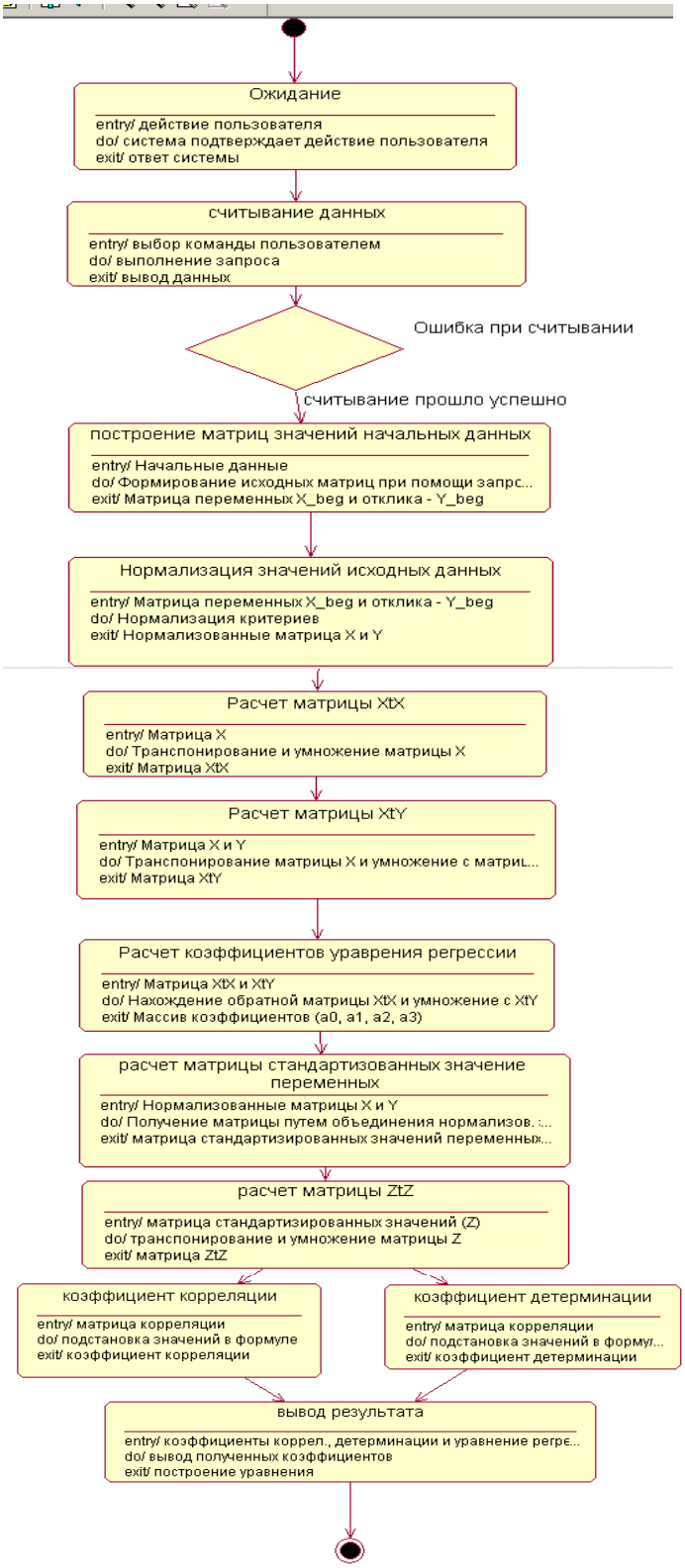


Рисунок 2.34 - Диаграмма состояний для объекта класса *Решение*

Таким образом, построение моделей поведения программного обеспечения позволяет детализировать объекты классов, т.е. перейти к их реализации

2.5.2 Методика реализации классов при использовании UML

Собственно реализация классов предполагает окончательное определение структуры и поведения его объектов. **Структура** объектов определяется совокупностью атрибутов и операций класса.

Каждый **атрибут** – поле или совокупность полей данных, содержащихся в объекте класса.

Поведение объектов класса определяется реализуемыми обязанностями. Обязанности выполняются посредством **операций** класса.

Таким образом, при проектировании класса, помимо имени и максимально полного списка атрибутов, необходимо уточнить его ответственность и операции. Причем как атрибуты, так и операции в процессе проектирования целесообразно дополнительно специфицировать.

В зависимости от степени детализации диаграммы классов обозначение **атрибута** может, помимо имени, включать: тип, описание видимости и значение по умолчанию. Для этого используют следующий формат: **<признак видимости> <имя>:<тип> = <значение по умолчанию>,** где признак видимости может принимать одно из трех значений: «+» – общий; «#» – защищенный; «-» – скрытый.

Имя атрибута представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах данного класса. Имя атрибута является единственным обязательным элементом синтаксического обозначения атрибута.

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме строки текста в квадратных скобках после имени соответствующего атрибута: **[нижняя_граница 1..верхняя_граница1, нижняя_граница2..верхняя_граница2, … ,нижняя_граница N..верхняя_граница N],** где «нижняя_граница» и «верхняя_граница» являются положительными целыми числами, каждая пара которых служит для обозначения отдельного замкнутого

интервала целых чисел. В качестве «верхней_границы» может использоваться специальный символ «*», который означает произвольное положительное целое число. Другими словами, это означает неограниченное сверху значение кратности соответствующего атрибута.

Тип атрибута представляет собой выражение, определяемое в зависимости от языка программирования, который предполагается использовать для реализации данной модели.

Исходное значение служит для задания некоторого начального значения для соответствующего атрибута в момент создания отдельного экземпляра класса. Здесь необходимо придерживаться правила принадлежности значения типу конкретного атрибута. Если исходное значение не указано, то значение соответствующего атрибута не определено на момент создания нового экземпляра класса.

Как упоминалось выше, **операциями** называют основные действия, реализуемые классом. В отличие от методов, операции не всегда реализуются классом непосредственно. Например, операция *Ввод числа* может быть реализована интерфейсным элементом «окно ввода».

Полное описание **операции** на диаграмме класса в UML выглядит следующим образом: **<признак видимости> <имя>(<список параметров>):<тип возвращаемого значения>.**

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной в пределах данного класса. Имя операции является единственным обязательным элементом синтаксического обозначения операции.

Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем виде: **<вид параметра><имя параметра>:<выражение типа>=<значение параметра по умолчанию>**

Вид параметра — есть одно из ключевых слов in, out или inout со значением in по умолчанию.

Имя параметра - это идентификатор соответствующего формального пара-

метра.

Выражение типа является зависимой от конкретного языка программирования спецификацией типа возвращаемого значения для соответствующего формального параметра.

Значение по умолчанию в общем случае представляет собой выражение для значения формального параметра, синтаксис которого зависит от конкретного языка программирования и подчиняется принятым в нем ограничениям.

Выражение типа возвращаемого значения также является зависимой от языка реализации спецификацией типа или типов значений параметров, которые возвращаются объектом после выполнения соответствующей операции. Двоеточие и выражение типа возвращаемого значения могут быть опущены, если операция не возвращает никакого значения. Для указания кратности возвращаемого значения данная спецификация может быть записана в виде списка отдельных выражений.

Ответственностью класса называют краткое неформальное перечисление основных функций объектов класса. Ответственность класса обычно определяют на начальных этапах проектирования, когда атрибуты и операции класса еще не определены.

Эту информацию отображают на диаграмме классов в специальных секциях нотации класса, представленного на рисунке 2.35.

Рисунок 2.35 – Полная нотация класса в UML

Исходный **список операций** класса формируют, анализируя диаграммы деятельностей, диаграммы взаимодействия и диаграммы последовательностей действий, построенные для различных сценариев с участием объектов проектируемого класса. На начальных этапах проектирования в секции операций класса обычно

указывают лишь имена основных операций, определяющих наиболее общее поведение объектов соответствующих классов. По мере уточнения добавляют новые операции, а информацию об уже имеющихся операциях детализируют.

Большинство **атрибутов** выявляется при анализе предметной области, требований технического задания и описаний потоков событий.

Кроме того, как указывалось выше, отношение ассоциации и его подвиды (агрегация и композиция) означают наличие обмена сообщениями между объектами классов. Для организации передачи сообщений необходимо, чтобы генерирующий сообщения объект содержал информацию о вызываемом объекте, что означает наличие у этого объекта соответствующего **указателя**. Причем, при отношении композиции объекты-части могут быть организованы как объектные поля объекта-целого.

Результаты уточнения структуры и поведения объектов классов позволяют завершить построение диаграммы классов этапа реализации.

На рисунке 2.36 показан вариант Диаграммы классов этапа реализации автоматизированной системы менеджера торгового предприятия.

Классы *Данные* и *Результаты*. Данные задач и их результаты должны храниться в базе данных, но они имеют различные структуры. Эту проблему можно решить, если хранить и данные, и результаты в упакованном виде, распаковывая их по мере надобности. Значит, соответствующие классы должны объявлять абстрактные операции *Упаковать()* и *Распаковывать(),* которые будут реализовываться классами-подтипами в зависимости от реальной структуры данных, определяемой типом задачи.

Класс *Алгоритм*. Объекты класса *Алгоритм* отвечают за реализацию метода решения задачи. Поскольку они посылают сообщение объектам класса *Задания*, то, естественно, должны хранить его адрес. Кроме того, класс *Алгоритм* должен объявлять абстрактную операцию *Выполнить().* Эта операция должна переопределяться классами *Алгоритм*, реализующий метод.

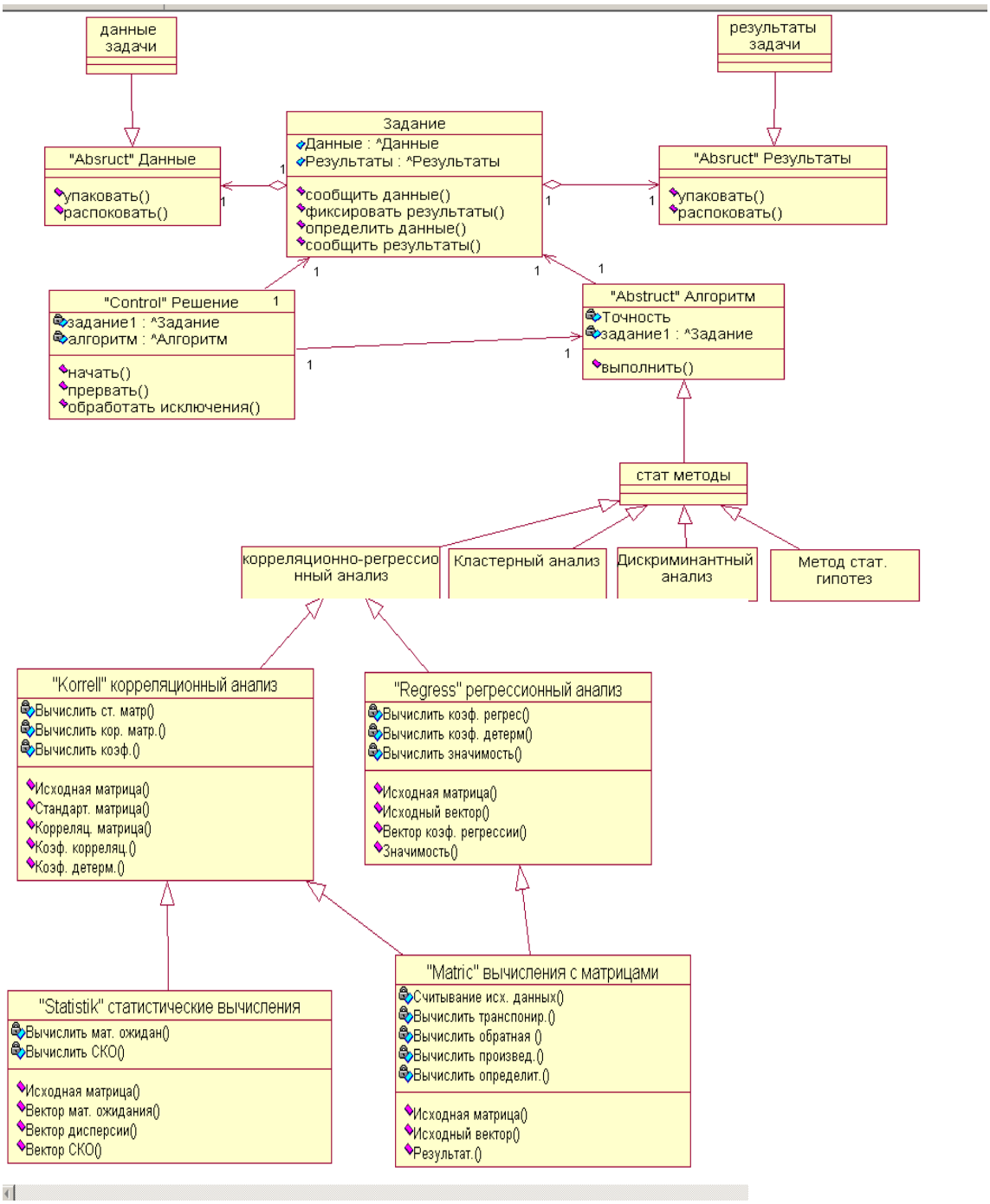


Рисунок 2.36 – Диаграмма классов автоматизированной системы менеджера торгового предприятия

Класс *Решение.* Объект класса *Решение* обращается к объектам классов *Задание* и *Алгоритм*, следовательно, необходимо хранить их адреса. Операции *Начать()* и *Прервать()* получены из диаграмм последовательностей. Операция *Обработать исключение()* должна реализовываться особым образом, так как будет по-

лучать управление через механизм исключений.

Таким образом, полностью реализованные классы обеспечивают последующую автоматическую генерацию программного кода на конкретном, указанном разработчиком, языке программирования.

2.5.3 Вопросы и задания для самоконтроля

1 Какой формат записи используется для обозначения атрибута диаграммы классов?

2 Какое значение может принимать признак видимости?

3 Какой формат записи используется для обозначения операции диаграммы классов?

4 Что показывают диаграммы состояний?

5 Что называют переходом из одного состояния в другое?

6 Какой формат записи используется для обозначения атрибута диаграммы классов?

7 Какое значение может принимать признак видимости?

8 Какой формат записи используется для обозначения операции диаграммы классов?

9 Что показывают диаграммы состояний?

10 Что называют переходом из одного состояния в другое?

2.6 Физическое представление архитектуры программного обеспечения при использовании UML

Ранее разработанные диаграммы отражали концептуальные аспекты построения моделей ПО системы, т.е. оперировали понятиями логического уровня

представления, которые не имеют самостоятельного материального воплощения. Для описания реальных сущностей предназначен другой аспект модельного представления, а именно физическое представление модели ПО. Для физического представления модели используются **модели реализации**, которые включают в себя две отдельные канонические диаграммы: диаграмму компонентов (ДКом) и диаграмму развертывания (размещения).

2.6.1 Компоновка программных компонентов

Диаграмма компонентов (**COMPONENT DIAGRAM)** является визуальной моделью ПО на физическом уровне.

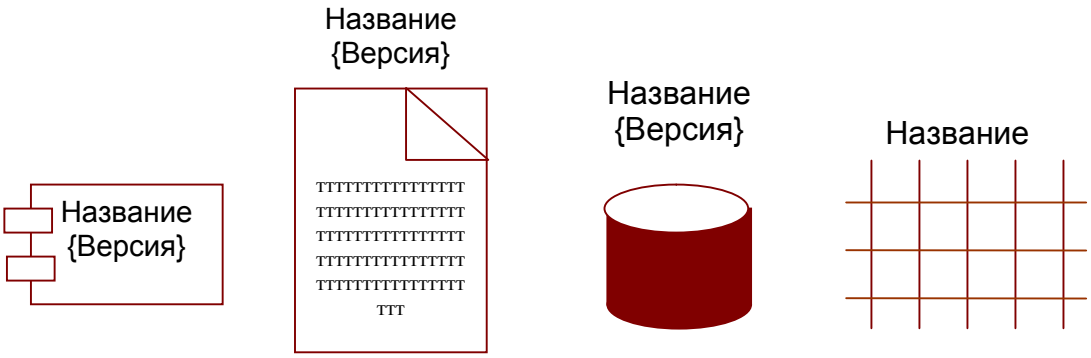
ДКом описывает объекты реального мира – компоненты ПО и позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код.

ДКом оперируют понятиями компонент и зависимость.

Под **компонентами** понимают физические заменяемые части ПО, которые соответствуют некоторому набору интерфейсов и обеспечивают их реализацию. По сути дела, это отдельные файлы различных типов: исполняемые (.exe), текстовые, графические, таблицы баз данных и т. п.

Зависимость между компонентами фиксируют, если один компонент содержит некоторый ресурс (модуль, объект, класс и т. д.), а другой его использует. На ДКом зависимость обозначают пунктиром со стрелкой на конце.

Нотации ДКом приведены на рисунке 2.37.



а – программный компонент; б – файл; в – база данных; г – таблица базы данных

Рисунок 2.37 – Нотации компонентов в UML

На рисунке 2.38 показана ДКом программной системы с интерфейсами.

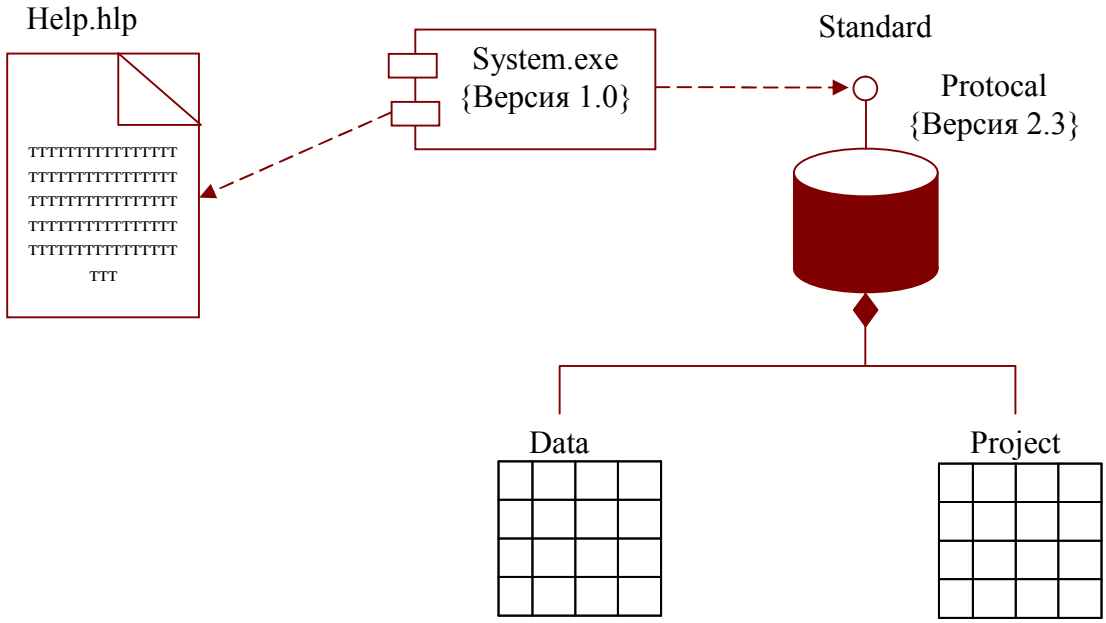


Рисунок 2.38 – ДКом автоматизированной системы менеджера

На ДКом целесообразно уточнять зависимость между компонентами, используя отношения обобщения, ассоциации, композиции или агрегации. На рисунке 2.38 показано, что база данных включает (отношение композиции) две таблицы.

Качество компоновки оценивают по степени независимости компонентов, т. е. по количеству и типу связей между компонентами.

Используя нотации UML, можно построить ДКом практически для любого предметной области. На рисунке 2.39 приведен пример ДКом клиентской части Интернет-приложения, написанного с использованием Java, которое в процессе работы демонстрирует некоторый рисунок и текст.

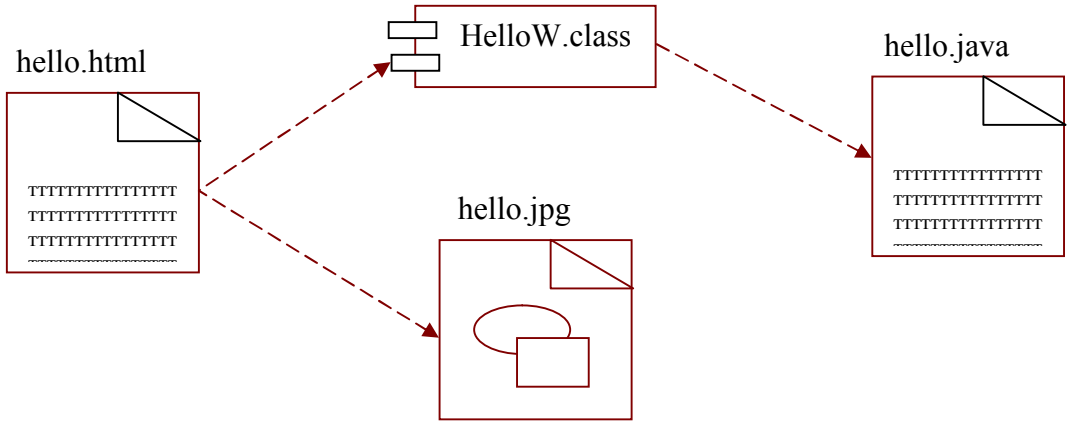


Рисунок 2.39 – ДКом Internet-приложения, которая выводит фотографию и текст

При «сборке» исполняемых файлов ДКом применяют для отображения взаимосвязей файлов, содержащих исходный код. На рисунке 2.40 показано, что основной файл Main.cpp зависит от заголовочного файла Model.h, реализация которого находится в файле Model.cpp.

