МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

КАФЕДРА БЕЗОПАСНОСТИ ИНФОРМАЦИОННЫХ СИСТЕМ

А.Н.Крутов

МЕТОДЫ ПРОГРАММИРОВАНИЯ. ООП. UML. RUP

Учебное пособие

для студентов
механико-математического факультета
специальности 075300 – «Организация и технология
защиты информации»

Издательство «Самарский университет»
2004

Печатается по решению Редакционно-издательского совета Самарского государственного университета по решению

УДК 519.683
ББК 32.973-01
К 846

Крутов А.Н. Методы программирования. ООП. UML. RUP. Учебное пособие. Самара: Изд-во «Самарский университет», 2004. 116 с.

Основой данного учебного пособия являются материалы лекций, прочитанных автором по курсу "Методы программирования" для специальности 075300– «Организация и технология защиты информации».

В пособии кратко рассматриваются современные методы разработки программного обеспечения.

Предназначено для студентов механико-математического факультета Самарского государственного университета специальности «Организация и технология защиты информации».

УДК 519.683
ББК 32.973-01

Рецензент канд. техн. наук, доцент Ю.Е.Ефимов

© Крутов А.Н., 2004

© Изд-во «Самарский университет», 2004

СОДЕРЖАНИЕ

Введение.....4

1. Сложность современных программных систем.....5

2. Объектная модель.....10

Эволюция объектной модели.....10
Составные части объектного подхода.....12
Применение объектной модели.....21

3. Классы и объекты.....22

Природа объекта.....22
Отношения между объектами.....24
Природа классов.....25
Отношения между классами.....26
Взаимосвязь классов и объектов.....30

4. Основы языка UML.....32

Принципы моделирования.....34
Введение в язык UML.....34
Концептуальная модель UML.....35
Основы структурного моделирования.....41
Диаграммы.....52

5. Введение в RUP.....71

Итеративная разработка программного обеспечения.....76
Управление требованиями.....78
Архитектура приложения.....79
Поток работ “Управление проектом”.....84

6. Задачи для лабораторных работ.....95

Заключение.....113

Библиографический список.....114

ВВЕДЕНИЕ

В настоящее время уровень информатизации общества достиг таких масштабов, что методы создания программных продуктов, разработанные всего несколько десятилетий назад уже не в состоянии удовлетворять все возрастающему набору требований со стороны как разработчиков, так и пользователей. Для разработчика важно создание такого кода, который будет оптимальным для текущей платформы и который будет относительно легко поддерживать, т.е. оперативно исправлять все обнаруженные ошибки и добавлять в код новую функциональность. Для пользователя в настоящее время важно активнее принимать участие в процессе создания программного продукта, так как очень часто набор его требований может меняться в процессе разработки. Этому посвящено данное пособие.

В силу специфики специальности 075300 – Организация и технология защиты информации, настоящее пособие содержит информацию преимущественно по проектированию архитектуры программных продуктов, а также грамотному ведению соответствующих проектов.

Первая глава является, по существу, вводной для настоящего пособия. В ней описываются основные проблемы, которые возникают при разработке современных программных продуктов и описываются методы их решения. Вторая глава посвящена описанию основных элементов и принципов объектной модели, а также объектно-ориентированного проектирования как относительно нового способа разработки программных продуктов. В третьей главе эта идея развивается далее. В ней вводятся понятия класса и объекта и рассматривается их взаимосвязь между собой. Четвертая глава посвящена описанию языка UML, с помощью которого возможно комплексная разработка архитектуры приложения, а также более грамотная поддержка и развитие уже существующей объектной модели. Пятая глава посвящена краткому описанию технологии RUP, с помощью которой можно производить комплексную разработку программных продуктов, начиная со стадии выяснения требований и заканчивая сдачей программы в промышленную эксплуатацию. В конце пособия по итогам всех приведенных тем предлагается список задач для лабораторных работ по курсу.

При подготовке данного методического пособия активно использовалась литература, приведенная в библиографическом списке. Данный список намеренно не содержит слишком большое количество монографий, чтобы акцентировать внимание студентов на самых главных из них.