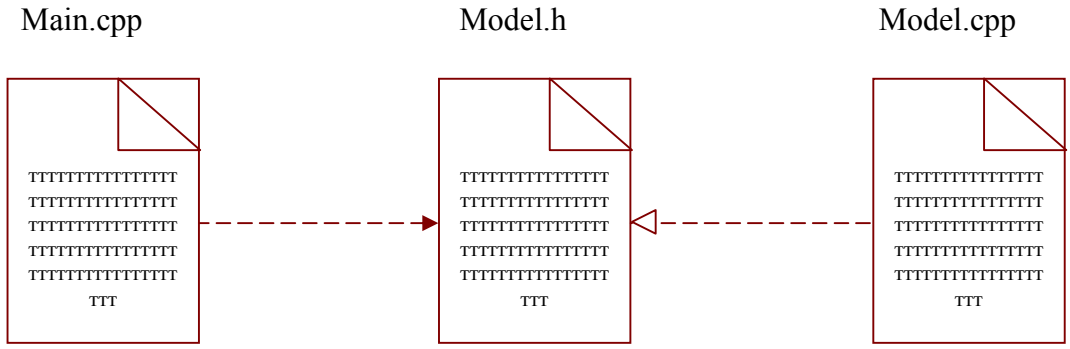


Рисунок 2.40 – ДКом исполняемого файла C++

Таким образом, ДКом для ПО «клиент-сервер» отражает архитектуру разрабатываемого системы, так как позволяет показать связи по управлению частей системы (компонентов).

2.6.2 Размещение программных компонентов для распределенных программных систем

Физическое представление ПО не может быть полным, если отсутствует информация о том, на какой платформе и на каких вычислительных средствах программа реализована. Если разрабатывается программная система, которая может выполняться локально на компьютере пользователя, без использования периферийных устройств и ресурсов, то в этом случае нет необходимости в разработке дополнительных диаграмм.

Однако при разработке корпоративных систем имеют место особенности.

Во-первых, ПО корпоративных систем реализуется в сетевом варианте на различных вычислительных платформах и технологиях доступа к распределенным базам данных. Наличие локальной корпоративной сети требует решения целого комплекса дополнительных задач по рациональному размещению компонентов ПО по узлам сети с целью обеспечения требуемой производительности.

Во-вторых, интеграция корпоративной системы с Интернетом определяет необходимость решения дополнительных задач при разработке ПО, таких как обеспечение информационной безопасности и устойчивости доступа к информации для корпоративных клиентов. Эти задачи зависят от реализации проекта в форме физически существующих узлов системы, таких как серверы, рабочие станции, брандмауэры, каналы связи и хранилища данных.

В-третьих, технологии доступа и манипулирования данными в рамках общей архитектуры «клиент-сервер» требуют размещения баз данных в различных сегментах корпоративной сети, их резервного копирования, архивирования, кэширования для обеспечения необходимой производительности системы в целом. Эти аспекты также требуют визуального представления с целью спецификации программных и технологических особенностей реализации распределенных архитектур.

Такой формой физического представления программной системы является диаграмма развертывания (синоним — диаграмма размещения).

Диаграмма развертывания **(Deployment diagram)** является визуальной моделью ПО, которая отражает физические взаимосвязи между программными и аппаратными компонентами системы.

Целями разработки диаграмм развертывания-размещения (ДРР) являются:

- распределение компонентов ПО по физическим узлам;
- графическое представление физических связей между всеми узлами реализации ПО на этапе исполнения;
- выявление узких мест системы и реконфигурация ее топологии для достижения требуемой производительности.

Для достижения этих целей ДРР разрабатывается совместно системными аналитиками, сетевыми инженерами (системными программистами) и системотехниками.

Нотации ДРР показаны на рисунке 2.41

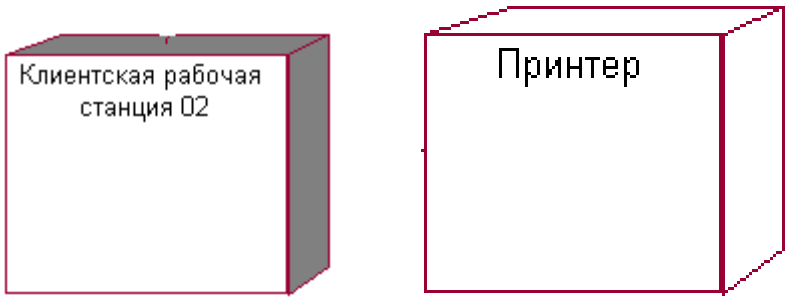


Рисунок 2.41 – Нотации ДРР

Каждой части аппаратных средств системы, например, компьютеру или серверу, на ДРР соответствует *узел. Соединения узлов* означают наличие в системе соответствующих коммуникационных каналов. Внутри узлов указывают размещенные на данном оборудовании программные компоненты разрабатываемого ПО, сохраняя указанные на ДКом отношения зависимости. С точки зрения ДРР локальная и глобальная сети – это тоже узлы, которые обладают некоторой спецификой.

Пример. Разработать ДРР для локальной сети автоматизированной системы торговой организации.

Локальная сеть торгового предприятия связывает сервер и компьютеры руководителя организации, менеджеров и сотрудников, отвечающих за занесение информации в базу данных. Серверную часть системы и базу данных целесообразно поместить на сервер. На компьютерах локальной сети в этом случае будут функционировать соответствующие клиентские части приложений.

На рисунке 2.42 изображена ДРР локальной сети торговой организации.

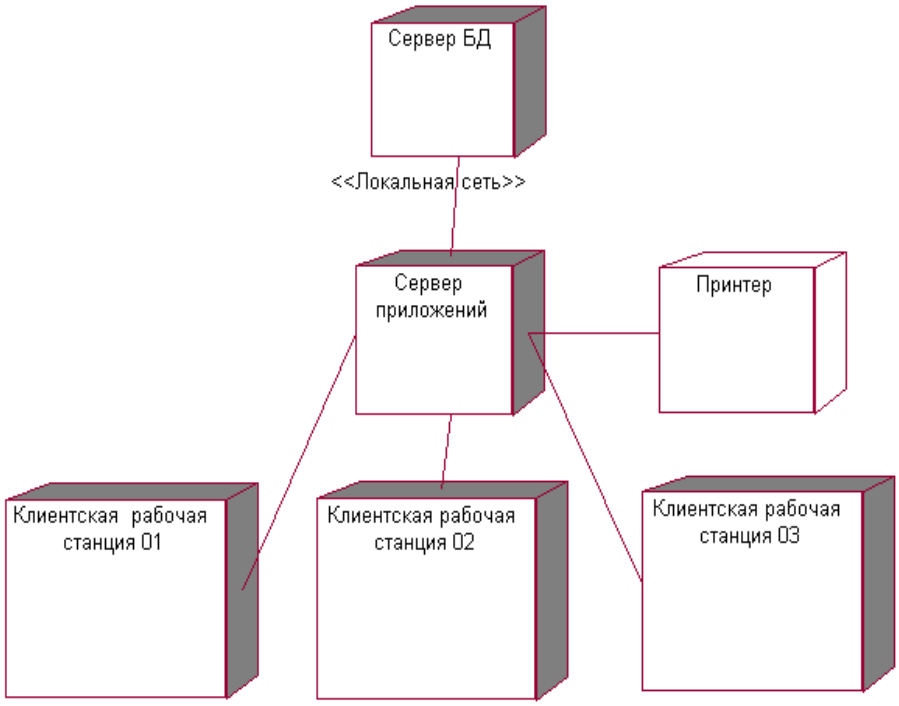


Рисунок 2.42 – ДРР локальной сети АИС торговой организации

Таким образом, диаграмма размещения отражает архитектуру распределенных программных систем на физическом уровне.

2.6.3 Методика генерации программного кода

Одним из главных достоинств Rational Rose является возможность генерации программного кода. Возможности генерации программ определяются версией Rose:

- Rose Modeler – позволяет создавать модель системы, но не поддерживает генерацию программного кода и обратное проектирование.

- Rose Professional – позволяет генерировать программный код на одном языке (определяется версией).

- Rose Enterprise – позволяет генерировать программный код на языках: Ada 83, Ada 95, ANSI C++, CORBA, Java, COM, Visual Basic, Visual C++, C++ и XML. Кроме того, поддерживается генерация кода и обратное проектирование баз данных.

На рисунке 2.43 показана технология генерации программного кода.

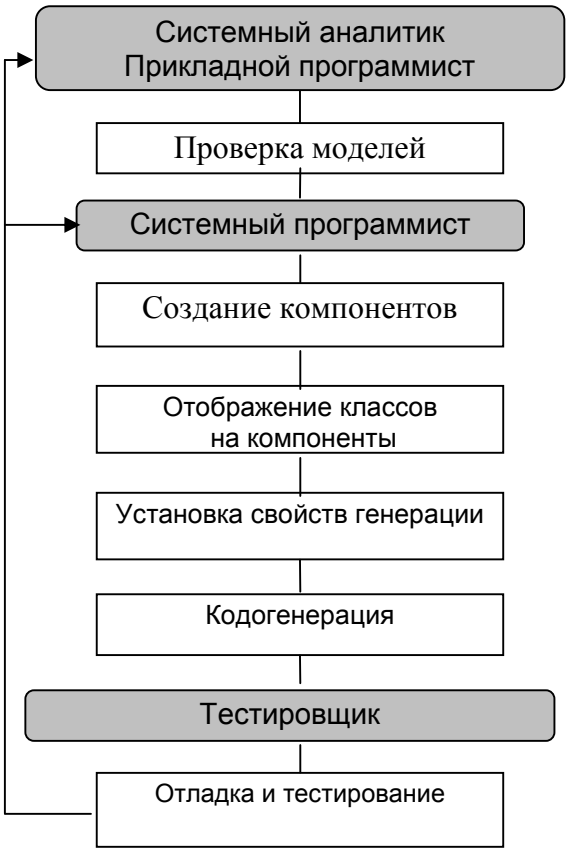


Рисунок 2.43 – Технология генерации программного кода

Обязательность этапов определяется языком генерируемого кода. Генерировать код программ на любом языке можно, не выполняя проверки модели, однако во время генерации это может приводить к ошибкам.

Первый этап технологии генерации программного кода - ***проверка модели*** в RRose осуществляется не зависящим от языка средством проверки моделей в ви-

де диаграммы классов (Check Model) и применяется для обеспечения корректности моделей перед генерацией программного кода.

В общем случае проверка моделей может выполняться на любом этапе работы над программным проектом. Однако, на этапа реализации диаграмма классов проверяется обязательно.

Прежде чем приступить к собственно генерации программного кода разработчику следует добиться устранения всех ошибок и предупреждений, о чем свидетельствует окно журнала событий (рисунок 2.44).

Рисунок 2.44 – Журнал событий при отсутствии ошибок по результатам проверки модели

К числу возможных ошибок относятся, например, не используемые ассоциации; классы, оставшиеся после удаления отдельных графических элементов в диаграммах; операции, не являющиеся именами сообщений на диаграммах поведения. Наиболее распространенные ошибки: сообщения на диаграмме последовательности или диаграмме кооперации, не отображенные на операцию; объекты этих диаграмм, не отображенные на класс.

Второй этап технологии генерации программного кода - **создание компонентов классов**, при котором на основе диаграммы классов (логическая модель) строится диаграмма компонентов (физическая реализация). Существуют компоненты самых разных типов: файлы исходного программного кода, исполняемые файлы, библиотеки исполнения в реальном времени, компоненты ActiveX, апплеты и т.д.

На рисунке 2.45 в качестве примера приведена диаграмма компонентов автоматизированной информационной системы менеджера.

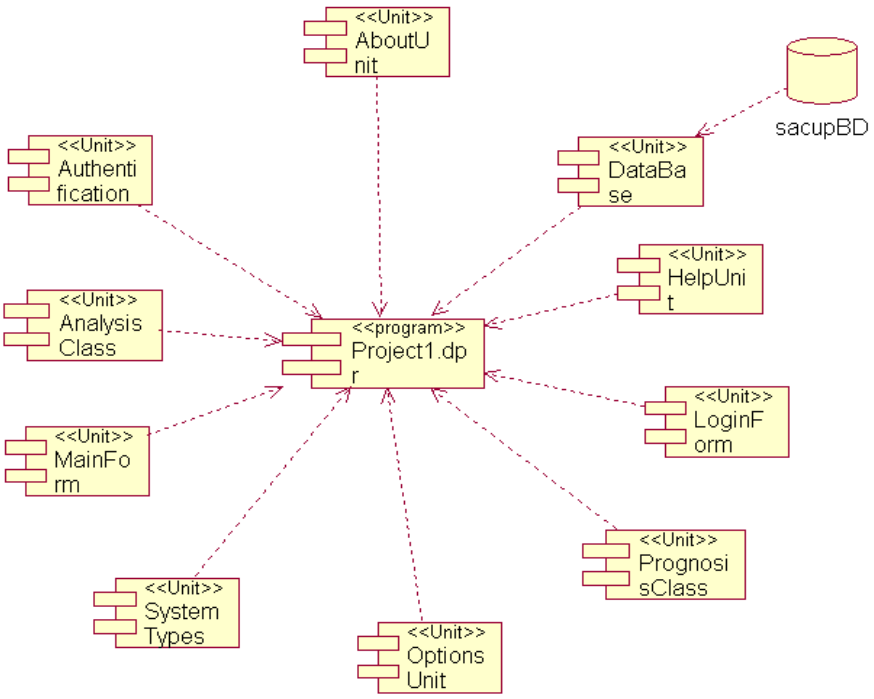


Рисунок 2.45 – ДКом АИС менеджера торгового предприятия

При генерации программ на Java или Visual Basic отображать компоненты не обязательно, так как Rational Rose в этом случае создает компоненты для каждого из классов автоматически.

Третий этап генерации - ***отображение классов на компоненты*** реализуется через окно спецификации, представленное на рисунке 2.46.

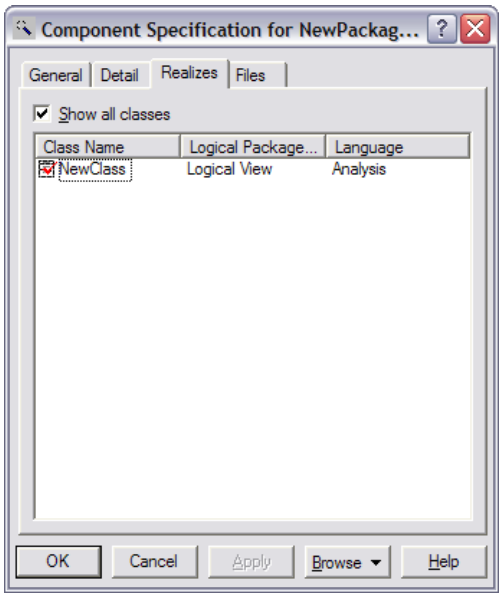


Рисунок 2.46 – Окно спецификации компонента

Каждый компонент исходного кода - это файл программного кода для одного или нескольких классов. В C++ каждый класс отображается на два компонента с исходным кодом: файл заголовка и основной файл (тело). В C++ PowerBuilder на один компонент может отображатся несколько классов. Компонентом с исходным программным кодом в PowerBuilder является файл библиотеки PowerBuilder (.pbl). Компоненты создаются и для элементов управления ActiveX, апплетов, DDL, исполняемых файлов, а также других исходных и скомпилированных файлов.

Установка свойств генерации программного кода. На рисунке 2.47 показано окно свойств генерации кода на C++.

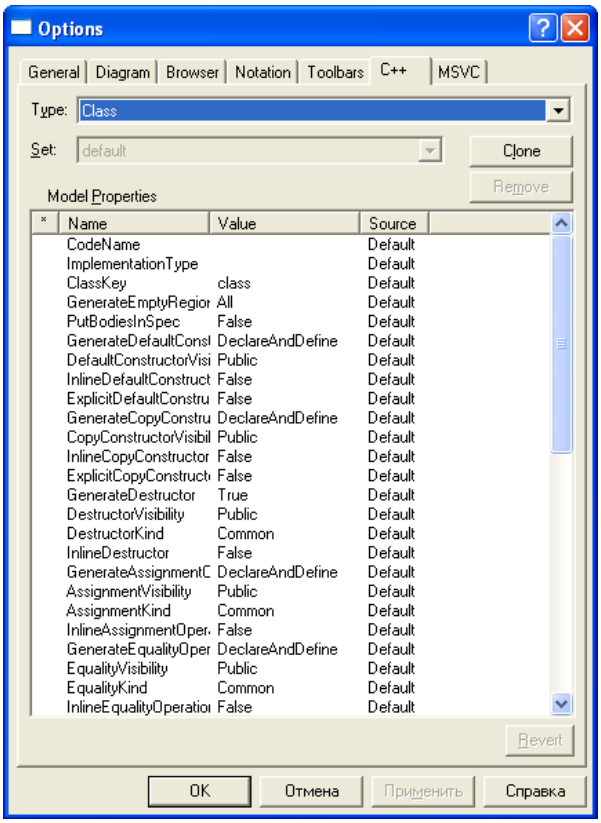


Рисунок 2.47 – Окно свойств генерации кода на C++

Для каждого языка в Rational Rose предусмотрен ряд определенных свойств генерации программного кода. Свойства определяются параметрами генерации программного кода для компонентов, классов, атрибутов и других элементов моделей. Rational Rose предлагают общепринятые параметры свойств генерации по умолчанию.

Собственно генерация программного кода реализуется через окно спецификации, представленное на рисунке 2.48.

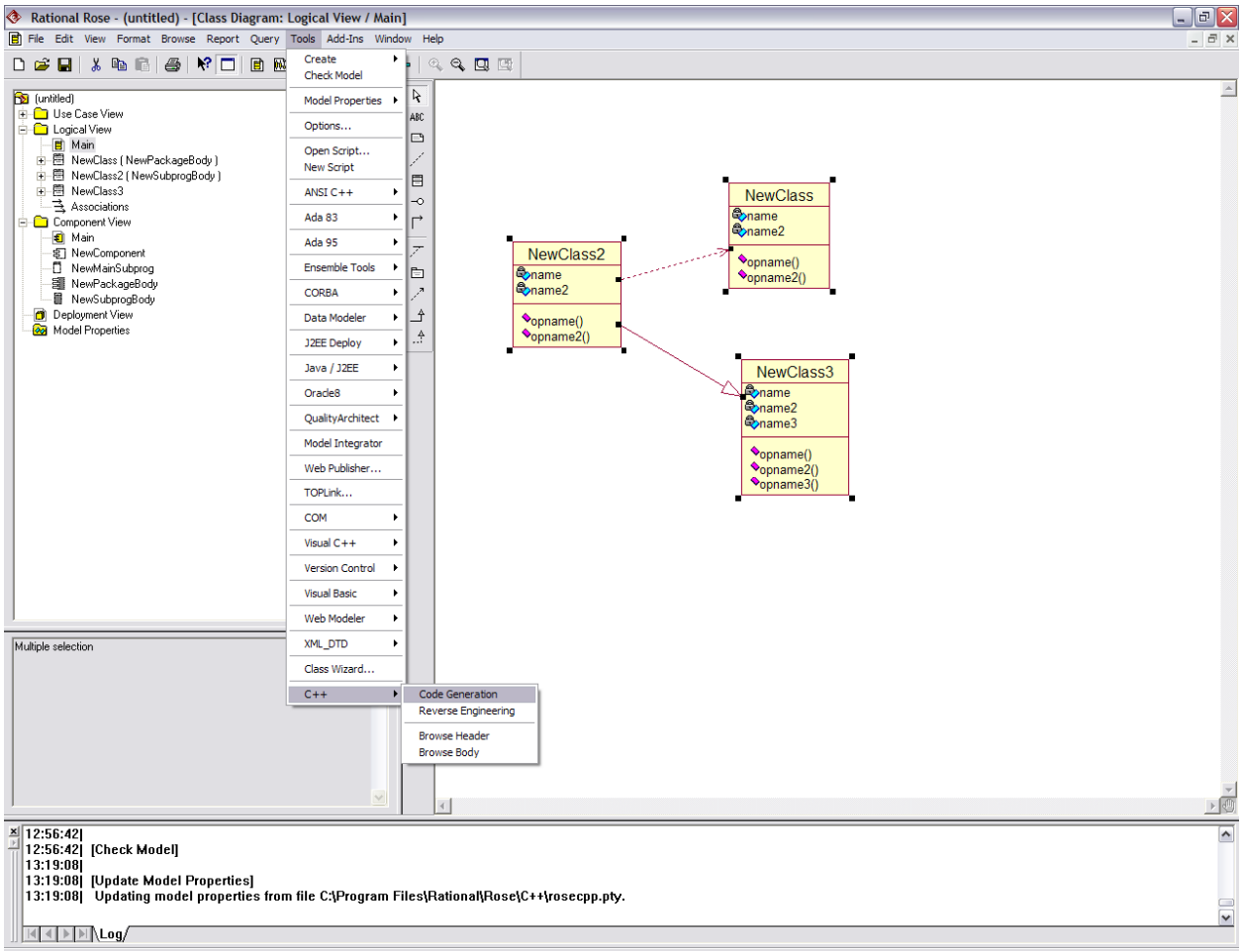


Рисунок 2.48 – Меню генерации кода

В диалоговом окне Add-In Manager, изображенном на рисунке 2.49, можно установить язык генерации.

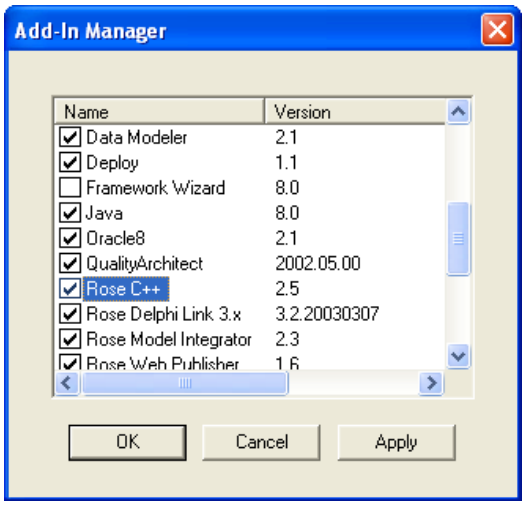


Рисунок 2.49 – Менеджер надстроек Add-Ins

Для генерации кода нужный элемент модели выделяется в браузере проекта и выполняется операция контекстного меню: Tools→C++→Code Generation (Язык C++→Генерировать код). В результате откроется диалоговое окно, изображенное на рисунке 2.50, с предложением выбора классов для кодогенерации на указанном языке программирования.

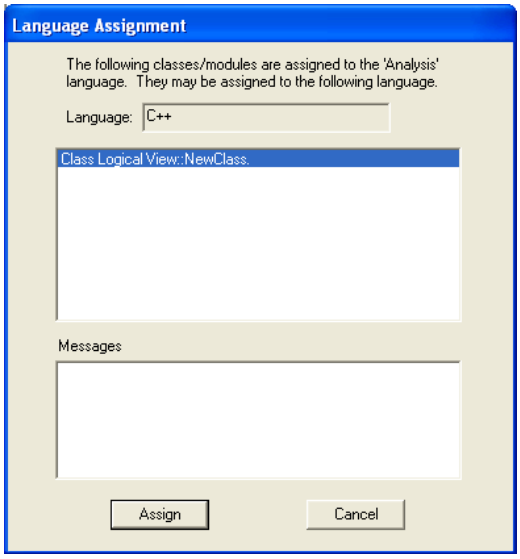


Рисунок 2.50 - Окно выбора классов для генерации программного кода

После кодогенерации происходит компиляция и в окне статуса (Code Generation Status) отображается информация о том, какой класс закодирован и количество ошибок и предупреждений (рисунок 2.51).

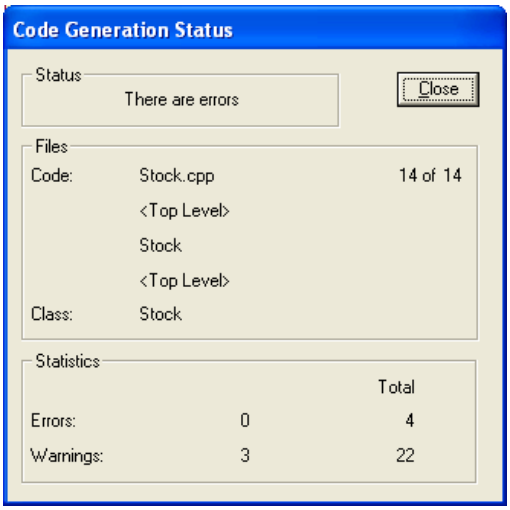


Рисунок 2.51 – Окно статуса компиляции

В результате кодогенерации Rational Rose создает два файла с расширением «.h» и «.cpp» (наименования то же, что и название класса).

Ошибки или предупреждения в процессе кодогенерации отображается в соответствующем окне журнала событий.

Для включения дополнительных элементов в программный код следует изменить свойства генерации программного кода, установленные по умолчанию.

Завершающим этапом разработки ПО является отладка и тестирование разработанных программных средств, которые выполняет специалист-тестировщик.

Отладка – это деятельность, направленная на обнаружение и исправление ошибок в ПО. *Тестирование* – это процесс выполнения программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Указанный набор данных называется тестовым или просто тестом. Следовательно, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПО ошибки, поиска места ошибки в программах и редактирование программ с целью устранения обнаруженной ошибки.

Таким образом, Case – средства являются эффективным инструментом сокращения сроков и затрат ресурсов при разработке *масштабных проектов* в составе команды или проектной группы.

2.6.4 Вопросы и задания для самоконтроля

1 Что показывают диаграммы компонентов?

2 Что понимают под компонентами?

3 В каких случаях целесообразно строить диаграммы компонентов?

4 В каком случае зависимость между компонентами фиксируют?

5 Какая диаграмма отображает физические взаимосвязи между программными и аппаратными компонентами системы?

6 Что означает соединение узлов ?

7 Чем являются локальная и глобальная сеть с точки зрения диаграммы размещения?

8 В каком случае необходима реорганизация программы?

9 Из каких основных этапов состоит процесс генерации кода?

10 В чем заключается процесс *создание компонентов?*

11 Что понимается под отладкой ПС?

12 Дайте определение понятию тестирование ПС.

3 Case-средства разработки программного обеспечения

3.1 Построение функциональной модели предметной области в среде Rational Rose Enterprise Edition

Любая программная система обладает множеством вариантов использования и множеством лиц, пользователей системы. Функциональность системы описывает диаграмма вариантов использования (ДВИ) на некотором уровне абстракции. Абстракция (abstraction) – сосредоточение на важнейших аспектах приложения и игнорирование всех остальных. Использование абстракции позволяет сохранить свободу принятия решений разработчика до этапа проектирования.

3.1.1 Инструментарий разработки диаграмм вариантов использования в среде Rational Rose Enterprise Edition 2003

Среда Rational Rose Enterprise Edition 2003(далее Rational Rose) предусматривает для ДВИ систему нотаций, представленную на рисунке 3.1.

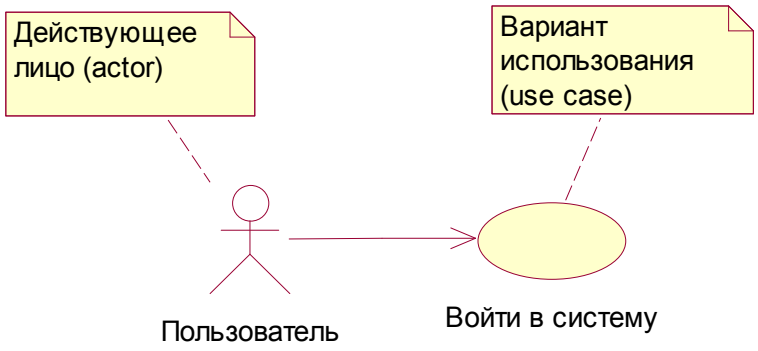


Рисунок 3.1 – Нотации диаграммы вариантов использования

Вариант использования объединяет поведение, имеющее отношение к элементу функциональности системы: нормальное поведение, вариации нормального пове-

дения, исключительные ситуации, сбойные ситуации и отмены запросов. Вариант использования должен иметь описание, объясняющее действия в этом варианте. Описание должно содержать сведения о типах пользователей, выполняющих данный вариант использования, и ожидаемый результат. Документирование варианта использования в среде Rational Rose осуществляется при помощи окна спецификаций в поле для документации (рисунок 3.2).

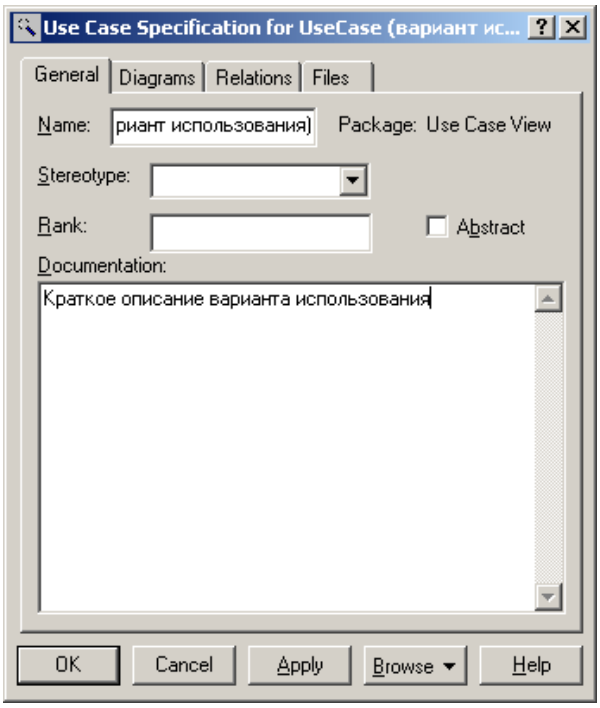


Рисунок 3.2 – Окно документирования варианта использования

Имеется несколько стандартных типов отношений между действующими лицами и вариантами использования.

Ассоциация – структурное отношение, описывающее совокупность связей, представленных соединениями между объектами модели.

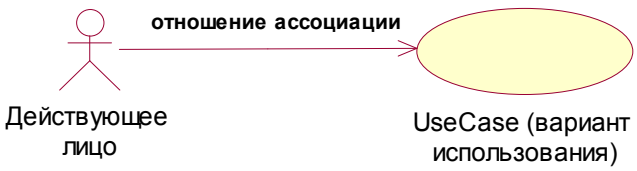


Рисунок 3.3 – Реализации отношения ассоциации

Зависимость – отношение между двумя сущностями, при которой изменение одной из сущностей (независимой), может повлиять на семантику второй сущности

(зависимой). Графически данное отношение обозначается пунктирной линией со стрелкой, направленной от того варианта использования, который является расширением для исходного варианта использования.

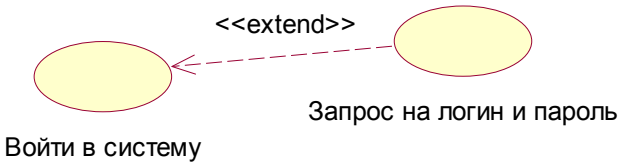


Рисунок 3.4 - Отношения зависимости между вариантами использования

Обобщение – отношение, при котором объект специализированного элемента может быть подставлен и использован вместо объекта обобщенного элемента. При этом потомок наследует все свойства и поведение своего родителя и может быть дополнен новыми свойствами поведения. Графическое отображение направления стрелки для данного отношения указывает на родительский вариант использования (рисунок 3.5).

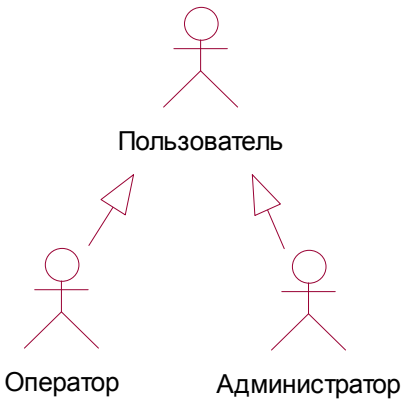


Рисунок 3.5 – Отношения обобщения между действующими лицами

Rational Rose предоставляет несколько способов создания новых элементов в окне диаграмм *Use Case*:

1 Используя контекстное меню папки Use Case View (New→Use Case).

2 Используя главное менюTools (Create →Use Case).

Для моделирования бизнес-процессов Rational Rose предоставляет дополнительные элементы *Use Case*, которые можно активизировать при помощи режима настройки инструментов. Но для создания рассматриваемой автоматизированной системы достаточно установленных в панелях инструментов по умолчанию.

Панель инструментов рабочего окна диаграммы Use Case приведена в таблице 3.1.

Таблица 3.1 - Пиктограммы панели инструментов диаграммы Use Case

Пиктограмма	Кнопка	Описание
	Selects or deselects an item (Выделение или отмена выделения объекта)	Превращает курсор в стрелку указателя для выделения объекта
	Text Box (Текст)	Добавляет к диаграмме текста
	Note (Примечание)	Добавляет к диаграмме примечание
	Anchor Note to Item (Прикрепление примечания к объекту)	Связывает примечание с вариантом использования или объектом на диаграмме
	Package (Пакет)	Помещает на диаграмму новый пакет
	Use Case (Вариант использования)	Помещает на диаграмму новый вариант использования
	Actor (Действующее лицо	Помещает на диаграмму новое действующее лицо
	Unidirectional Association (Однонаправленная ассоциация)	Рисует связь между действующим лицом и вариантом использования
	Dependency or Instantiates (Зависимость или наполнение)	Рисует зависимость между элементами диаграммы
	Generalization (Обобщение)	Рисует связь использования или расширения между вариантами использования, или рисует связь наследования между действующими лицами

Таким образом, инструментарий среды Rational Rose Enterprise Edition 2003 позволяет описать модель функциональности разрабатываемой автоматизированной системы в виде диаграммы вариантов использования.

3.1.2 Построение диаграммы вариантов использования в среде Rational Rose Enterprise Edition 2003

В качестве примера построения диаграммы вариантов использования в среде Rational Rose Enterprise Edition 2003 используется следующая предметная область автоматизации информационных процессов.

Описание предметной области: организация предоставляет услуги по трудоустройству, ведется банк данных о существующих вакансиях. По каждой вакансии поддерживается определенная информация. Необходимо спроектировать базу данных АИС и разработать аналитическое приложение.

Пользователь: менеджер по работе с кадрами.

Цель приложения: автоматизация информационного процесса определения подходящей вакансии по данным резюме (анкеты) соискателя.

Метод аналитической обработки: дискриминантный анализ.

На рисунке 3.6 приведена диаграмма вариантов использования для приложения АИС «Трудоустройство».

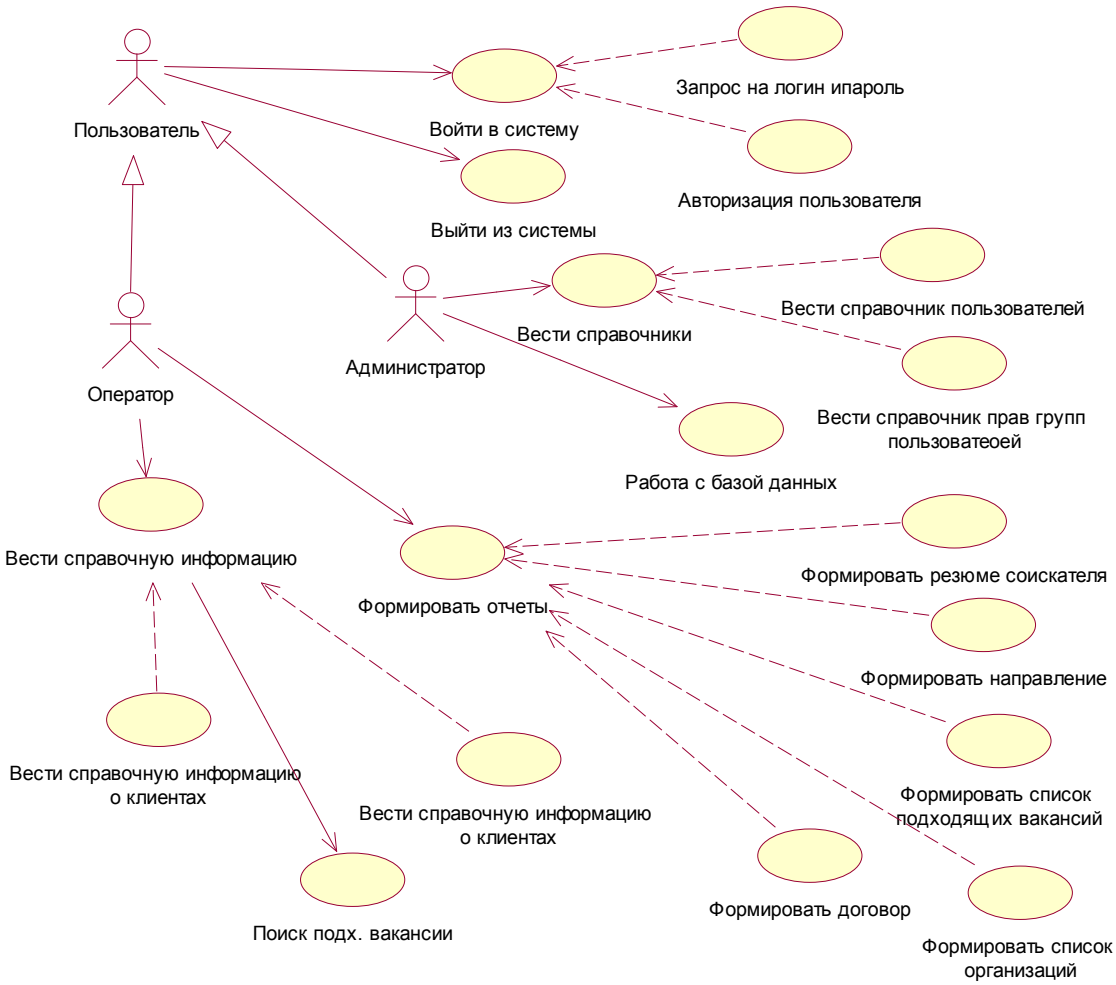


Рисунок 3.6 – ДВИ менеджера по работе с кадрами

Таким образом, использование инструментария пакета Rational Rose Enterprise Edition 2003 позволяет описать функциональность разрабатываемой автоматизированной системы.

3.1.3 Вопросы и задания для самоконтроля

1 Что представляет собой пакет Rational Rose?

2 Какие преимущества дает применение Rational Rose при разработке программных систем?

3 Какие UML диаграммы доступны в Rational Rose?

4 Для чего используется диаграмма Use Case?

5 Как создать новую диаграмму?

6 Какие значки находятся в строке инструментов диаграммы Use Case и каково их назначение?

7 Как настроить панель инструментов для диаграмм в Rational Rose?

8 Какие типы связи существуют между элементами диаграммы вариантов использования?

9 Какие значки специфичны только для диаграммы Use Case?

10 Как при помощи диаграммы создать сценарий поведения?

3.2 Построение концептуальной модели предметной области в среде Rational Rose Enterprise Edition 2003

Концептуальную модель предметной области, являющейся диаграммой классов этапа анализа жизненного цикла программной системы, можно представить в виде ER-диаграммы с отношениями «сущность-связь». Для этого необходимо разработать модель данных. Моделирование данных является важнейшим процессом разработки ПО. Поэтому разработчики CASE-средств в своих продуктах уделяют моделированию данных особое внимание.

Являясь признанным лидером в области объектных методологий, фирма Rational Software Corporation имеет собственное средство моделирования данных, позволяющее строить физические модели для конкретных СУБД. Data Modeler является инструментом генерации программного кода структуры данных. Rational Rose позволяет автоматически осуществлять переход от логической модели к физической (возможен обратный процесс). Для этого введено соответствие элементов моделей, представленное в таблице 3.2.

Таблица 3.2 – Элементы логического и физического описаний

Логическая модель	Физическая модель
Class (Класс)	Table (Таблица)
Operation (Операция)	Constraint (Ограничение)
Attribute (Атрибут)	Column (Колонка)
Package (Пакет)	Scheme (Схема)
Component (Компонент)	Database (База данных)
Association (Ассоциация)	Relationship (Связь)
Нет	Trigger (Тригер)
Нет	Index (Индекс)

3.2.1 Инструментарий разработки модели данных в среде Rose Data Modeler

После установки Rational Rose в редакции Rational Rose Professional Data Modeler Edition в разделе главного меню Tools появляется раздел Data Modeler (рисунок 3.7).

В разделе Data Modeler имеются два пункта: «Add Schema» и «Reverse Engeneer…». Первый из них используется для создания новых схем БД, а второй - для построения модели на основе существующей схемы БД.

Создать модель данных можно следующим способом:

1) в браузере вызвать контекстное меню схемы используя правую кнопкой;

2) в появившемся меню выбрать Data Modeler→New→Data Model Diagram;

3) ввести имя новой диаграммы;

4) открыть созданную диаграмму(двойным нажатием мыши).

Диаграмма модели данных имеет специализированную панель инструментов, представленную в таблице 3.3, для добавления таблиц, отношений и других элементов моделирования данных.

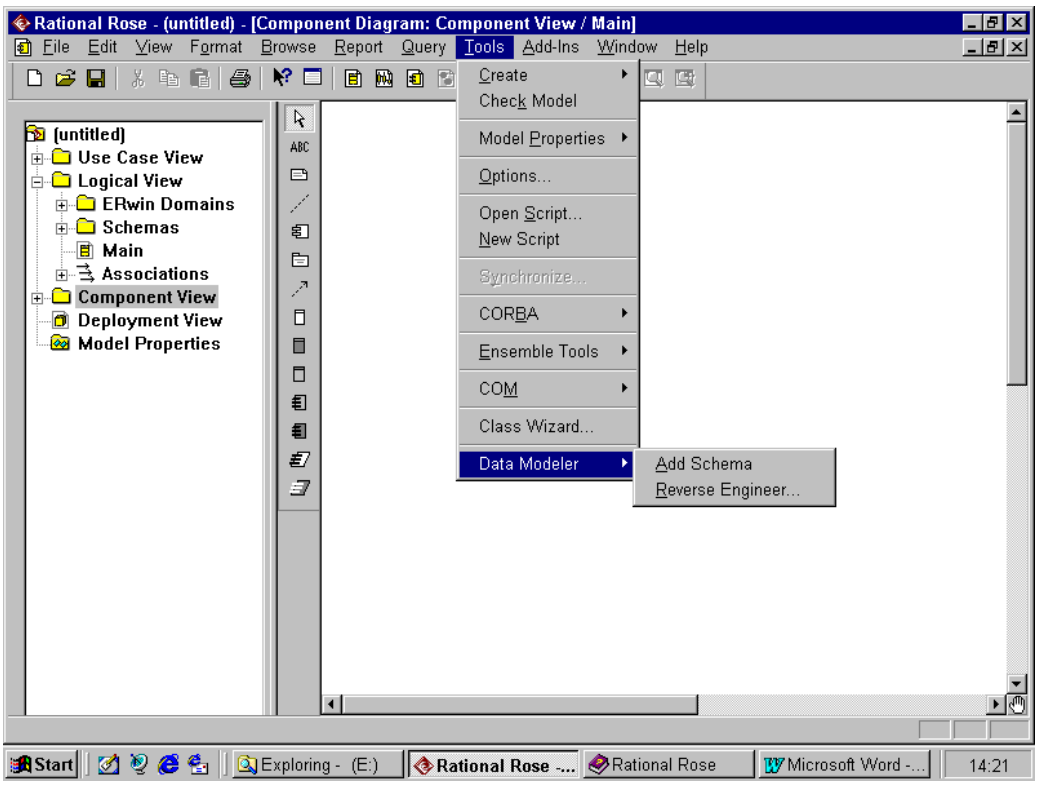


Рисунок 3.7- Отображение компоненты Data Modeler в меню Rational Rose

Таблица 3.3 – Панель инструментов моделирования данных

Значки	Назначение
	Курсор принимает форму стрелки для выделения элемента
	Добавляет в диаграмму текстовое поле
	Добавляет к элементу диаграммы примечание
	Соединяет примечание с элементом диаграммы
	Добавляет в диаграмму таблицу
	Рисует неидентифицируемое отношение между двумя таблицами
	Рисует идентифицируемое отношение между двумя таблицами
	Добавляет в диаграмму представление
	Рисует зависимость между двумя таблицами

Инструментарий Data Modeler предоставляет следующие возможности:

- создание и редактирование таблиц и их элементов (колонок, ограничений, индексов, триггеров и т. п.);

- создание и редактирование идентифицирующих связей между таблицами;

- создание и редактирование неидентифицирующих связей.