1. СЛОЖНОСТЬ СОВРЕМЕННЫХ ПРОГРАММНЫХ СИСТЕМ

В цикле лекций, изложенных в настоящем пособии преимущественно рассматривается процесс разработки современных промышленных программных продуктов. Существенная черта промышленной программы - уровень сложности: один разработчик практически не в состоянии охватить все аспекты такой системы. Грубо говоря, сложность промышленных программ превышает возможности человеческого интеллекта. Увы, но подобная сложность присуща всем большим программных системам. Т.е. с ней можно справиться, но избавиться от нее нельзя. Поэтому в настоящем курсе необходимо рассмотреть более надежные способы конструирования сложных систем. Для лучшего понимания того, чем мы собираемся управлять, сначала ответим на вопрос: почему сложность присуща всем большим программным системам ?

Сложность программного обеспечения - отнюдь не случайное его свойство.
Она вызывается четырьмя основными причинами:

- трудностью реальной предметной области, из которой, исходит заказ на разработку
- трудностью управления процессом разработки
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания поведения больших дискретных систем.

Сложность реального мира

Проблемы, которые решаются с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований. Кроме этого сложность обычно возникает из-за "нестыковки" между пользователями системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения.

Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему.

Большая программная система - это крупное капиталовложение, поэтому нельзя позволить себе выкидывать сделанное при каждом изменении внешних требований. Тем не менее даже большие системы имеют тенденцию к эволюции в процессе использования.

Трудности управления процессом разработки

Основная задача разработчиков состоит в создании иллюзии простоты для пользователя, его защите от сложности описываемого предмета или процесса. Размер исходных текстов программы отнюдь не входит в число ее главных достоинств, поэтому необходимо делать исходные тексты более компактными, изобретая хитроумные и мощные методы, а также используя среды разработки уже существующих проектов и программ. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы она ни была, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

Гибкость программного обеспечения

Программирование обладает предельной гибкостью и разработчик при правильной организации рабочего процесса может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Возможно создание своими силами всех базовых строительных блоков будущей конструкции, из которых составляются элементы более высоких уровней абстракции, которые возможно применить при решении задач из различных предметных областей. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. Поэтому программные разработки остаются очень трудоемким, но достаточно творческим делом.

Проблема описания поведения больших дискретных систем

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений описывает состояние прикладной программы в каждый момент времени. Ввиду того, что исполнение программы осуществляется на цифровом компьютере, то соответствующий процесс является дискретным. Поэтому возможны сложности при описании программными средствами непрерывных процессов, которые приходится реализовывать с помощью дискретных методов. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Это, является главной причиной обязательного тестирования систем не только путем исследования самых тривиальных случаев, но и путем дополнительных исследований в данной предметной области.

Признаки сложных систем:

1. Сложные системы, как правило, являются иерархическими и состоят из взаимозависимых подсистем
2. Выбор компонент, которые считаются элементарными, является относительно произвольным и оставляется на усмотрение исследователя
3. Внутрикомпонентные связи в сложных системах обычно сильнее, чем связи между компонентами
4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных
5. Любая работающая сложная система является результатом развития работавшей более простой системы

Как показывает практика, наиболее успешны те программные системы, в которые заложены хорошо продуманные структуры классов и объектов и которые обладают пятью признаками сложных систем, описанными выше. Очень редко можно встретить программную систему, разработанную точно по графику, уложившуюся в бюджет и удовлетворяющую требованиям заказчика, в которой бы не были учтены соображения, изложенные выше. В дальнейшем структуры классов и объектов системы будут называться **архитектурой системы**.

Человеческие возможности и сложные системы

Главной причиной, почему при разработке сложных программных систем возникают серьезные проблемы является физическая ограничен-

ность возможностей человека. Когда происходит процесс проектирования новой системы, необходимо думать сразу о многом. К сожалению, один человек не может следить за всем этим одновременно. Эксперименты психологов показывают, что максимальное количество структур единиц информации, за которыми человеческий мозг может одновременно следить приблизительно равно семи плюс-минус два. Таким образом имеется серьезная проблема. Сложность программных систем возрастает, но способность нашего мозга справиться с этой сложностью ограничена. Помочь человеку разобраться с данной проблемой призваны три метода: декомпозиция, абстракция и иерархия.

Роль декомпозиции

На самом деле, способ управления сложными системами был известен еще в древности. Он гласит "разделяй и властвуй". При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае пропускная способность человеческого мозга не превышается: для понимания любого уровня системы в данном случае необходимо одновременно держать в уме информацию лишь о немногих ее частях. Выделяют два типа декомпозиции: **алгоритмическая и объектно-ориентированная**.