Для этого используются настройки окна спецификаций, представленного на рисунке 3.8

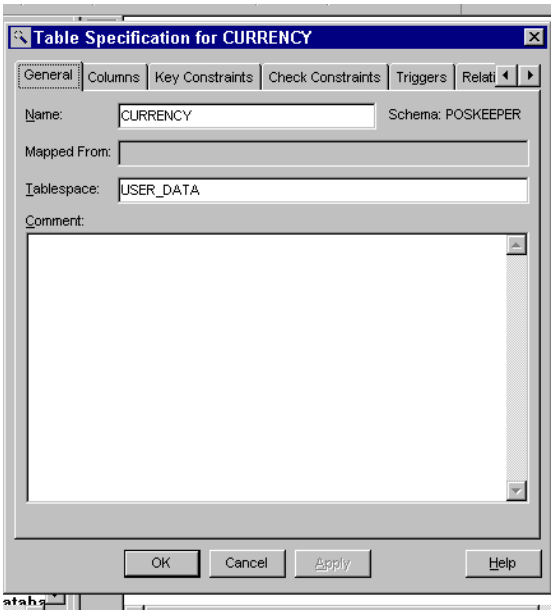


Рисунок 3.8 - Окно спецификации таблицы

Спецификации возможностей редактирования таблиц БД представлены в таблице 3.4.

Отношения в модели данных подобны отношениям в объектной модели. В объектной модели отношение связывает два класса, а в модели данных — две таблицы. В Rational Rose поддерживаются два основных типа отношений: идентифицируемые отношения (identifying relationship) и неидентифицируемые отношения (non-identifying relationship).

В обоих случаях для поддержки отношений в дочернюю таблицу добавляется внешний ключ. При идентифицируемом отношении внешний ключ становится частью первичного ключа в дочерней таблице. В этом случае дочерняя таблица не может содержать запись, не связанную с записью в родительской таблице.

Таблица 3.4 - Спецификации таблиц БД

Закладка	Описание
General	Вводится общая информация о таблице.
Columns	Задается описание колонок. Здесь можно добавить или отредактировать свойства колонок, задать тип, длину, обязательность (NULL, NOT NULL), а также пометить, что колонка входит в состав первичного ключа. Типы колонок соответствуют типам конкретной выбранной СУБД.
Key Constraints	Задаются ограничения на колонки таблицы. Здесь можно задать ограничение на уникальность первичного ключа, ограничение на уникальность альтернативных ключей, а также просто определить индекс.
Check Constraints	Задаются выражения – инварианты, которые должны выполнятся для всех строк таблицы.
Triggers	Содержит список триггеров, который можно отредактировать, в том числе добавив новый триггер.
Relationships	При наличии связей между таблицами, закладка содержит полный список связей.

Неидентифицируемые отношения тоже создают внешний ключ в дочерней таблице, но он не становится частью первичного ключа в дочерней таблице. При неидентифицируемом отношении мощность (множественность) определяет то, будет ли запись в дочерней таблице существовать без связи с записью в родительской таблице. Если мощность равна 1, должна присутствовать родительская запись. Если мощность равна 0..1, присутствие родительской записи необязательно. Неидентифицируемые отношения моделируются ассоциациями.

Редактирование свойств связей осуществляется в окне спецификаций (рисунок 3.9). Спецификации связей представлены в таблице 3.5

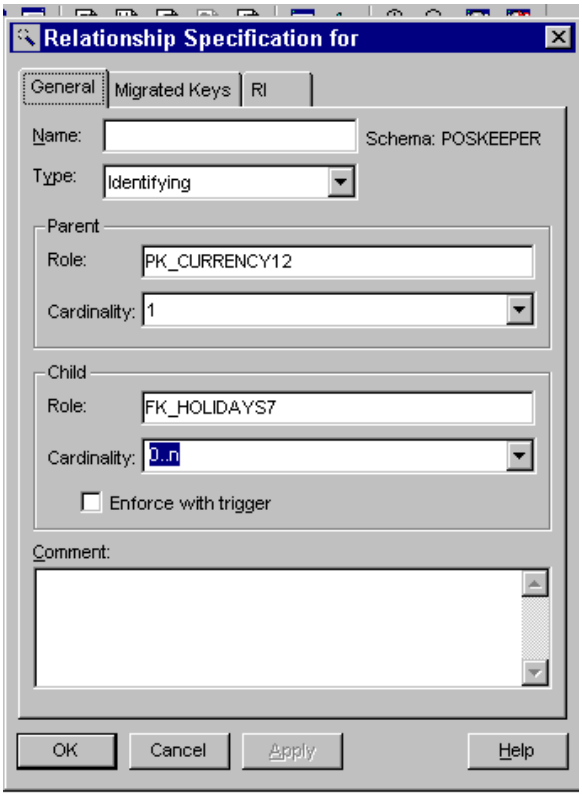


Рисунок 3.9 - Окно спецификации связи

Таблица 3.5 - Спецификация связей

Закладка	Описание
General	Задаются основные свойства связи: имя связи; тип связи; наименования ролей (Parent, Child); кардинальность для каждой роли.
Migrated Key	Содержит список внешних ключей, образующихся в результате создания связи.
RI	Задаются условия ссылочной целостности. Ссылочная целостность обеспечивается двумя способами: на основе триггеров; на основе декларативной ссылочной целостности (с использованием ограничений внешних ключей).

3.2.2 Построение диаграммы классов этапа анализа

Описание предметной области. Организация предоставляет услуги по трудоустройству. Ведется банк данных о существующих вакансиях. По каждой вакансии имеется следующая информация: предприятие, предоставляющее вакансию; название вакансии (должность); требования к соискателю: пол, возраст, образование, знание определенных видов деятельности (выбор из перечня - знание электронного документооборота, определенных прикладных программ и т.п.), коммуникабельность; обязанности (выбор из перечня – заключение договоров, распространение агитационного материала, работа с клиентами и т.п.); предполагаемая оплата, единицы измерения оплаты - рубли; оформление трудовой книжки (да, нет); наличие социального пакета (да, нет); срок начала открытия вакансии; срок закрытия вакансии (вакансия занята).

На рисунке 3.10 приведен фрагмент модели данных предметной области, разработанной в среде Rational Rose 2003 с использованием утилиты Rose Data Modeler.

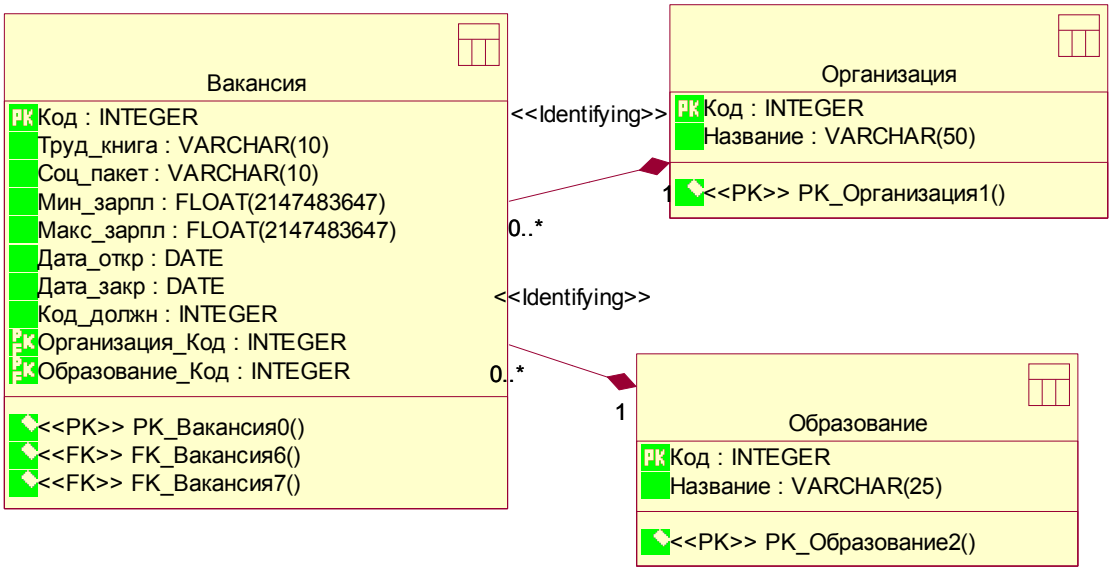


Рисунок 3.10 – Модель данных предметной области

Таким образом, использование инструментария Data Modeler позволяет разработать модель данных предметной области.

3.2.3 Вопросы и задания для самоконтроля

1 Что такое язык UML?

2 Какое средство моделирования данных позволяет строить физические модели для конкретных СУБД?

3 Как создать новую модель данных?

4 Какие возможности предоставляет инструментарий Data Modeler?

5 Какие UML диаграммы доступны в Rational Rose?

6 Для чего используется диаграмма Use Case?

7 Какие значки находятся в строке инструментов диаграммы модели данных и каково их назначение? Как настроить панель инструментов для диаграмм в Rational Rose?

8 Какие типы связи существуют между элементами диаграммы модели данных ?

9 Каким образом происходит редактирование свойств связей?

3.3 Построение моделей поведения программного обеспечения в среде Rational Rose

Чтобы представить поведение системы на логическом уровне в среде Rational Rose Enterprise 2003 используют диаграммы состояний, деятельности, последовательности и кооперации.

Диаграмма Statechart (диаграмма состояний) предназначена для описания состояний объекта и условий перехода между ними. Описание состояний позволяет точно описать модель поведения объекта при получении различных сообщений и взаимодействии с другими объектами. Модель состояний состоит из нескольких диаграмм состояний, по одной на каждый класс, поведение которого во времени важно для приложения.

При построении диаграмм состояний используют следующие понятия:

Автомат (State machine) - это описание последовательности состояний, через которые проходит объект на протяжении своего жизненного цикла, реагируя на события, и описание реакций на эти события.

Состояние (State) - это ситуация в жизни объекта, на протяжении которой объект удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого-то события.

Событие (Event) - это спецификация существенного факта, который происходит во времени и пространстве. Другими словами событием можно считать стимул, способный вызвать срабатывание перехода.

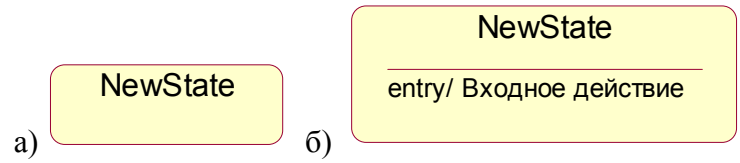
Переход (Transition) - это отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, должен выполнить некоторые действия и перейти во второе состояние как только произойдет определенное событие и будут выполнены заданные условия.

Действие (Action) - атомарное вычисление, которое приводит к смене состояния или возврату значения.

Деятельность (Activity) - неатомарное вычисление внутри автомата.

3.3.1 Инструментарий разработки диаграммы состояний в среде Rational Rose

Графически отображение состояния показано на рисунке 13.1



а) состояние без указания в нем действий или переходов; б) состояние с указанием действий или переходов

Рисунок 3.11 – Графическое изображение состояний на диаграмме состояний

Список действий состояния содержит перечень внутренних действий или деятельностей, которые выполняются в процессе нахождения моделируемого объекта в данном состоянии.

Составное состояние (composite state) - такое сложное состояние, которое состоит из других вложенных в него состояний. Вложенные состояния можно отобразить или скрыть при необходимости.

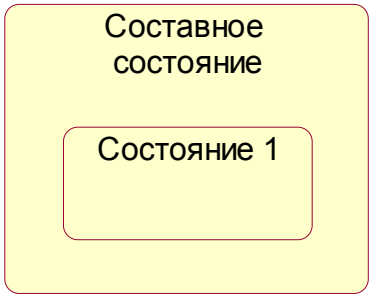


Рисунок 3.12 – Пример составного состояния

История состояния позволяет показать, что в следующий раз, когда система попадает в указанное состояние, она должна не начинать с начала состояний, а сразу перейти на последнее состояние, из которого вышла, то есть при первом входе в некоторое состояние производятся единичные действия, которые при следующем входе проделывать уже не нужно.

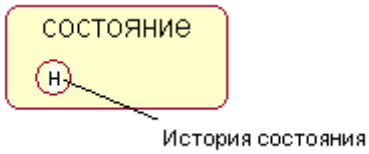


Рисунок 3.13 – Графическое представление истории состояния в среде Rational Rose

Назначение кнопок панели инструментов диаграммы состояний представлено в таблице 3.6.

Таблица 3.6 - Назначение кнопок панели инструментов диаграммы состояний

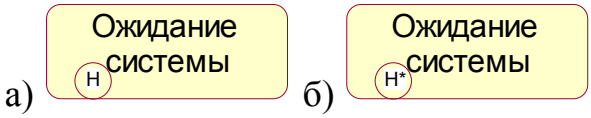
Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	State	Добавляет на диаграмму *состояние*
	Start State	Добавляет на диаграмму начальное *состояние*
	End State	Добавляет на диаграмму конечное *состояние*
	State Transition	Добавляет на диаграмму *переход*
	Transition to Self	Добавляет на диаграмму рефлексивный *переход*
	Horizontal Synchronization	Добавляет на диаграмму горизонтально расположенный символ синхронизации (по умолчанию отсутствует)
	Vertical Synchronization	Добавляет на диаграмму вертикально расположенный символ синхронизации (по умолчанию отсутствует)
	Decision	Добавляет на диаграмму символ принятия решения для альтернативных *переходов* (по умолчанию отсутствует)

Когда действие или деятельность в некотором состоянии завершается, поток управления сразу переходит в следующее состояние действия или деятельности. Для описания этого потока используются переходы, показывающие путь из одного состояния действия или деятельности в другое.

Простой переход (simple transition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим.

Срабатывание перехода зависит от наступления некоторого события, или от выполнения определенного условия, называемого сторожевым условием. Объект перейдет из одного состояния в другое в том случае, если произошло указанное событие и сторожевое условие приняло значение «истина».

При необходимости в диалоговом окне спецификации свойств выбранного состояния можно задать вложенное историческое состояние. Для этого следует выставить отметку у свойства State/activity history (Историческое состояние/деятельность) и нажать кнопку Apply. В результате внутри исходного состояния появится вложенное историческое состояние (рисунок 3.14 а).



(а) и состояния глубокой истории (б) для состояния «Ожидание системы»

Рисунок 3.14 - Добавление вложенного исторического состояния

Чтобы обычное историческое состояние превратить в состояние глубокой истории, следует дополнительно выставить отметку у свойства Sub state/activity history (Историческое подсостояние/деятельность), которое становится доступным для редактирования после выбора первого свойства, и нажать кнопку Apply. В результате внутри исходного состояния появится вложенное состояние глубокой истории (рисунок 3.14 б).

Чтобы обычное состояние превратить в составное, следует при добавлении нового состояния поместить его внутри границы того состояния, которое необходимо сделать составным (рисунок. 3.15).

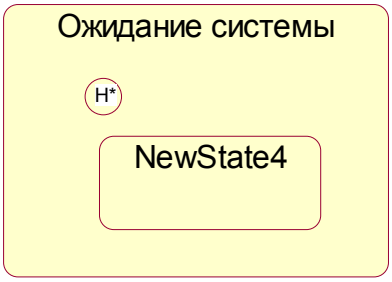


Рисунок 3.15 – Составное состояние

В окне спецификаций дополнительно можно определить следующие свойства состояний: задать текстовый стереотип состояния, определить внутренние действия на входе и выходе, а также внутреннюю деятельность. Эти свойства доступны для редактирования на вкладке General (Общие) и Actions (Действия). На вкладке Transitions (Переходы) можно определять и редактировать переходы, которые входят и выходят из рассматриваемого состояния. Последняя вкладка Swimlanes (Дорожки) служит для спецификации дорожек, которые, в контексте языка UML, определяются для диаграммы деятельности.

Для редактирования свойств перехода между состояниями необходимо воспользоваться окном спецификаций, представленном на рисунке 3.16.

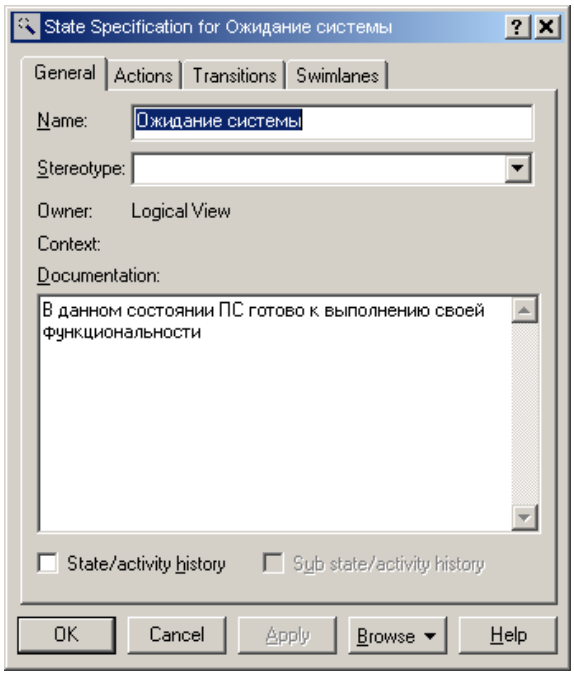


Рисунок 3.16 - Диалоговое окно спецификации свойств состояния

Спецификация дополнительных свойств осуществляется на вкладках Detail (Подробно) и General (Основное). Наиболее важными полями, доступными на соответствующих вкладках, являются поле ввода Guard Condition и поле ввода Action. Поле ввода Guard Condition служит для задания сторожевого условия, определяющего правило срабатывания соответствующего перехода. Поле ввода Action предназначено для спецификации действий, которое происходит при срабатывании перехода до того, как моделируемая система попадет в целевое состояние.

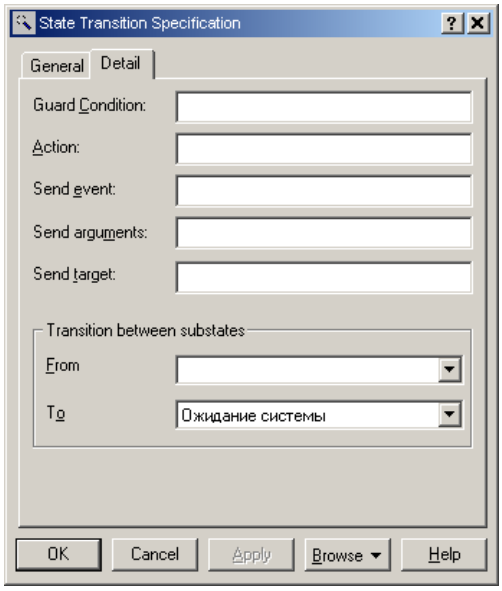


Рисунок 3.17 - Диалоговое окно спецификации свойств перехода, открытое на вкладке Detail (Подробно)

При необходимости можно определить сообщение о событии, происходящем при срабатывании перехода, а также визуализировать вложенность состояний и подключить историю отдельных состояний.

3.3.2 Построение диаграммы состояний метода приложения

На рисунке 3.18 представлен пример диаграммы состояний для объекта класса «Diskr_analiz» (дискриминантного анализа).

В результате анализа предметной области в качестве метода, реализующего процесс определения подходящей вакансии по данным резюме (анкеты) соискателя был выбран метод дискриминантного анализа, который позволяет относить отдельные объекты (соискателей) к определенным группам (вакансиям). Теоретические предпосылки дискриминантного анализа представлены в приложении Б.

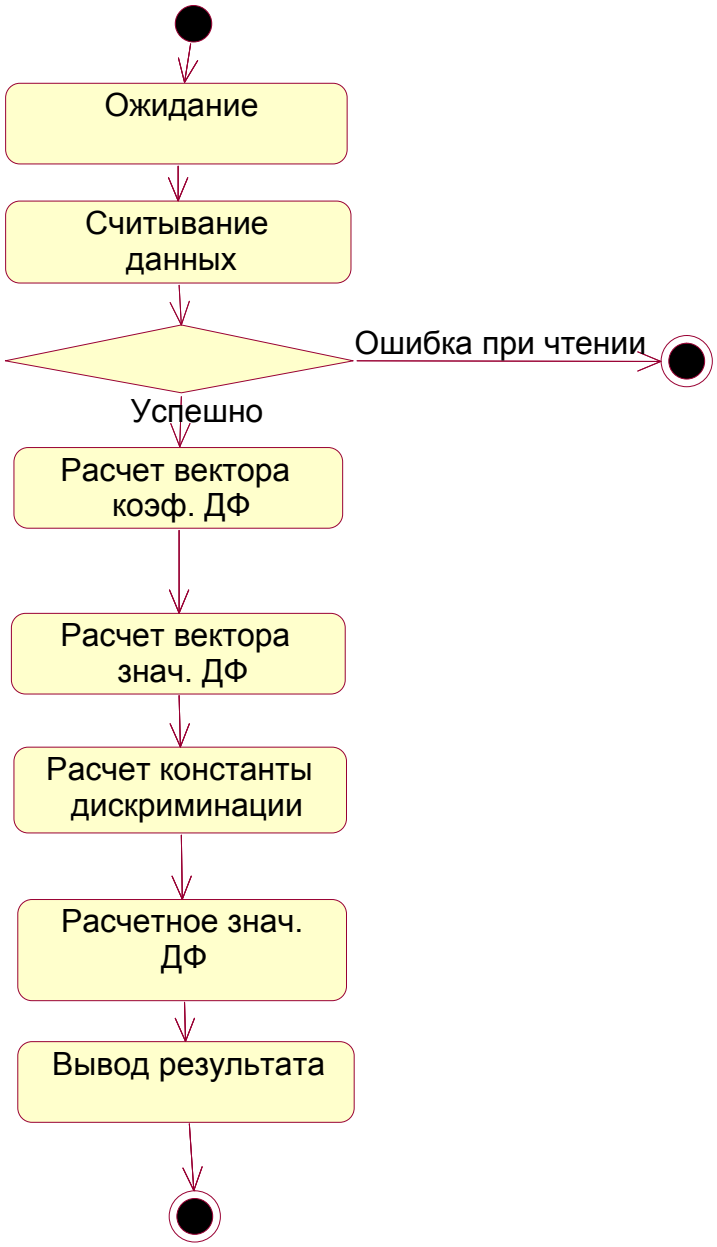


Рисунок 3.18 - Диаграмма состояний объекта класса «Diskr_analiz»

3.3.3 Вопросы и задания для самоконтроля

1 Для чего предназначена диаграмма состояний (Statechart)?

2 Как создать новую диаграмму состояний в среде Rational Rose 2003?

3 Какие бывают переходы между состояниями?

4 Какие спецификации можно задать для переходов между состояниями?

5 Что такое история состояний?

6 Что такое композитное состояние и как его создать?

7 Какие значки специфичны только для диаграммы состояний, расскажите о назначении каждого из них?

8 Что такое сценарий поведения системы? Для чего его создают?

9 Что такое сторожевое условие?

10 Как настроить панель инструментов, если на ней нет нужных значков?

3.4 Построение диаграммы классов этапа проектирования в среде Rational Rose

При помощи инструментария Rational Rose Enterprise Edition 2003 описывается внутренняя структура системы, наследование и взаимное положение классов друг относительно друга в виде диаграммы классов, отражающих логическое представление системы.

Диаграмма классов – основная диаграмма, обеспечивающая кодогенерацию.

Посредством инструментария Rational Rose Enterprise Edition 2003 возможно изменение в любой момент свойств любого класса или его связей, при этом другие спецификации, связанные с изменяемым классом, будут автоматически обновлены.

3.4.1 Инструментарий разработки диаграмм классов в среде Rational Rose Enterprise Edition 2003

Диаграмма *классов* является основным логическим представлением модели и содержит детальную информацию о внутреннем устройстве объектно-ориентированной программной системы.

Панель инструментов диаграммы классов представлена в таблице 3.7

Таблица 3.7 - Пиктограммы панели инструментов диаграммы классов

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Class	Добавляет на диаграмму *класс*
	Interfase	Добавляет на диаграмму *интерфейс*
	Unidirectional Association	Добавляет на диаграмму направленную *ассоциацию*
	Association Class	Добавляет на диаграмму *ассоциацию класс*
	Package	Добавляет на диаграмму пакет
	Dependency or Instantiates	Добавляет на диаграмму отношение зависимости
	Generalization	Добавляет на диаграмму отношение обобщения
	Realize	Добавляет на диаграмму отношение реализации

Продолжая разработку модели приложения для АИС «Трудоустройство», в качестве примера данного проекта, была построена для этой модели диаграмма классов. С этой целью было изменно предложенное по умолчанию имя диаграммы Main на «Диаграмма классов «Трудоустройство»», а имя добавленного на диаграмму класса на «Matrix» (рисунок 3.19).

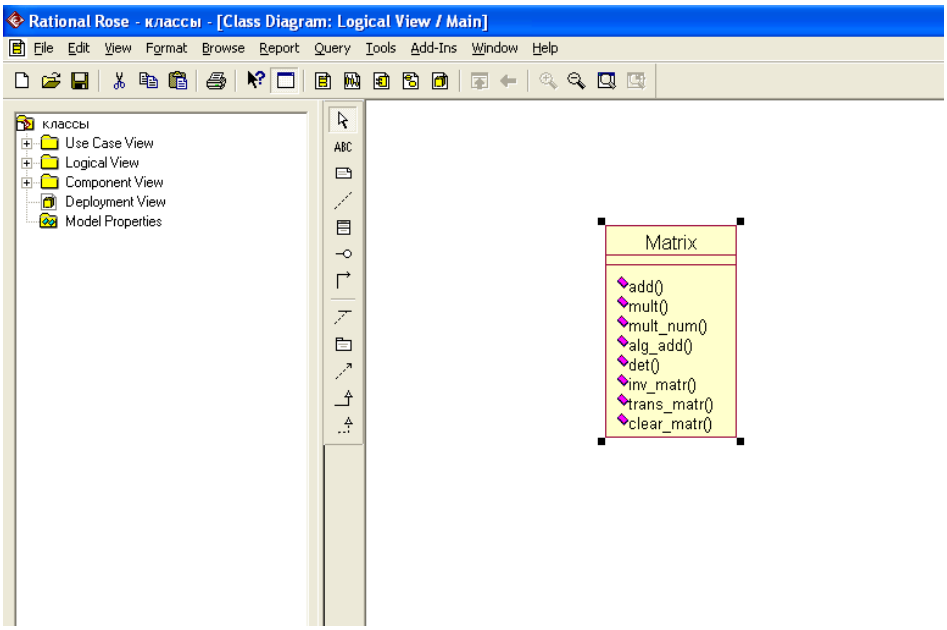


Рисунок 3.19 - Диаграмма классов после добавления на нее класса «Matrix»

Для класса «Matrix» необходимо уточнить его назначение в модели с помощью указания стереотипа и пояснений текста в форме документации, открыть диалоговое окно спецификации свойств этого класса (рисунок 3.20). И на вкладке General (Общие) выбрать из вложенного списка Stereotype (стереотип).

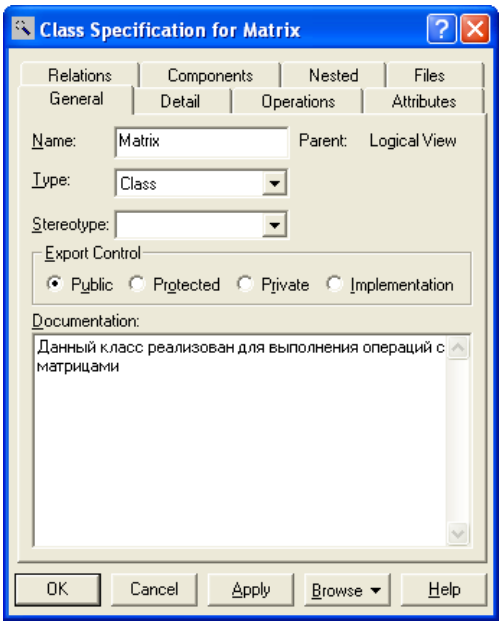


Рисунок 3.20 - Диалоговое окно спецификации свойств класса

Для отдельного класса можно уточнить также и другие его свойства, доступные для редактирования на вкладке Detail (Подробно) окна спецификации свойств этого класса. Например, на этой вкладке с помощью вложенного списка Multiplicity (Кратность) можно задать количество объектов или экземпляров данного класса, для чего следует выбрать строку с буквой n. Данное значение означает, что у класса может быть любое конечное число экземпляров (рисунок 3.21). Поле ввода с именем Space (Пространство) служит для указания объема абсолютной или относительной памяти, которая требуется, по оценке разработчика, для реализации каждого объекта данного класса. Применительно к рассматриваемой модели это поле можно оставить пустым.

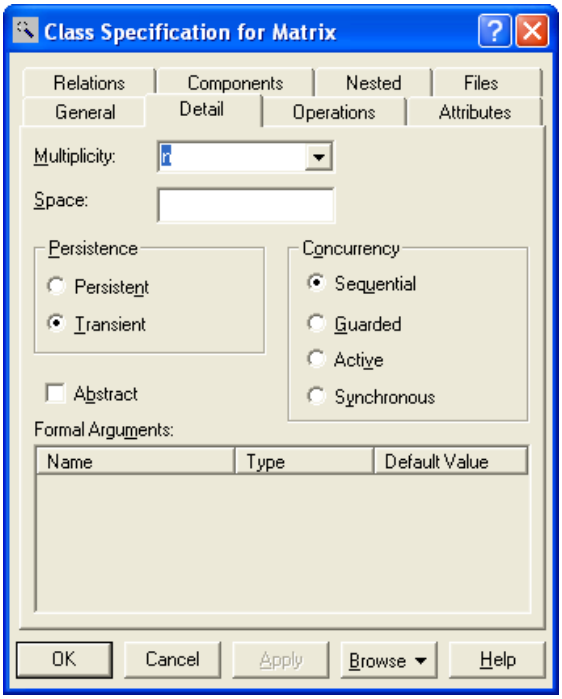


Рисунок 3.21 - Диалоговое окно спецификации свойств класса

Далее можно задать устойчивость классов в группе выбора Persistence (устойчивость). При этом выбор свойства Persistent (Устойчивый) означает, что информация об объектах данного класса должна быть сохранена в системе. Выбор свойства Transient (Временный) означает, что нет необходимости сохранять информацию об объектах данного класса в системе после завершения работы про-

граммного приложения. Применительно к рассматриваемой модели следует выбрать свойство Persistent.

В группе выбора Concurrency (Параллельность) можно указать условия на возможность реализации объектов данного класса в параллельных потоках управления. Для выбора могут быть использованы следующие свойства:

Sequential (Последовательный) - свойство по умолчанию, которое означает, что объекты класса будут вести себя нормально только при наличии одного потока управления, т. е. соответствующие операции объектов должны выполняться последовательно. В то же время при наличии нескольких потоков управления стабильное поведение объектов класса не гарантируется.

Guarded (Безопасный) - означает, что при наличии нескольких потоков управления объекты класса будут вести себя ожидаемым от них образом. Для этого объекты в различных потоках должны взаимодействовать друг с другом для того, чтобы гарантировать отсутствие конфликта между ними.

Active (Активный) - означает, что класс должен иметь свой собственный поток управления.

Synchronous (Синхронный) - означает, что объекты класса будут вести себя ожидаемым от них образом при наличии нескольких потоков управления. При этом нет необходимости во взаимодействии объектов в различных потоках управления, поскольку объекты данного класса могут самостоятельно разрешать возможные конфликты.

Для того, чтобы специфицировать класс как абстрактный, т.е. не имеющий экземпляров, следует на этой же вкладке поставить отметку в свойстве Abstract (Абстрактный).

Добавление и редактирование атрибутов классов

Из всех графических элементов среды Rational Rose 2003 класс обладает максимальным набором свойств, главными из которых являются его *атрибуты* и *операции*.

Добавить *атрибут* к созданному ранее классу можно одним из следующих способов:

- с помощью контекстного меню для класса, выделенного на диаграмме классов *(New→ Attribute* (*Новый →Атрибут*)). В этом случае активизируется курсор ввода текста в области графического изображения класса на диаграмме.

- с помощью контекстного меню для класса, выделенного в браузере проекта (*New→Attribute (Новый→Атрибут*)). В этом случае активизируется курсор ввода текста в области иерархического представления класса в браузере проекта под именем соответствующего класса.

- с помощью контекстного меню вызванного при позиционировании курсора в области открытой вкладки *атрибутов* в диалоговом окне свойств *Class Specification* соответствующего класса(*Insert (Вставить*).

После добавления *атрибута* к классу по умолчанию ему присваивается имя name и некоторый *квантор видимости* (рисунок 3.22).

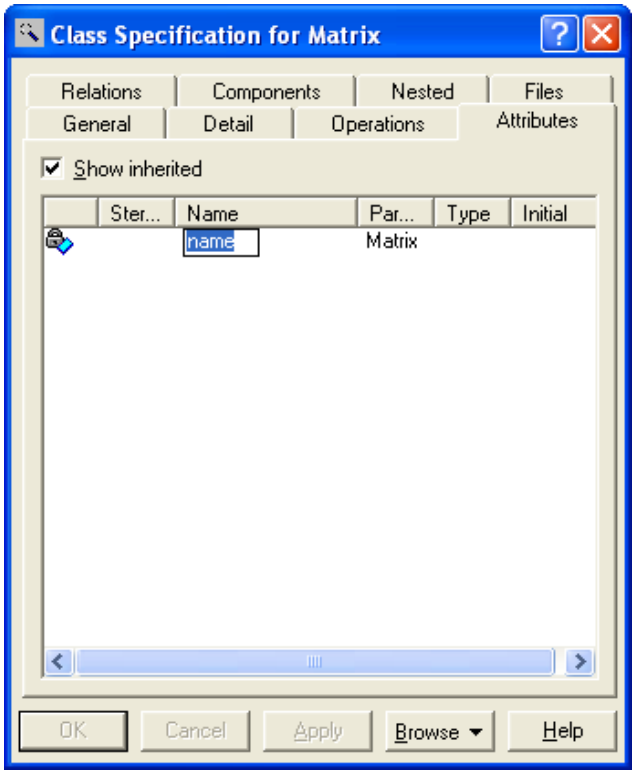


Рисунок 3.22 - Диалоговое окно спецификации свойств класса после добавления нового атрибута

Видимость *атрибутов* на диаграмме классов изображается в форме специальных пиктограмм. Используемые пиктограммы видимости изображаются перед именем соответствующего *атрибута* и представлены в таблице 3.8

Таблица 3.8 - Пиктограммы видимости атрибутов классов

Графическое изображение	Текстовый аналог	Назначение пиктограммы
	Public	Общедоступный или открытый. В нотации языка UML такому *атрибуту* соответствует знак «+»
	Protected	Защищенный. В нотации языка UML такому *атрибуту* соответствует знак «#»
	Private	Закрытый. В нотации языка UML такому *атрибуту* соответствует знак «-»
	Implementation	Реализация.

Для редактирования свойств *атрибутов* предназначено специальное диалоговое окно спецификации *атрибута* Class Attribute Specification, которое открывается двойным щелчком мыши на строке выбранного *атрибута* в окне спецификации свойств класса. В окне свойств отдельного *атрибута* класса можно задать *тип* данных *атрибута* и его начальное *значение*, а также назначить *атрибуту* стереотип из раскрывающегося списка или изменить его *квантор видимости*.

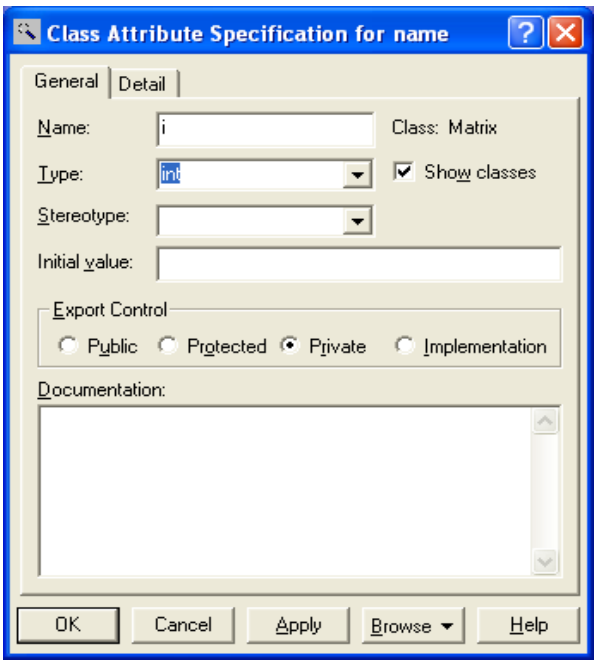


Рисунок 3.23 - Диалоговое окно спецификации свойств атрибута

Добавление и редактирование операций классов

Функционирование любой системы основано на выполнении отдельными его элементами тех или иных действий. В рассматриваемой модели все действия представляются с помощью *операций* классов. Таким образом, следующий этап разработки диаграммы классов связан со спецификацией *операций* классов.

Добавить *операцию* к созданному ранее классу можно одним из следующих способов:

- с помощью контекстного меню класса, выделенного на диаграмме классов: *New →Operation (Новая →Операция*) В этом случае активизируется курсор ввода в области графического изображения класса на диаграмме.

- с помощью контекстного меню класса, выделенного в браузере проекта: *New→Operation (Новая→Операция*). В этом случае активизируется курсор ввода в области иерархического представления класса в браузере под именем соответствующего класса.

- с помощью контекстного меню, вызванного при позиционировании курсора в области открытой вкладки *Операций* в диалоговом окне свойств *Class Specification* соответствующего класса (*Insert*).

После добавления *операции* к классу по умолчанию ей присваивается имя opname и некоторый *квантор видимости*. Видимость *операций* на диаграмме классов также изображается в форме специальных пиктограмм. Используемые пиктограммы видимости изображаются перед именем соответствующей *операции* и представлены в таблице 3.9

Таблица 3.9 - Пиктограммы видимости операций классов

Графическое изображение	Текстовый аналог	Назначение пиктограммы
	Public	Общедоступный или открытый. В нотации языка UML такому *атрибуту* соответствует знак «+»
	Protected	Защищенный. В нотации языка UML такому *атрибуту* соответствует знак «#»
	Private	Закрытый. В нотации языка UML такому *атрибуту* соответствует знак «-»
	Implementation	Реализация.

Каждая из *операций* классов имеет собственное диалоговое окно спецификации свойств Operation Specification, которое может быть открыто по двойному щелчку на имени *операции* на соответствующей вкладке спецификации класса или на имени этой *операции* в браузере проекта (рисунок 3.24).

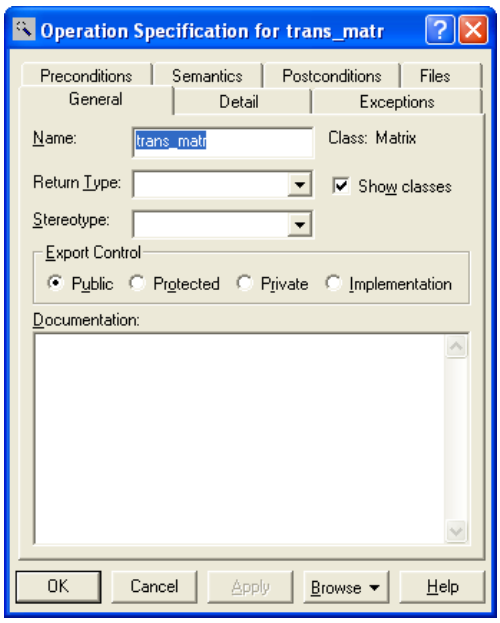


Рисунок 3.24 - Диалоговое окно спецификации свойств операции

Для *операций* классов кроме *квантора видимости* можно также задать: *аргументы* и их тип, *тип возвращаемого результата*, стереотип *операции*, а также определить протокол и размер, задать исключительные ситуации, специфицировать предусловия и постусловия и целый ряд других свойств. Для отдельной *операции* эти дополнительные свойства доступны для редактирования на вкладке Detail (Подробно) диалогового окна спецификации свойств выбранной *операции* (рисунок 3.25).

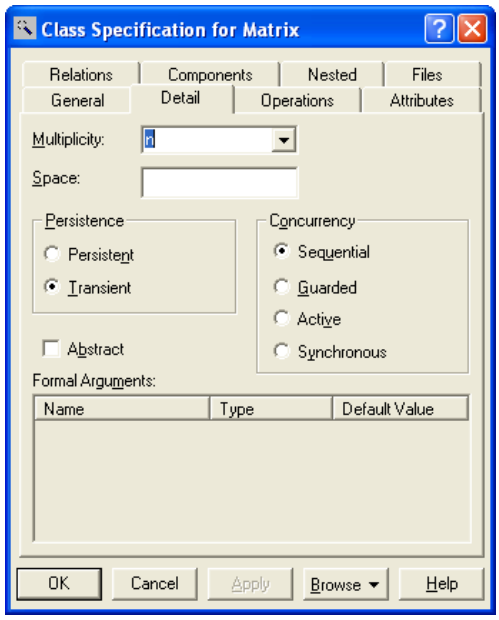


Рисунок 3.25 - Диалоговое окно спецификации свойств операции, открытое на вкладке Detail (Подробно)

На вкладке Detail в многостраничном поле Arguments (Аргументы) можно определить *аргументы редактируемой операции*. Для этого следует выполнить *операцию* контекстного меню Insert (Вставить). После этого в этом поле появится аргумент данной *операции* с именем по умолчанию argname. Для редактирования свойств аргумента предназначено специальное окно свойств аргумента.