Алгоритмическая декомпозиция

Алгоритмической декомпозицией является структурное проектирование "сверху вниз". Оно заключается в разделении алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса. При данном виде декомпозиции решение большой задачи выражается через решения ряда более простых задач. Если для решения какой-то мелкой подзадачи писалась процедура, к ней прилагается спецификация, объясняющая ее смысл и формат запуска так, чтобы обращение к ней могло происходить без знания внутренней структуры этой процедуры. Этот метод работал хорошо до тех пор, пока не были предприняты первые попытки создания систем, где число таких подпрограмм исчислялось тысячами. Никакие спецификации не помогут даже самому одаренному программисту запомнить тысячи подпрограмм.

Объектно-ориентированная декомпозиция

Данный вид декомпозиции основан на введении в качестве критерия декомпозиции принадлежность ее элементов (объектов) к различным абстракциям данной проблемной области и определения связей между элемен-

тами. Таким образом, каждый объект обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует вполне определенное поведение. Объекты реализуют какие-то методы, им можно послать сообщение, запросив какой-либо набор данных, выполнить определенный набор процедур и т.п.

Ответить на вопрос какой вид декомпозиции лучше а какой хуже нельзя, так как они представляют собой различные точки зрения на решение одной и той же задачи и только их комбинация может привести к положительным результатам. Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия. Однако конструирование сложной системы одновременно двумя способами все же невозможно, поскольку эти способы по сути ортогональны. Поэтому на начальном этапе необходимо начать разработку в рамках одной идеологии, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения.

Опыт показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться с приданием организованности сложности программных систем.

Объектная декомпозиция имеет несколько чрезвычайно важных преимуществ перед алгоритмической. Объектная декомпозиция уменьшает размер программных систем за счет повторного использования общих механизмов, что приводит к существенной экономии средств. Объектно-ориентированные системы более гибки и проще эволюционируют со временем, потому что их схемы базируется на устойчивых формах. Действительно, объектная декомпозиция существенно снижает риск при создании сложной программной системы, так как в данном случае она развивается из меньших систем, которые были уже реализованы и отлажены ранее. Более того, объектная декомпозиция помогает разобраться в сложной программной системе.

Роль абстракции

Как было сказано ранее, человек может одновременно воспринять достаточно ограниченное количество информации. Будучи не в состоянии полностью воссоздать сложный объект, производится выделение из него наиболее важных с точки зрения исследователя свойств, а остальные просто игнорируются. Таким образом получается некоторая обобщенная, идеализированная модель объекта, с которой непосредственно происходит работа. Это особенно верно, когда мир рассматривается с объектно-ориентированной точки зрения, поскольку объекты как абстракции реаль-

ного мира представляют собой отдельные насыщенные связные информационные единицы.

Роль иерархии

Другим способом, расширяющим информационные единицы, является организация внутри системы иерархий классов и объектов. Объектная структура важна, так как она иллюстрирует схему взаимодействия объектов друг с другом, которое осуществляется с помощью механизмов взаимодействия. Структура классов не менее важна: она определяет общность структур и поведения внутри системы. Классифицируя объекты по группам родственных абстракций, происходит четкое разделение общих и уникальных свойств разных объектов, что помогает справляться со свойственной им сложностью. Определить иерархии в сложной программной системе не всегда легко, так как это требует разработки моделей многих объектов, поведение каждого из которых может отличаться чрезвычайной сложностью. Однако после их определения, структура сложной системы и, в свою очередь, ее понимание во многом проясняются.

2. ОБЪЕКТНАЯ МОДЕЛЬ

Эволюция объектной модели

Поколения языков программирования

Ниже представлена группировка наиболее известных языков высокого уровня в четыре поколения в зависимости от того, какие языковые конструкции в них появились.

Первое поколение (1954-1958)

Fortran I, ALGOL-58, Flowmatic, IPL V (математические формулы)

Для языков данного поколения основным строительным блоком является подпрограмма. Программы имеют относительно простую структуру и состоят только из глобальных данных и подпрограмм. Механизмами языков практически не поддерживаются логически разнотипные данные.

Второе поколение (1959-1961)

FORTRAN II (подпрограммы, раздельная компиляция), ALGOL-60 (блочная структура, типы данных), COBOL (описание файлов, работа с данными), Lisp (обработка списков, указатели, сборка мусора)

Для языков второго поколения характерно начало использования подпрограмм как механизма абстрагирования, что привело к разработке языков, поддерживающих разнообразные механизмы передачи параметров, а также заложению основ структурного программирования.