На вкладке Detail в поле Protocol (Протокол) можно специфицировать порядок выполнения *операций* класса, например, указать, что одна *операция* не может быть вызвана раньше другой. Соответствующий текст в данное поле вводится с клавиатуры и попадает в генерируемый код в форме комментария. В поле Qualification (Квалификация) можно уточнить детали реализации *операции*, связанные с конкретным языком программирования. Соответствующий текст также вводится в данное поле с клавиатуры и попадает в генерируемый код в форме комментария.

Далее на этой же вкладке в полях Size (Размер) и Time (Время) можно специфицировать предполагаемый объем памяти и время, необходимое для выполнения *операции*. Соответствующая информация попадает в генерируемый код в форме комментария.

В группе выбора Concurrency (Параллельность) можно специфицировать условия на возможность параллельного выполнения данной *операции*. Для выбора могут быть использованы следующие свойства:

- Sequential (последовательная) - свойство по умолчанию, которое означает, что данная *операция* класса может быть выполнена только при наличии одного потока управления, т. е. соответствующая *операция* класса должна выполняться последовательно. При наличии нескольких потоков управления выполнение данной *операции* класса не гарантируется.

- Guarded (безопасная) - означает, что при наличии нескольких потоков управления выполнение данной *операции* класса гарантируется только в том случае, когда обеспечено взаимодействие объектов друг с другом в различных потоках.

- Synchronous (синхронная) - означает, что выполнение данной *операции* класса гарантируется при наличии нескольких потоков управления. При этом нет необходимости во взаимодействии объектов в различных потоках управления, поскольку данная *операция* класса будет выполняться в отдельном потоке управления вплоть до своего завершения.

Добавление отношений ассоциации на диаграмму классов и редактирование ее свойств

Для добавления на диаграмму ассоциации между двумя классами необходимо воспользоваться соответствующей пиктограммой на панели инструментов. При *направленной ассоциации* направление стрелки (её указатель) будет показывать на класс «приемник» (рисунок 3.26). После добавления направленной ассоциации между классами имя отношения, предложенное средой по умолчанию, можно изменить воспользовавшись окном спецификации. Доступ к диалоговому окну спецификации свойств *ассоциации* Association Specification можно получить после выделения линии *ассоциации* на диаграмме классов или в браузере проекта и двойного щелчка на ней левой кнопки мыши (рисунок 3.27).

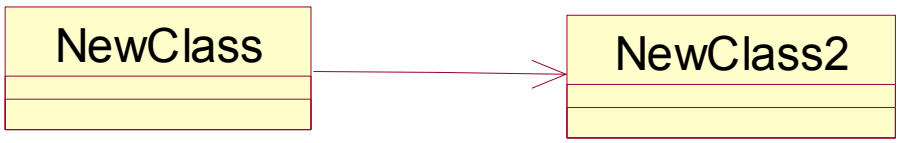


Рисунок 3.26 - Фрагмент диаграммы классов после добавления на неё направленной ассоциации

В данном окне для *ассоциации* можно задать также *кратность* каждого из концов *ассоциации*, стереотип, использовать ограничения и роли, а также некоторые другие свойства.

Если *ассоциация* является ненаправленной, то порядок выбора классов может быть произвольный, а после добавления *ассоциации* на диаграмму классов следует

изменить значение соответствующего свойства данной *ассоциации*. С этой целью необходимо перейти на вкладку Role A Detail в окне спецификации свойств *ассоциации* и убрать отметку у свойства *Navigable (Навигация)*.

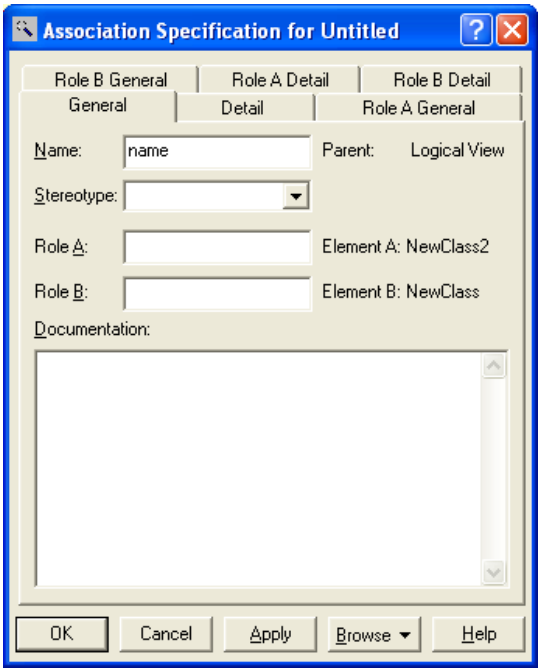


Рисунок 3.27 - Диалоговое окно спецификации свойств ассоциации

Добавление отношений агрегации и композиции на диаграмму классов и редактирование их свойств

Агрегация – это частный случай ассоциации, описывающий объекты, состоящие из частей. Например, газонокосилка состоит из ножа, двигателя, нескольких колес и корпуса. Здесь газонокосилка является агрегатом (незакрашенный ромбик), а остальные детали – составляющими частями (стрелка).

Композиция – это частный случай агрегации. Композиция подразумевает, что части принадлежат целому. Например, университет, состоящий из факультетов, которые в свою очередь, состоят из кафедр.

Добавить на диаграмму отношение *агрегации* между двумя классами можно следующими способами:

- щелкнуть на кнопке с изображением отношение *агрегации* на специальной панели инструментов и провести линию *агрегации* от одного класса к другому;

- провести линию *ассоциации* между выбранными классами и изменить ее свойства таким образом, чтобы превратить данную *ассоциацию* в *агрегацию*.

В первом случае может оказаться, что по умолчанию на специальной панели инструментов диаграммы классов отсутствует кнопка с пиктограммой *агрегации*. В этом случае её необходимо предварительно добавить на панель инструментов. Во втором случае следует открыть окно спецификации свойств *ассоциации* Association Specification и на вкладке деталей соответствующего конца *ассоциации* выставить отметку в строке выбора Aggregate (*агрегация*) (рисунок 3.28).

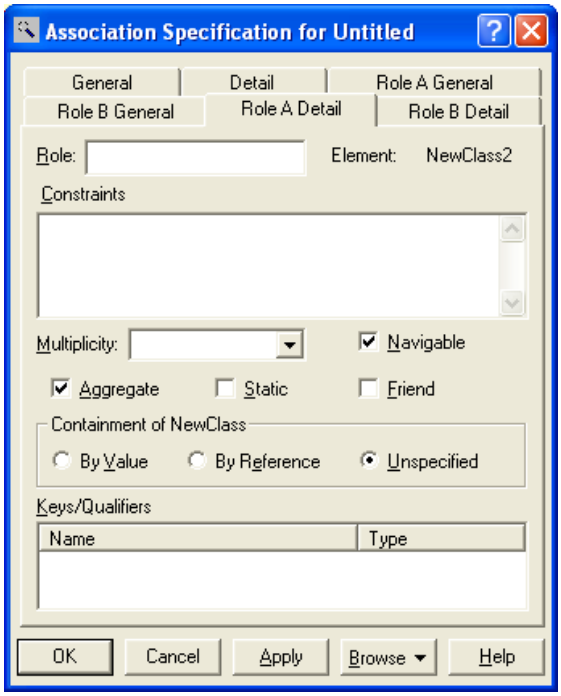


Рисунок 3.28 - Диалоговое окно спецификации свойств ассоциации

Соответствующий фрагмент диаграммы классов после изменения *ассоциации* между классами на отношение *агрегации* будет иметь следующий вид (рисунок 3.29).

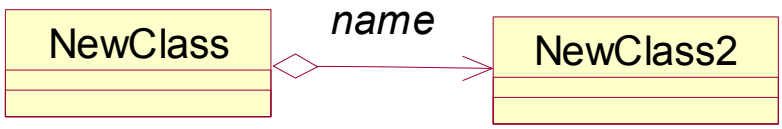


Рисунок 3.29 - Фрагмент диаграммы классов модели после добавления на нее отношения агрегации

Для изображения отношения *композиции* можно также вначале изобразить обычную *ассоциацию*, после чего, открыв окно ее свойств на вкладке деталей соответствующего конца *ассоциации*, поставить отметку в строке выбора *Aggregate* (агрегация) и в секции *Containment* (локализация) выбрать опцию By Value (по значению). По умолчанию эта опция не специфицирована, т.е. выставлена отметка опции Unspecified (рисунок 3.30).

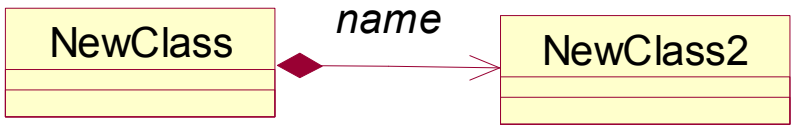


Рисунок 3.30 - Фрагмент диаграммы классов модели после добавления на нее отношения композиции

Добавление отношения обобщения на диаграмму классов и редактирование ее свойств

Добавление на диаграмму отношения *обобщения* между двумя классами выполняется таким же способом что и отношение ассоциации или агрегации. Пример диаграммы классов с использованием отношения обобщения приведен на рисунке 3.31.

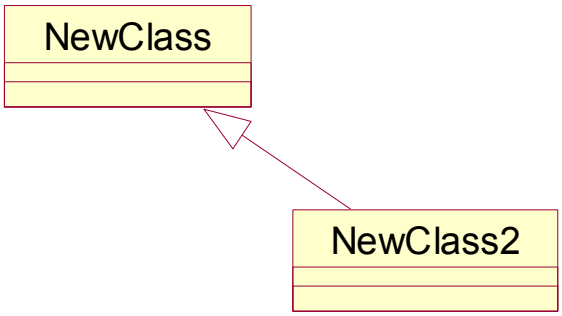


Рисунок 3.31 – Пример диаграммы классов после добавления на неё отношения обобщения

Для изменения имени отношения *обобщения*, предложенное средой по умолчанию, а также установки необходимых свойств и настроек можно воспользоваться окном спецификации свойств *обобщения* (рисунок 3.32).

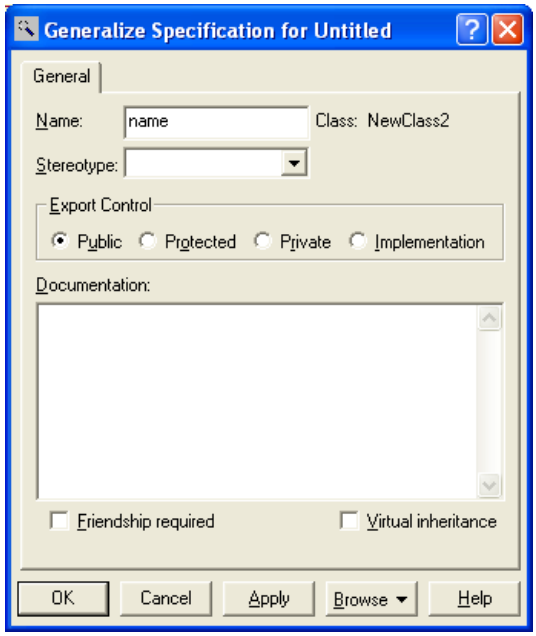


Рисунок 3.32 - Диалоговое окно спецификации свойств отношения обобщения

3.4.2 Построение диаграммы классов программной системы

В результате анализа построенной диаграммы состояний была построена диаграмма классов представленная на рисунке 3.33

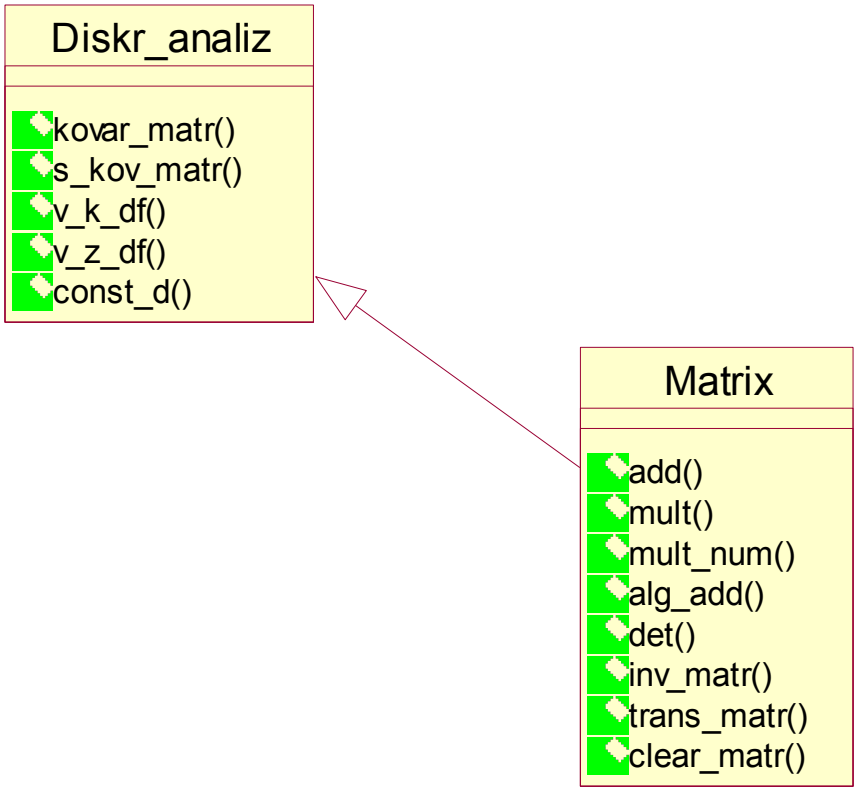


Рисунок 3.33 – Диаграмма классов для алгоритма дискриминантного анализа

3.4.3 Вопросы и задания для самоконтроля

1 Каково назначение диаграммы классов?

2 Какими способами можно создать диаграмму?

3 Какие инструменты доступны для диаграммы?

4 Какие команды предоставляет контекстное меню класса?

5 Как настроить свойства атрибутов класса?

6 Как настроить свойства методов класса?

7 Какие типы отношений классов вы знаете?

3.5 Построение диаграммы компонентов в среде Rational Rose Enterprise Edition 2003

Диаграммой компонентов (Component diagram) называется диаграмма UML, на которой показаны компоненты системы и зависимости между ними.

Компонентом называется физический модуль кода. Компонентами бывают как библиотеки исходного кода, так и исполняемые файлы. Например, .h и .cpp и .exe - будут отдельными компонентами.

3.5.1 Инструменты разработки диаграмм компонентов в среде Rational Rose Enterprise Edition2003

Диаграмма *компонентов* служит частью физического представления модели, играет важную роль в процессе ООАП и является необходимой для генерации программного кода. Для разработки диаграмм *компонентов* в браузере проекта предназначено отдельное представление *компонентов* (Component View), в котором уже содержится диаграмма *компонентов* с пустым содержанием и именем по умолчанию Main (Главная).

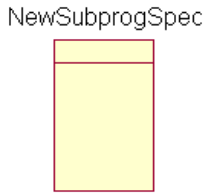
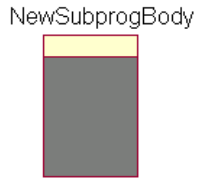
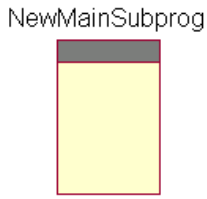
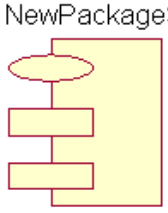
В результате работы с диаграммой компонентов появляется новое окно с чистым рабочим листом диаграммы *компонентов* и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разработки диаграммы *компонентов* (таблица 3.10).

Таблица 3.10 - Пиктограммы панели инструментов диаграммы компонентов

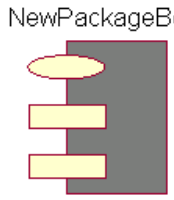
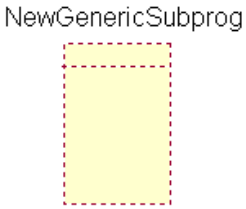
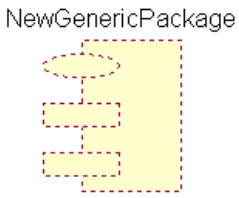
Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Component	Добавляет на диаграмму *компонент*
	Package	Добавляет на диаграмму пакет
	Dependency	Добавляет на диаграмму отношение зависимости
	Subprogram Specification	Добавляет на диаграмму спецификацию подпрограммы
	Subprogram Body	Добавляет на диаграмму тело подпрограммы
	Main Program	Добавляет на диаграмму главную программу
	Package Specification	Добавляет на диаграмму спецификацию пакета
	Package Body	Добавляет на диаграмму тело пакета
	Task Specification	Добавляет на диаграмму спецификацию задачи
	Task Body	Добавляет на диаграмму тело задачи
	Generic Subprogram	Добавляет на диаграмму типовую подпрограммы(по умолчанию отсутствует)
	Generic Package	Добавляет на диаграмму типовой пакет (по умолчанию отсутствует)
	Database	Добавляет на диаграмму базу данных (по умолчанию отсутствует)

Использование *стереотипов* существенно увеличивают наглядность графического представления диаграммы *компонентов* и позволяют уточнить характер реализации модели программистом на выбранном языке программирования. Графическое изображение *стереотипов* и их краткая характеристика приводятся в таблице 3.11.

Таблица 3.11 - Пиктограммы стереотипов компонентов

Графическое изображение и имя по умолчанию	Название стереотипа	Характеристика стереотипа компонента
1	2	3
	Subprogram Specification	Спецификация подпрограммы. Содержит описание переменных, процедур и функций и не содержит определений классов
	Subprogram Body	Тело подпрограммы. Содержит реализацию процедур и функций, не относящихся к каким-то классам, при этом не содержит определений классов или реализаций операций других классов
	Main Program	Главная программа. Реализует базовую логику работы программного приложения и содержит ссылки на другие *компоненты* модели
	Package Specification	Спецификация пакета. Содержит определение класса, его атрибутов и операций. В языке программирования C++ спецификации пакета соответствует отдельный файл с расширением «h»

Продолжение таблицы 3.11

1	2	3
	Package Body	Тело пакета. Содержит код реализации операций класса. В языке программирования C++ спецификации пакета соответствует отдельный файл с расширением «cpp»
	Task Specification	Спецификация задачи. Может содержать определение класса, его атрибутов и операций, которые предполагается использовать в независимом потоке управления
	Task Body	Тело задачи. Может содержать реализацию операций класса, которые имеют независимый поток управления.
	Generic Subprogram	Типовая подпрограмма. Содержит описание переменных, процедур и функций, которые могут быть использованы в нескольких программных приложениях. При этом типовая подпрограмма не содержит определений классов
	Generic Package	Типовой пакет. Содержит определение класса, его атрибутов и операций, которое может быть использовано в нескольких программных приложениях
	Database	База данных. Содержит определение одного или нескольких классов, их атрибутов и, возможно, операций. При этом соответствующие классы могут быть реализованы в форме одной или нескольких таблиц базы данных

Каждому из *компонентов*, как правило, соответствует отдельный файл исходной сборки программного приложения.

Добавление компонента на диаграмму компонентов и редактирование его свойств

Добавить компоненты на диаграмму компонентов можно одним из следующих способов:

1) используя соответствующие пиктограммы компонента на специальной панели инструментов;

2) используя главное меню Tools (Create→Component);

3) с помощью операции контекстного меню компонента (New→Component).

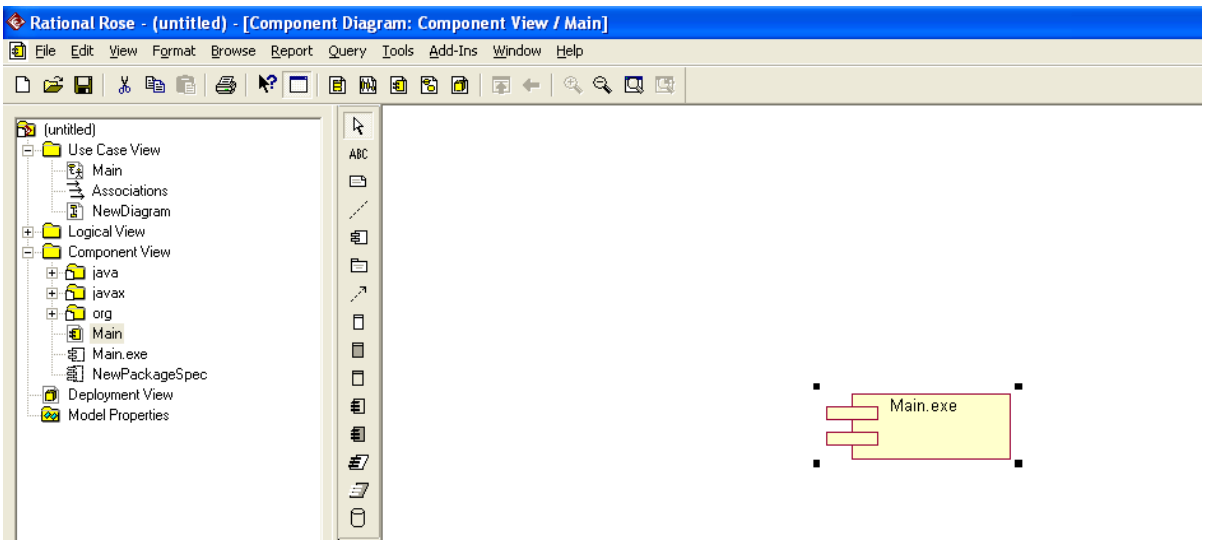


Рисунок 3.34 - Диаграмма компонентов после добавления компонента Main.exe

Для каждого компонента можно определить различные свойства, такие как стереотип, язык программирования, декларации, реализуемые классы. Редактирование этих свойств для компонента осуществляется с помощью диалогового окна спецификации свойств (рисунок 3.35).

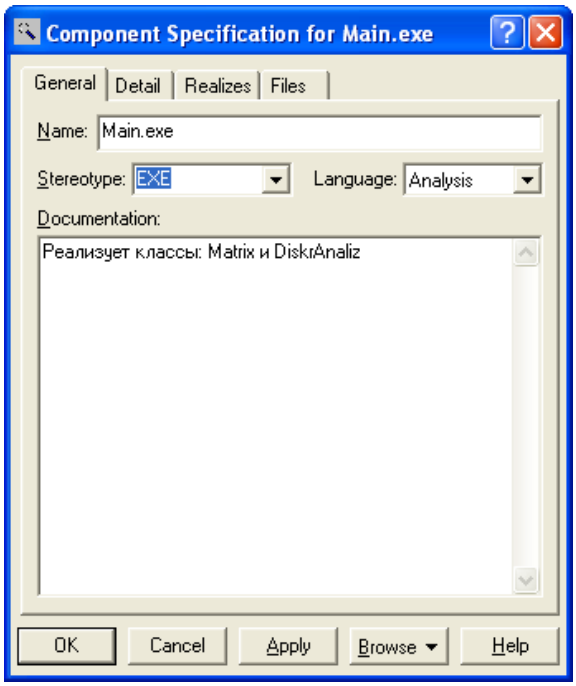


Рисунок 3.35 - Диалоговое окно спецификации свойств компонента Main.exe

В частности, для компонента `Main.exe` можно выбрать стереотип <<EXE>> из предлагаемого вложенного списка, поскольку применительно к разрабатываемой модели предполагается реализация этого компонента в форме исполнимого файла. При этом на вкладке `Realizes` (Реализует) содержатся все классы (включая и актеров) которые на данный момент присутствуют в модели.

По умолчанию в среде Rational Rose 2003 для всех добавляемых на диаграмму компонентов в качестве языка реализации используется язык анализа, который в последствии следует изменить на тот язык программирования, который предполагается использовать для написания программного кода. В дальнейшем при генерации программного кода необходимо будет дополнительно выбрать те классы, которые реализует тот или иной компонент модели.

Добавление отношения зависимости и редактирование его свойств

Добавление отношения зависимости на диаграмму компонентов аналогично добавлению соответствующего отношения на диаграмму вариантов использования.

3.5.2 Пример построения диаграммы компонентов

Ниже представлена диаграмма компонентов для программного средства, реализующего алгоритм дискриминантного анализа (рисунок 3.36).

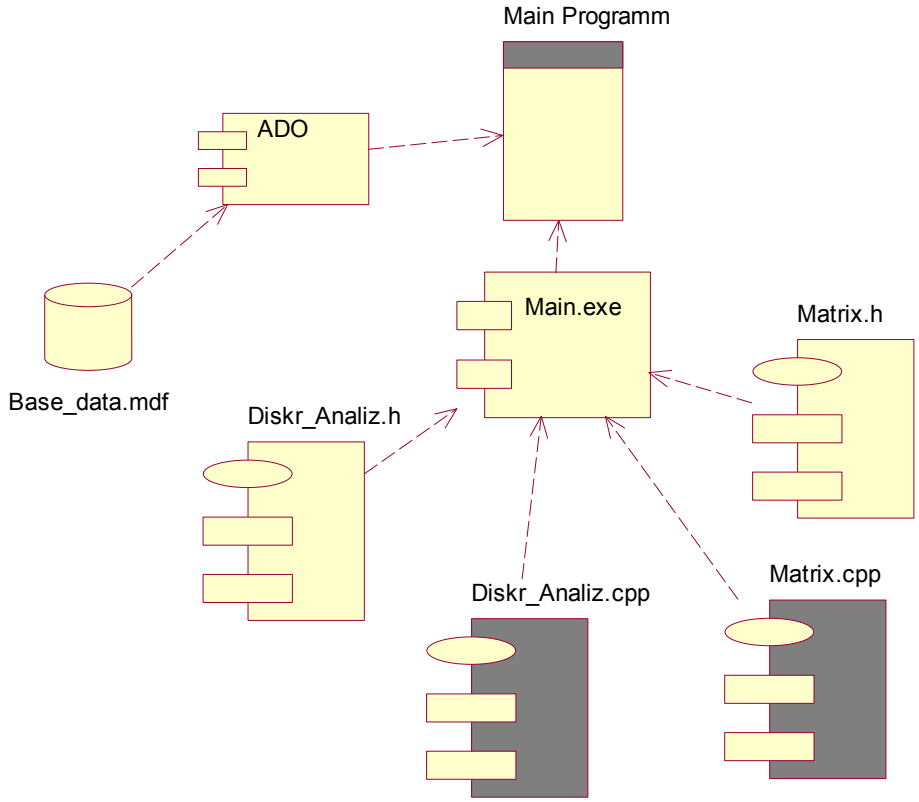


Рисунок 3.36 – Диаграмма компонентов рассматриваемого ПС

Таким образом, в результате анализа диаграммы классов и проектирования физич еского представления системы была построена диаграмма компонентов.

3.5.3 Вопросы и задания для самоконтроля

1 Каково назначение диаграммы компонентов?

2 Какими способами можно создать диаграмму компонентов?

3 Какие инструменты доступны для данной диаграммы?

4 Что такое *стереотип*? Для чего используют стереотипы?

5 Перечислите некоторые из существующих стереотипов?

6 Какие свойства можно определить для любого из компонент?

7 Каким образом можно добавить отношение зависимости и осуществить редактирование её свойств?

8 Что необходимо сделать чтобы назначить компоненту стереотип *exe* ?

9 С помощью какой вкладки можно увидеть все классы присутствующие в модели?

3.6 Генерация программного кода в среде Rational Rose Enterprise Edition 2003

Возможность генерации программного кода является мощным преимуществом case-средства Rational Rose. Варианты генерации программ меняются в зависимости от установленной версии Rational Rose.

3.6.1 Подготовка к генерации программного кода

В разделе 2.6.3 представлена методика генерации программного кода, состоящая из следующих этапов:

- проверка модели;
- создание компонентов;
- отображение классов на компоненты;
- установка свойств генерации программного кода;
- генерация программного кода.

Проверка модели

Проверка модели поможет выявить неточности и недостатки построенной модели, нежелательные с точки зрения последующей генерации программ.

В общем случае проверка модели может выполняться на любом этапе работы над проектом. Однако после завершения разработки графических диаграмм она является обязательной, поскольку позволяет выявить целый ряд ошибок разработчика.

Для проверки модели необходимо выполнить операцию главного меню: Tools→Check Model (Инструменты→Проверить модель). В окне журнала отобразятся результаты проверки разработанной модели (рисунок 3.37). Прежде чем приступить к генерации текста программного кода необходимо добиться устранения всех ошибок и предупреждений.

Рисунок 3.37- Вид журнала при отсутствии ошибок по результатам проверки модели

Создание компонентов

Перед генерацией программного кода, как правило, необходимо создать диаграмму компонентов показывающую физическое представление системы. После создания компонентов необходимо добавить зависимости между ними на диаграмме Компонентов, показывающие зависимости во время компиляции системы.

При генерации программ на C++, Java или Visual Basic выполнять подобный шаг не обязательно. Rational Rose создаст автоматически соответствующий компонент для каждого из классов.

Отображение классов на компоненты

Каждый компонент исходного кода - это файл с исходным программным кодом для одного или нескольких классов. В C++ каждый класс отображается на два компонента с исходным кодом: файл заголовка и основной файл (тело).

Для некоторых языков программирования Rational Rose может генерировать программный код самостоятельно. При генерации в Rational Rose программ Java и Visual Basic производится так же и генерация нужных компонентов и отображение классов. Но ни для одного из языков программирования не генерируются зависимости! Поэтому рекомендуется выполнять этот шаг независимо от применяемого языка программирования.

Установка свойств генерации программного кода

Для каждого языка в Rational Rose предусмотрен ряд определенных свойств генерации программного кода. Можно установить несколько параметров генерации программного кода для классов, атрибутов, компонентов и других элементов модели. Этими свойствами определяется способ генерации программ. В Rational Rose предлагаются общепринятые параметры по умолчанию.

Для того чтобы пользоваться возможностями C++, необходимо описать назначение этих свойств, список которых доступен во вкладке C++ спецификаций класса (рисунок 3.38)

Перед генерацией программного кода рекомендуется анализировать эти свойства и вносить необходимые изменения.

Назначение свойств:

1) **CodeName** - устанавливает имя класса в создаваемом коде. Данное свойство необходимо устанавливать только в том случае, если имя класса должно быть отлично от имени заданного в модели Rational Rose. Данное свойство необходимо использовать для создания работоспособного кода C++, если для классов в модели используются русские имена.

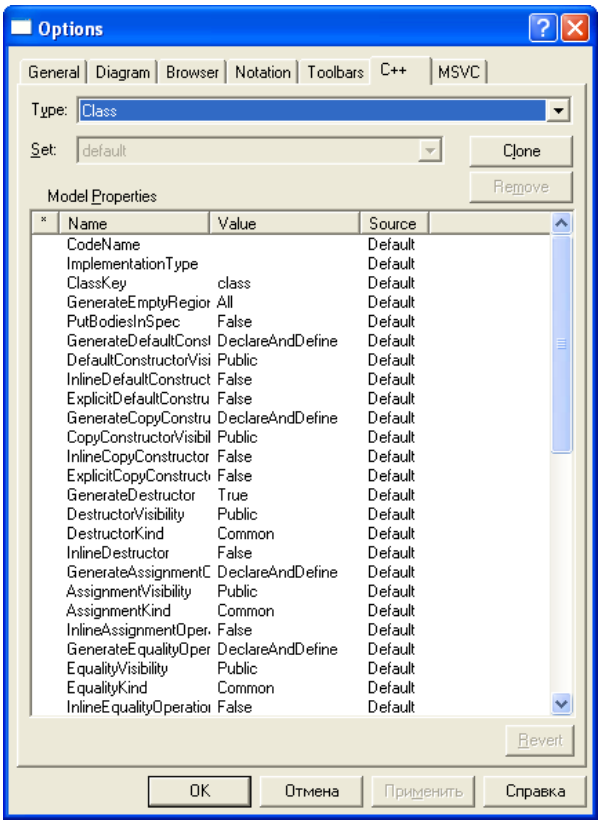


Рисунок 3.38 – Окно свойств генерации кода на C++

2) **ImplementationType** - позволяет использовать простые типы вместо определения класса, устанавливаемого Rational Rose по умолчанию. При задании этого параметра создается директива typedef.

3) **ClassKey** - используется для задания типа класса, такого как class, struct, или union. Если тип не указан, то создается класс.

4) **GenerateEmptyRegion** - свойство указывает, как будет создаваться пустой раздел protected: None - пустой раздел не будут создан; Preserved - пустой раздел будет создан, если будет установлено свойство «preserve=yes»; Unpreserved — пустой раздел будет создан, если будет установлено свойство «preserve=no»; All — всегда будет создаваться.

5) **PutBodiesInSpec** - если установлено как true, то в заголовочный файл попадет и описание тела класса. Используется для компиляторов, которым необходимо определение шаблона класса в каждом компилируемом файле.

6) **GenerateDefaultConstructor** - позволяет установить, необходимо ли создавать конструктор для класса по умолчанию. Может принимать следующие значения:

– declare and define - создается определение для конструктора и скелет конструктора в теле класса;

– declare only - создается только определение;

– do not declare - не создается ни определения, ни скелета конструктора.

7) **DefaultConstructorVisibility** - устанавливает раздел, в котором будет определен конструктор по умолчанию: public, protected, private, implementation.

8) **InlineDefaultConstructor** - устанавливает, будет ли конструктор по умолчанию создаваться как inline подстановка. Если конструктора по умолчанию нет, то данное свойство не оказывает на код никакого эффекта.

9) **ExplicitDefaultConstructor** - устанавливает конструктор по умолчанию как explicit (явно заданный).

10) **InlineRelationalOperations** - определяет, будут ли функции операторов сравнения создаваться как inline подстановка.

11) **GenerateStorageMgmtOperations** - определяет, будут ли переопределяться операторы new и delete в классе.

12) **StorageMgmtVisibility** - определяет раздел, в который будут помещены операторы new и delete.

13) **InlineStorageMgmtOperations** - определяет, будут ли операторы new и delete определены как inline подстановка.

14) **GenerateSubscriptOperation** - определяет, будет ли переопределен оператор .

15) **Subscript Visibility** определяет - раздел, в который будет помещен оператор.

16) **SubscriptKind** - определяет вид функций оператора : Common - обычная, Virtual - виртуальная, Abstract - абстрактная.

17) **SubscriptResultType** - определяет тип возвращаемого выражения для оператора .

18) **InlineSubscriptOperation** - определяет, будет ли оператор определен как inline подстановка.

19) **GenerateDereferenceOperation** - определяет, будет ли переопределен оператор *.

20) **Dereference Visibility** - определяет раздел, в который будет помещен оператор *.

21) **DereferenceKind** - определяет вид функций оператора *: Common - обычная, Virtual - виртуальная, Abstract - абстрактная.

22) **DereferenceResultType** - определяет тип возвращаемого выражения для оператора *.

23) **InlineDereferenceOperation** - определяет, будет ли оператор * определен, как inline подстановка.

24) **GenerateIndirectionOperation** - определяет, будет ли переопределен оператор ->.

25) **IndirectionVisibility** - определяет раздел, в который будет помещен оператор ->.

26) **IndirectionKind** - определяет вид функций оператора ->: Common - обычная, Virtual - виртуальная, Abstract - абстрактная.

27) **IndirectionResultType** - определяет тип возвращаемого выражения для оператора ->.

28) **InlineIndirectionOperation** - определяет, будет ли оператор -> определен как inline подстановка.

29) **GenerateStreamOperations** - определяет, будут ли переопределены операторы потоков (« и »).

3.6.2 Кодогенерация

Пакет Rational Rose Enterprise Edition предлагает в меню Tools несколько вариантов, специфичных для конкретного языка программирования (рисунок 3.39).

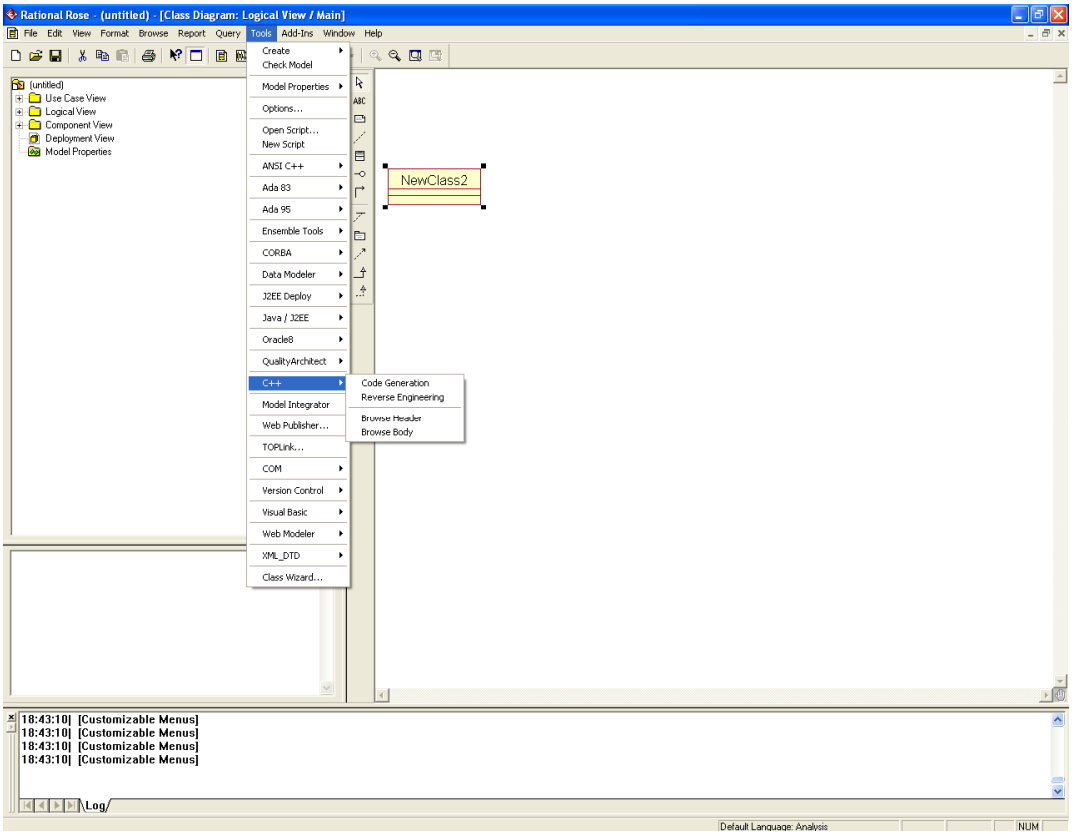


Рисунок 3.39 - Пункты меню генерации кода

Для того чтобы показать или скрыть необходимые пункты меню, необходимо выбрать пункт Add-Ins → Add-Ins (Надстройки → Менеджер надстроек). В диалоговом окне Add-In Manager (рисунок 3.40) с помощью соответствующих флажков можно показать или скрыть нужные варианты для различных языков.

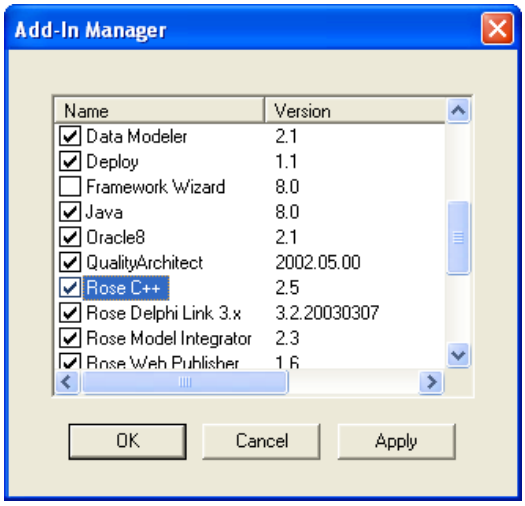


Рисунок 3.40 - Менеджер надстроек Add-Ins

Генерация программного кода в среде IBM Rational Rose Enterprise Edition 2003 возможна для отдельного класса или компонента. Для этого необходимо выделить в браузере проекта нужный элемент модели и выполнить операцию контекстного меню: Tools→C++→Code Generation- (Язык C++→Генерировать код). В результате этого будет открыто диалоговое окно с предложением выбора классов для генерации программного кода на выбранном языке программирования (рисунок 3.41). После выбора соответствующих классов и нажатия кнопки OK происходит кодогенерация.

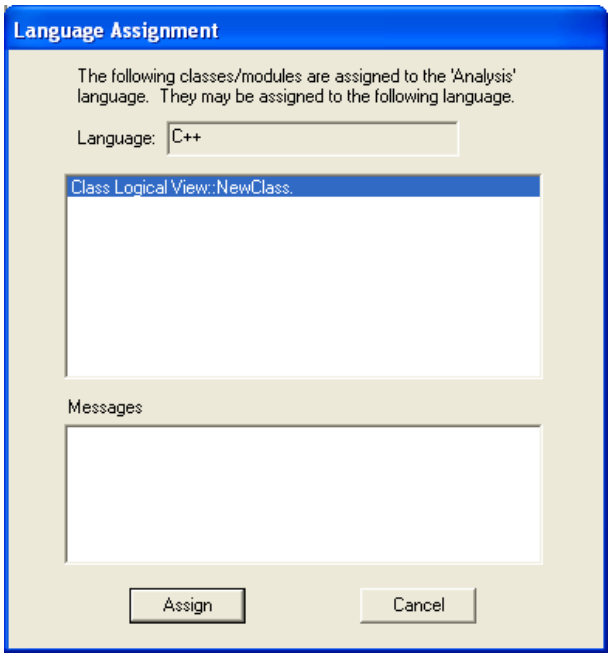


Рисунок 3.41 - Окно выбора классов для генерации программного кода

Затем происходит компиляция и выдается окно статуса (Code Generation Status), в котором показана информация о том, какой класс был закодирован, а также количество ошибок и предупреждений возникших при кодогенерации (рисунок 3.42). Сгенерированные файлы с текстом программного кода содержат минимум информации. Для включения дополнительных элементов в программный код следует изменить свойства генерации программного кода, установленные по умолчанию.

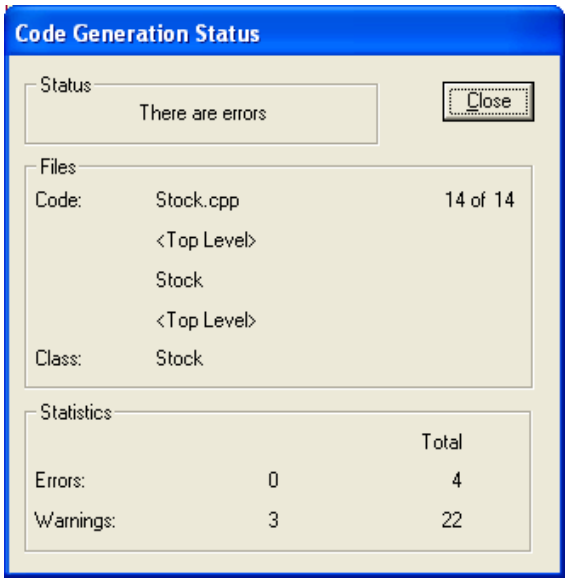


Рисунок 3.42 – Окно статуса компиляции

Результаты генерации

В результате кодогенерации Rational Rose создает два файла с расширением “.h” и “.cpp”, названия они имеют такие же, что и название класса. Следует заметить, что при установленной на компьютер интегрированной среды сгенерированные файлы с текстом программного кода автоматически открываются в этой среде после двойного щелчка на пиктограмме этих файлов.