Третье поколение (1962-1970)

PL/1, ALGOL-68, Pascal, Simula (классы, абстрактные данные)

Создание технологии модульного программирования. Однако защита данных обеспечивается слабо, т.к. поддержка абстрактных данных находится лишь в зачаточном состоянии.

Четвертое поколение (1970-1980)

C++, Object Pascal, Ada (объекты, классы)

Основным элементом конструкции языков данного поколения является модуль, а не подпрограмма, как это было ранее.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (OOP) - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Объектно-ориентированное проектирование

Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование, напротив, основное внимание уделяет правильному и эффективному структурированию сложных систем.

Объектно-ориентированное проектирование (OOD) - это методология проектирования, соединяющая в себе процесс объектной декомпози-

ции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Объектно-ориентированный анализ

Объектно-ориентированный анализ направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ (ООА) - это методология, при которой требования к системе воспринимаются сточки зрения классов и объектов, выявленных в предметной области.

Как соотносятся ООА, OOD и ООР? На результатах ООА формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии ООР.

Составные части объектного подхода

Для объектно-ориентированного стиля программирования концептуальной базой является объектная модель, которая имеет четыре главных свойства:

- абстрагирование
- инкапсуляция
- модульность
- иерархия

а также три дополнительных:

- типизация
- параллелизм
- сохраняемость

Абстрагирование

Абстрагирование является одним из основных методов, используемых для решения сложных задач. Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объек-

тов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя. Выбор правильного набора абстракций представляет собой главную задачу объектно-ориентированного проектирования. Выделяют четыре вида абстракции, а именно: абстракция сущности, абстракция поведения, абстракция виртуальной машины и произвольная абстракция.

Для объектно-ориентированного проектирования наиболее ценными являются абстракции сущности, так как они соответствуют сущностям предметной области.

Примеры абстракций

В данном разделе приводится несколько примеров. Однако внимание концентрируется не столько на выделении абстракций, сколько на способе их выражения. Приводимый в данном разделе код будет впоследствии модифицироваться при рассмотрении дальнейших свойств объектной модели.

Например, требуется написать простейший графический редактор. В настоящем пособии последовательно разрабатывается объектная модель именно этой задачи. Предварительно проанализировав поставленное задание решается, что одной из ключевых абстракций в рамках данной задачи будет прямая. Будем считать, что работа происходит на плоскости. Ввиду того, что любая прямая может иметь различный цвет, ширину линии и т.п., дополнительно объявляется тип Draw_Style:

```
Draw_Style=record
  Pen_Color:TColor;
  Pen_Width:integer;
  Brush_Color:TColor;
  Pen_Style:TPenstyle;
  Brush_Style:TBrushstyle
end;

TMyLine=class
Public
  Object_Attribute:Draw_Style;
  Point1,Point2: TPoint;
  constructor Create;
  destructor Destroy;
  procedure Draw;
end;
```

Инкапсуляция

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать конкретные абстракции от их реализации.

Обращаясь в примеру создания программы графического редактора следует отметить, что в описанной выше структуре класса TMyLine наблюдаются некоторые недостатки. Так, совсем не обязательно давать доступ к переменным Point1, Point2 и Object_Attribute. Представляется лучшим поместить эти переменные в секцию protected, а доступом к соответствующим значениями организовать через специальные свойства:

```
TMyLine=class
protected
  Object_Attribute:Draw_Style;
  vPoint1,vPoint2: TPoint;
  function GetObject_Attribute: Draw_Style;
  procedure SetObject_Attribute(CurObject_Attribute:
                                              Draw_Style);
  function GetPoint1: TPoint;
  procedure SetPoint1 (CurPoint1: TPoint);
  function GetPoint2: TPoint;
  procedure SetPoint2 (CurPoint2: TPoint);
public
  constructor Create;
  destructor Destroy;
  procedure Draw;
  property Object_Attribute: Draw_Style
                read GetObject_Attribute write SetObject_Attribute;
  property Point1: TPoint read GetPoint1 write SetPoint1;
  property Point2: TPoint read GetPoint2 write SetPoint2;
end;
```

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта и реализация его методов.

Сочетание абстракции и инкапсуляции практически означает наличие двух частей в классе: интерфейса и реализации. Интерфейс отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя реализация описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия объекта с

любыми другими объектами; реализация скрывает от всех детали, не имеющие отношения к процессу взаимодействия.

Разумная инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать алгоритм по критерию памяти, заменяя хранение данных вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов. На практике же иногда просто необходимо ознакомиться с реализацией класса, чтобы понять его назначение, особенно, если нет внешней документации.