В заключение следует отметить, что эффект от использования Case – средства Rational Rose проявляется при разработке *масштабных проектов* в составе команды или проектной группы.

Результаты кодогенерации приведены в Приложении А**.**

3.6.3 Вопросы и задания для самоконтроля

1 Какие диаграммы необходимо предварительно разработать, чтобы выполнить кодогенерацию?

2 Как посмотреть исходный код?

3 Какие установки свойств доступны на вкладке C++?

4 Каким образом можно задать или изменить уже заданные свойства выбранного языка для генерации?

5 Какова структура создаваемого кода?

6 Что необходимо добавить в шаблоны классов для получения работоспособного приложения?

7 Какие шаги нужно предпринять для обновления модели по исходному коду?

8 Какие языки программирования позволяют генерировать код без предварительного создания диаграммы компонентов?

9 Каким образом можно отобразить выбранный класс на Компонент?

10 Где отображаются ошибки и предупреждения возникшие при генерации программного кода?

Список использованных источников

1 Иванова, Г.С. Технология программирования/ Г.С. Иванова – М.: МГТУ им. Баумана, 2002. – 320 с.

2 Орлов, С. А. Технологии разработки программного обеспечения/ С. А. Орлов – СПб.: Издательский дом «Питер», 2002. – 321 с.

3 Вендров, А.М. Проектирование программного обеспечения экономических информационных систем/ А.М. Вендров – М.: Финансы и статистика, 2005. – 544 с.

4 Вендров, А.М. Практикум по проектированию программного обеспечения/ А.М. Вендров – М.: Финансы и статистика, 2002. – 176 с.

5 Леоненков, А.В. UML/ А.В. Леоненков,– СПб.: БХВ-Петербург, 2001. – 216 с.

6 Благодатских, В.А. Стандартизация разработки программного обеспечения/ В.А. Благодатских [и др.] – М.: Финансы и статистика, 2003. – 329 с.

7 Гома, Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений/ Х. Гома– М.: ДМК Пресс, 2002. – 704 с. : ил.

8 Ларман, К. Применение UML и шаблонов проектирования/ К. Ларман 2-е изд. – М.; СПб.; Киев: Вильямс, 2004. – 624 с.

9 Боггс, У. UML и Rational Rose/ У. Боггс – М.: Лори, 2001. – 581с.

10 Мюллер, Р.Д. Базы данных и UML. Проектирование/ Р.Д. Мюллер – М.: Лори, 2002. – 420 с.

11 Нейбург, Э.Д. Проектирование баз данных с помощью UML. /Э.Д. Нейбург, Р.А. Максимчук. – М.: Вильямс, 2002. – 288 с.

12 Буч Г. Язык UML. Руководство пользователя. /Г. Буч, Д. Рамбо, А. Джекобсон. 2-е изд., – М.: ДМК Пресс; – СПб. : Питер, 2004. – 432 с.

13 Яблочников, Е. И. Использование диаграмм UML для визуального моделирования предметной области при построении АСТПП/ Е. И. Яблочников //Информационные технологии, 2004, № 4. С. 58-62.

14 Мацяшек, Л.А. Анализ требований и проектирование систем. Разработка информационных систем с использованием UML/ Л.А. Мацяшек – М.; СПб.; Киев: Вильямс, 2002. – 432 с.

15 Леоненков, А.В. Объектно – ориентированный анализ и проектирование с использованием UML и IBM Rational Rose / А.В. Леоненков – БИНОМ. Лаборатория знаний, , 2006 – 453 с.

16 Леоненков, А.В. Визуальное моделирование в среде IBM Rational Rose 2003/ А.В. Леоненков,– интернет университет информационных технологий – ИНТУИТ, 2001 ISBN: 978-5-94774-408-2.

Приложение А

(обязательное)

Пример сгенерированного кода на языке программирования C++

```
{Class Diskr_analiz}

//## begin module%1.5%.codegen_version preserve=yes
//   Read the documentation to learn more about C++ code generator
//   versioning.
//## end module%1.5%.codegen_version

//## begin module%470F79570399.cm preserve=no
//   %X% %Q% %Z% %W%
//## end module%470F79570399.cm

//## begin module%470F79570399.cp preserve=no
//## end module%470F79570399.cp

//## Module: Diskr_analiz%470F79570399; Pseudo Package body
//## Source file: C:\Program Files\Rational\Rose\C++\source\Diskr_analiz.cpp

//## begin module%470F79570399.additionalIncludes preserve=no
//## end module%470F79570399.additionalIncludes

//## begin module%470F79570399.includes preserve=yes
//## end module%470F79570399.includes

// Diskr_analiz
#include "Diskr_analiz.h"
//## begin module%470F79570399.additionalDeclarations preserve=yes
//## end module%470F79570399.additionalDeclarations


// Class Diskr_analiz

Diskr_analiz::Diskr_analiz()
  //## begin Diskr_analiz::Diskr_analiz%470F79570399_const.hasinit preserve=no
  //## end Diskr_analiz::Diskr_analiz%470F79570399_const.hasinit
  //## begin Diskr_analiz::Diskr_analiz%470F79570399_const.initialization preserve=yes
  //## end Diskr_analiz::Diskr_analiz%470F79570399_const.initialization
{
  //## begin Diskr_analiz::Diskr_analiz%470F79570399_const.body preserve=yes
  //## end Diskr_analiz::Diskr_analiz%470F79570399_const.body
}

Diskr_analiz::Diskr_analiz(const Diskr_analiz &right)
  //## begin Diskr_analiz::Diskr_analiz%470F79570399_copy.hasinit preserve=no
  //## end Diskr_analiz::Diskr_analiz%470F79570399_copy.hasinit
  //## begin Diskr_analiz::Diskr_analiz%470F79570399_copy.initialization preserve=yes
  //## end Diskr_analiz::Diskr_analiz%470F79570399_copy.initialization
{
  //## begin Diskr_analiz::Diskr_analiz%470F79570399_copy.body preserve=yes
  //## end Diskr_analiz::Diskr_analiz%470F79570399_copy.body
}


Diskr_analiz::~Diskr_analiz()
{
  //## begin Diskr_analiz::~Diskr_analiz%470F79570399_dest.body preserve=yes
  //## end Diskr_analiz::~Diskr_analiz%470F79570399_dest.body
}


Diskr_analiz & Diskr_analiz::operator=(const Diskr_analiz &right)
```

```
{
  //## begin Diskr_analiz::operator=%470F79570399_assign.body preserve=yes
  //## end Diskr_analiz::operator=%470F79570399_assign.body
}


int Diskr_analiz::operator==(const Diskr_analiz &right) const
{
  //## begin Diskr_analiz::operator==%470F79570399_eq.body preserve=yes
  //## end Diskr_analiz::operator==%470F79570399_eq.body
}

int Diskr_analiz::operator!=(const Diskr_analiz &right) const
{
  //## begin Diskr_analiz::operator!=%470F79570399_neq.body preserve=yes
  //## end Diskr_analiz::operator!=%470F79570399_neq.body
}



//## Other Operations (implementation)
void Diskr_analiz::Vector_DF ()
{
  //## begin Diskr_analiz::Vector_DF%470F7D1003D8.body preserve=yes
  //## end Diskr_analiz::Vector_DF%470F7D1003D8.body
}

void Diskr_analiz::Vector_zn_DF ()
{
  //## begin Diskr_analiz::Vector_zn_DF%470F7FB703A9.body preserve=yes
  //## end Diskr_analiz::Vector_zn_DF%470F7FB703A9.body
}

void Diskr_analiz::Const_diskr ()
{
  //## begin Diskr_analiz::Const_diskr%470F800F037A.body preserve=yes
  //## end Diskr_analiz::Const_diskr%470F800F037A.body
}

void Diskr_analiz::Rasch_zn_DF ()
{
  //## begin Diskr_analiz::Rasch_zn_DF%470F802C005D.body preserve=yes
  //## end Diskr_analiz::Rasch_zn_DF%470F802C005D.body
}

// Additional Declarations
  //## begin Diskr_analiz%470F79570399.declarations preserve=yes
  //## end Diskr_analiz%470F79570399.declarations

//## begin module%470F79570399.epilog preserve=yes
//## end module%470F79570399.epilog



{Class Kovar_matrix}

//## begin module%1.5%.codegen_version preserve=yes
//   Read the documentation to learn more about C++ code generator
//   versioning.
//## end module%1.5%.codegen_version

//## begin module%470F792B01A5.cm preserve=no
//   %X% %Q% %Z% %W%
//## end module%470F792B01A5.cm

//## begin module%470F792B01A5.cp preserve=no
//## end module%470F792B01A5.cp

//## Module: Kovar_matrix%470F792B01A5; Pseudo Package body
//## Source file: C:\Program Files\Rational\Rose\C++\source\Kovar_matrix.cpp
```

```
//## begin module%470F792B01A5.additionalIncludes preserve=no
//## end module%470F792B01A5.additionalIncludes

//## begin module%470F792B01A5.includes preserve=yes
//## end module%470F792B01A5.includes

// Kovar_matrix
#include "Kovar_matrix.h"
//## begin module%470F792B01A5.additionalDeclarations preserve=yes
//## end module%470F792B01A5.additionalDeclarations


// Class Kovar_matrix

Kovar_matrix::Kovar_matrix()
  //## begin Kovar_matrix::Kovar_matrix%470F792B01A5_const.hasinit preserve=no
  //## end Kovar_matrix::Kovar_matrix%470F792B01A5_const.hasinit
  //## begin Kovar_matrix::Kovar_matrix%470F792B01A5_const.initialization preserve=yes
  //## end Kovar_matrix::Kovar_matrix%470F792B01A5_const.initialization
{
  //## begin Kovar_matrix::Kovar_matrix%470F792B01A5_const.body preserve=yes
  //## end Kovar_matrix::Kovar_matrix%470F792B01A5_const.body
}

Kovar_matrix::Kovar_matrix(const Kovar_matrix &right)
  //## begin Kovar_matrix::Kovar_matrix%470F792B01A5_copy.hasinit preserve=no
  //## end Kovar_matrix::Kovar_matrix%470F792B01A5_copy.hasinit
  //## begin Kovar_matrix::Kovar_matrix%470F792B01A5_copy.initialization preserve=yes
  //## end Kovar_matrix::Kovar_matrix%470F792B01A5_copy.initialization
{
  //## begin Kovar_matrix::Kovar_matrix%470F792B01A5_copy.body preserve=yes
  //## end Kovar_matrix::Kovar_matrix%470F792B01A5_copy.body
}

Kovar_matrix::Kovar_matrix ()
  //## begin Kovar_matrix::Kovar_matrix%470F7A3501C5.hasinit preserve=no
  //## end Kovar_matrix::Kovar_matrix%470F7A3501C5.hasinit
  //## begin Kovar_matrix::Kovar_matrix%470F7A3501C5.initialization preserve=yes
  //## end Kovar_matrix::Kovar_matrix%470F7A3501C5.initialization
{
  //## begin Kovar_matrix::Kovar_matrix%470F7A3501C5.body preserve=yes
  //## end Kovar_matrix::Kovar_matrix%470F7A3501C5.body
}


Kovar_matrix::~Kovar_matrix()
{
  //## begin Kovar_matrix::~Kovar_matrix%470F792B01A5_dest.body preserve=yes
  //## end Kovar_matrix::~Kovar_matrix%470F792B01A5_dest.body
}


Kovar_matrix & Kovar_matrix::operator=(const Kovar_matrix &right)
{
  //## begin Kovar_matrix::operator=%470F792B01A5_assign.body preserve=yes
  //## end Kovar_matrix::operator=%470F792B01A5_assign.body
}


int Kovar_matrix::operator==(const Kovar_matrix &right) const
{
  //## begin Kovar_matrix::operator==%470F792B01A5_eq.body preserve=yes
  //## end Kovar_matrix::operator==%470F792B01A5_eq.body
}

int Kovar_matrix::operator!=(const Kovar_matrix &right) const
{
  //## begin Kovar_matrix::operator!=%470F792B01A5_neq.body preserve=yes
  //## end Kovar_matrix::operator!=%470F792B01A5_neq.body
```

```
}



//## Other Operations (implementation)
void Kovar_matrix::Sovm_kov_matrix ()
{
  //## begin Kovar_matrix::Sovm_kov_matrix%470F7A5E02EE.body preserve=yes
  //## end Kovar_matrix::Sovm_kov_matrix%470F7A5E02EE.body
}

// Additional Declarations
  //## begin Kovar_matrix%470F792B01A5.declarations preserve=yes
  //## end Kovar_matrix%470F792B01A5.declarations

//## begin module%470F792B01A5.epilog preserve=yes
//## end module%470F792B01A5.epilog

{Class_Matrix}

//## begin module%1.5%.codegen_version preserve=yes
//   Read the documentation to learn more about C++ code generator
//   versioning.
//## end module%1.5%.codegen_version

//## begin module%470F762F032C.cm preserve=no
//   %X% %Q% %Z% %W%
//## end module%470F762F032C.cm

//## begin module%470F762F032C.cp preserve=no
//## end module%470F762F032C.cp

//## Module: Matrix%470F762F032C; Pseudo Package body
//## Source file: C:\Program Files\Rational\Rose\C++\source\Matrix.cpp

//## begin module%470F762F032C.additionalIncludes preserve=no
//## end module%470F762F032C.additionalIncludes

//## begin module%470F762F032C.includes preserve=yes
//## end module%470F762F032C.includes

// Matrix
#include "Matrix.h"
//## begin module%470F762F032C.additionalDeclarations preserve=yes
//## end module%470F762F032C.additionalDeclarations


// Class Matrix

Matrix::Matrix()
  //## begin Matrix::Matrix%470F762F032C_const.hasinit preserve=no
  //## end Matrix::Matrix%470F762F032C_const.hasinit
  //## begin Matrix::Matrix%470F762F032C_const.initialization preserve=yes
  //## end Matrix::Matrix%470F762F032C_const.initialization
{
  //## begin Matrix::Matrix%470F762F032C_const.body preserve=yes
  //## end Matrix::Matrix%470F762F032C_const.body
}

Matrix::Matrix(const Matrix &right)
  //## begin Matrix::Matrix%470F762F032C_copy.hasinit preserve=no
  //## end Matrix::Matrix%470F762F032C_copy.hasinit
  //## begin Matrix::Matrix%470F762F032C_copy.initialization preserve=yes
  //## end Matrix::Matrix%470F762F032C_copy.initialization
{
  //## begin Matrix::Matrix%470F762F032C_copy.body preserve=yes
  //## end Matrix::Matrix%470F762F032C_copy.body
}

Matrix::~Matrix()
```

```
{
  //## begin Matrix::~Matrix%470F762F032C_dest.body preserve=yes
  //## end Matrix::~Matrix%470F762F032C_dest.body
}


Matrix & Matrix::operator=(const Matrix &right)
{
  //## begin Matrix::operator=%470F762F032C_assign.body preserve=yes
  //## end Matrix::operator=%470F762F032C_assign.body
}


int Matrix::operator==(const Matrix &right) const
{
  //## begin Matrix::operator==%470F762F032C_eq.body preserve=yes
  //## end Matrix::operator==%470F762F032C_eq.body
}

int Matrix::operator!=(const Matrix &right) const
{
  //## begin Matrix::operator!=%470F762F032C_neq.body preserve=yes
  //## end Matrix::operator!=%470F762F032C_neq.body
}
//## Other Operations (implementation)
void Matrix::Proizv ()
{
  //## begin Matrix::Proizv%470F7891009C.body preserve=yes
  //## end Matrix::Proizv%470F7891009C.body
}

void Matrix::Standart ()
{
  //## begin Matrix::Standart%470F78AC0242.body preserve=yes
  //## end Matrix::Standart%470F78AC0242.body
}

void Matrix::Obratn ()
{
  //## begin Matrix::Obratn%470F78C3006D.body preserve=yes
  //## end Matrix::Obratn%470F78C3006D.body
}

void Matrix::Transp ()
{
  //## begin Matrix::Transp%470F78D00177.body preserve=yes
  //## end Matrix::Transp%470F78D00177.body
}

// Additional Declarations
  //## begin Matrix%470F762F032C.declarations preserve=yes
  //## end Matrix%470F762F032C.declarations

//## begin module%470F762F032C.epilog preserve=yes
//## end module%470F762F032C.epilog
```

Приложение Б

(обязательное)

Дискриминантный анализ.

Дискриминантный анализ – это совокупность методов, позволяющих решать задачи идентификации объектов по заданному набору характерных признаков.

Весь процесс проведения дискриминантного анализа разбивается на два этапа и каждый из них можно рассматривать как совершенно самостоятельный метод.

Первый этап – выявление и формальное описание различий между существующими множествами (группами) наблюдаемых объектов.

Второй этап – непосредственная классификация новых объектов, т.е. отнесение каждого объекта к одному из существующих множеств.

Пусть имеется множество единиц наблюдения, каждая из которых характеризуется несколькими признаками (переменными): x_{ij} – значения j-й переменной у i-го объекта $\mathrm{i} = \overline{1,\mathrm{n}}$; $j = \overline{1,p}$.

Предположим, что все множество объектов разбито на несколько подмножеств (два и более). Из каждого подмножества взята выборка объемом n_k, где k – номер подмножества (класса) $k = \overline{1,q}$.

Признаки, которые используются для того, чтобы отличать одно подмножество от другого, называются *дискриминантными переменными.*

Число дискриминантных переменных не ограничено, но на практике выбор должен осуществляться на основании логического анализа исходной информации. Число объектов наблюдения должно превышать число дискриминантных переменных, т.е. $\mathrm{p} < \mathrm{n}$. Предполагается, что дискриминантные переменные – линейно независимые нормально распределенные многомерные величины.

Рассмотрим случай для двух дискриминантных переменных. Функция $f(X)$ называется *канонической дискриминантной функцией,* а величины X_1 и X_2 – дискриминантными переменными

$$f(X) = a_1 X_1 + a_2 X_2 . \qquad (\text{Б.1})$$

Дискриминантная функция может быть как линейной, так и нелинейной. Выбор вида этой функции зависит от геометрического расположения разделяемых классов в пространстве дискриминантных переменных.

Коэффициенты дискриминантной функции (a_i) определяются таким образом, чтобы $f_1(X)$ и $f_2(X)$ как можно больше отличались между собой.

Вектор коэффициентов дискриминантной функции (A) определяется по формуле

$$A = S_*^{-1}(\overline{X}_1 - \overline{X}_2). \qquad \text{(Б.2)}$$

Полученные значения коэффициентов подставляют в формулу (1.1) и для каждого объекта в обоих множествах вычисляют дискриминантные функции $f(X)$, затем находят среднее значение для каждой группы ($\overline{f}_k$). Таким образом, каждому i-му наблюдению, которое первоначально описывалось m-переменными, будет соответствовать одно значение дискриминантной функции, и размерность признакового пространства снижается.

Классификация при наличии двух обучающих выборок. Перед тем как приступить непосредственно к процедуре классификации, нужно определить границу, разделяющую два множества. Такой величиной может быть значение функции, равноудаленное от $\overline{f}_1$ и $\overline{f}_2$, т.е.

$$c = \frac{1}{2}(\overline{f}_1 - \overline{f}_2). \qquad \text{(Б.3)}$$

Величина *c* называется *константой дискриминации.*

Объекты, расположенные над разделяющей поверхностью $f(X) = a_1x_1 + a_2x_2 + \cdots + a_px_p = c$ находятся ближе к центру множества M_1, следовательно, могут быть отнесены к первой группе, а объекты, расположенные ниже этой поверхности, ближе к центру второго множества, т.е. относятся ко второй группе. Если граница между группами будет выбрана как сказано выше, то в этом случае суммарная вероятность ошибочной классификации будет минимальной.

Классификация при наличии k-обучающих выборок. Рассмотрим особенности классификации объектов, возникающие при наличии k-обучающих выборок (k>2). Как и в случае с двумя обучающими выборками, предполагается, что каждое множество является нормально распределенным с различными векторами средних значений. Оценка совместной ковариационной матрицы S_* рассчитывается по следующей формуле:

$$S_* = \frac{\sum_{i=1}^{k}(n-1)\cdot S_i}{\sum_{i=1}^{k} n_i}, \qquad \text{(Б.4)}$$

где k – количество обучающих выборок;

S_i – матрица ковариации для i-й выборки;

n_i – численность i-й выборки.

В этом случае каждому множеству ставится в соответствие своя дискриминантная функция вида

$$f_i = a_{0i} + a_{1i}x_1 + a_{2i}x_2 + \cdots + a_{mi}x_m .$$

Вектор коэффициентов этой функции a_{ij} ($j = \overline{1,m}$) рассчитывается по формуле $a_{ij} = \overline{X}_i \cdot S_*^{-1}$, а свободный член $a_{0i} = -\frac{1}{2}\overline{X}_i \cdot S_*^{-1}\overline{X}_i$.

Новый классифицируемый объект с переменными $Y_1, Y_2, \ldots, Y_m$ будет отнесен к тому множеству M_i, для которого величина $f_i = c_i + a_{1i}Y_1 + a_{2i}Y_2 + \cdots + a_{mi}Y_m$ будет максимальной.

В заключение необходимо отметить, что в данном приложении рассмотрен только один из методов проведения дискриминантного анализа. Более подробно другие методы и алгоритмы дискриминантного анализа описаны в специальной литературе.