Модульность

Модульность – разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи между модулями.

Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность. Однако гораздо важнее тот факт, что внутри модульной программы создается множество хорошо определенных и документированных интерфейсов. Эти интерфейсы необходимы для исчерпывающего понимания программы в целом.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми. При коллективной разработке программ распределение работы осуществляется как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. При этом более опытные программисты отвечают за интерфейс модулей, а менее опытные за реализацию.

Иерархия

Иерархия - это упорядочение абстракций, расположение их по уровням.

Пример иерархии: одиночное наследование. Важным элементом объектно-ориентированных систем и основным видом иерархии является именно данный тип наследования. Вообще, наследование означает такое отношение родитель/потомок, когда один класс заимствует структурную и

функциональную часть одного или нескольких других классов (соответственно, одиночное и множественное наследование). Иными словами, наследование означает такую иерархию абстракций, в которой подклассы наследуют строение нескольких суперклассов. Часто подкласс достраивает или переписывает компоненты вышестоящего класса.

Возвращаясь к задаче о создании графического редактора, можно несколько расширить список рассматриваемых объектов. Так, кроме объекта TMyLine можно ввести объект прямоугольник, TMyRectangle, окружность TMyCircle и ряд других. Если писать структуру данных объектов по аналогии с TMyLine, то получатся следующие конструкции:

```
TMyRectangle=class
protected
  Object_Attribute:Draw_Style;
  vPoint1,vPoint2: TPoint; // координаты левой нижней и правой верхней вершин
  function GetObject_Attribute: Draw_Style;
  procedure SetObject_Attribute(CurObject_Attribute: Draw_Style);
  function GetPoint1: TPoint;
  procedure SetPoint1(CurPoint1: TPoint);
  function GetPoint2: TPoint;
  procedure SetPoint2(CurPoint2: TPoint);
public
  constructor Create;
  destructor Destroy;
  procedure Draw;
  property Object_Attribute: Draw_Style read GetObject_Attribute
                                         write SetObject_Attribute;
  property Point1: TPoint read GetPoint1 write SetPoint1;
  property Point2: TPoint read GetPoint2 write SetPoint2;
end;

TMyCircle=class
protected
  Object_Attribute:Draw_Style;
  vCenter: TPoint; // координаты центра окружности
  VRadius:integer;
  function GetObject_Attribute: Draw_Style;
  procedure SetObject_Attribute(CurObject_Attribute: Draw_Style);
  function GetCenter:TPoint;
  procedure SetCenter(CurCenter:TPoint);
  function GetRadius:integer;
  procedure SetRadius (CurRadius:integer);
public
  constructor Create;
  destructor Destroy;
  procedure Draw;
  property Object_Attribute: Draw_Style read GetObject_Attribute
                                         write SetObject_Attribute;
```

```
  property Center: TPoint read GetCenter write SetCenter;
  property Radius: integer read GetRadius write SetRadius;
end;
```

Нетрудно видеть, что структура объектов TMyLine и TMyRectangle практически совпадает. Кроме этого все объекты имеют конструктор, деструктор, переменную настройки отображения Object_Attribute и метод Draw. Поэтому логично будет ввести в качестве базового объекта некий объект с абстрактными методами, от которого будут наследоваться все объекты:

```
TMyBaseGraphClass=class
protected
  Object_Attribute:Draw_Style;
  function GetObject_Attribute: Draw_Style;
  procedure SetObject_Attribute(CurObject_Attribute: Draw_Style);
public
  constructor Create;
  destructor Destroy;
  procedure Draw; virtual; abstract;
  property Object_Attribute: Draw_Style read GetObject_Attribute
                                        write SetObject_Attribute;
end;


TMyLine=class(TMyBaseGraphClass)
protected
  vPoint1,vPoint2: TPoint;
  function GetPoint1: TPoint;
  procedure SetPoint1 (CurPoint1: TPoint);
  function GetPoint2: TPoint;
  procedure SetPoint2 (CurPoint2: TPoint);
public
  constructor Create; override;
  destructor Destroy; override;
  procedure Draw; override;
  property Point1: TPoint read GetPoint1 write SetPoint1;
  property Point2: TPoint read GetPoint2 write SetPoint2;
end;

TMyRectangle=class(TMyLine)
public
  procedure Draw; override;
end;

TMyCircle=class(TMyBaseGraphClass)
protected
  vCenter: TPoint; // координаты центра окружности
  VRadius:integer;
```

```
  function GetCenter:TPoint;
  procedure SetCenter(CurCenter: TPoint);
  function GetRadius:integer;
  procedure SetRadius(CurRadius:integer);
public
  constructor Create; override;
  destructor Destroy; override;
  procedure Draw; override;
  property Center: TPoint read GetCenter write SetCenter;
  property Radius: integer read GetRadius write SetRadius;
end;
```

В наследственной иерархии общая часть структуры и поведения сосредоточены в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии обобщение-специализация. Суперклассы при этом отражают наиболее общие, а подклассы- более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты. В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля". Классы лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов.

Пример иерархии: множественное наследование

Данный тип иерархии отличается от предыдущего тем, что в данном случае происходит наследование не от одного суперкласса, а от нескольких сразу. Однако при таком типе наследования возникают две проблемы - конфликты имен между разными суперклассами и повторное наследование. Первый случай, это когда в двух или большем числе суперклассов определено поле или операция с одинаковым именем. В этом случае в различных языках, поддерживающих множественное наследование, эта проблема решается по-разному. В C++ этот вид конфликта должен быть явно разрешен вручную, а в Smalltalk берется то, которое встречается первым. Проблема с повторным наследованием возникает тогда, когда класс наследует двум классам, а они порознь наследуют одному и тому же четвертому. Получается ромбическая структура наследования и надо решить, должен ли самый нижний класс получить одну или две отдельные копии самого верхнего класса? В некоторых языках повторное наследование запрещено, к которым относится, в частности, язык Object Pascal.

Примеры иерархии: агрегация

Для иллюстрации данного типа иерархии рассмотрим пример. Пусть все объекты, которые были объявлены предназначены для рисования на какой-либо форме. Тогда для корректной работы с ними необходимо сделать так, чтобы сама форма имела список всех построенных на текущий момент графических объектов:

```
type
 TMyForm = class(TForm)
 private
   GraphArray: array of TMyBaseGraphClass;
 end;
```

Реализованный таким образом объект TMyForm является примером иерархии типа агрегации. Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

Типизация

Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

Идея согласования типов занимает в понятии типизации центральное место. Например, возьмем физические единицы измерения. Деля расстояние на время мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу - есть. Все это примеры сильной типизация, когда прикладная область накладывает правила и ограничения на использование и сочетание абстракций.

Так, например, абсолютно абсурдно выглядит попытка присвоить переменные совершенно разных классов:

```
var p:TMyForm;
    s:TMyLine;
begin
  ...
  p:=s;
end;
```

Для объектов, находящихся в родственных отношениях подобная операция присваивания допускается. Однако она разрешена только в од-

ном направлении, а именно, переменной класса-предка можно присваивать значение переменной класса-наследника. Т.е. если есть два класса a и b:

```
a=class
...
end;

b=class(a)
...
end;

var p1:a;
   p2:b;
```

то допустимо присваивание a:=b и недопустима операция b:=a;

Параллелизм

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере.

Процесс (поток управления) - это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система имеет много таких потоков: век одних недолог, а другие живут в течении всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени. В то время, как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов.

Сохраняемость

Каждый программный объект существует в памяти и живет во времени. Так, существуют объекты, которые присутствуют лишь во время вычисления выражения, но есть и такие, как базы данных, которые существуют независимо от программы. Этот спектр сохраняемости объектов охватывает:

- Промежуточные результаты вычисления выражений.
- Локальные переменные в вызове процедур.
- Глобальные переменные и динамически создаваемые данные.
- Данные, сохраняющиеся между сеансами выполнения программы.
- Данные, сохраняемые при переходе на новую версию программы.
- Данные, которые способны пережить программу, работающую с ними

Традиционно, первыми тремя уровнями занимаются языки программирования, а последними - базы данных.

Сохраняемость - способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

Применение объектной модели

Как уже говорилось выше, объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Однако это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Объектный подход обеспечивает ряд существенных удобств, которые другими моделями не предусматривались. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем. Согласно нашему опыту, есть еще пять преимуществ, которые дает объектная модель.

Во-первых, объектная модель позволяет в полной мере использовать **выразительные возможности** объектных и объектно-ориентированных языков программирования. К ним относятся работа с абстракциями данных и введение иерархии классов в процессе проектирования.

Во-вторых, использование объектного подхода существенно **повышает уровень унификации разработки** и пригодность для повторного использования не только программ, но и проектов, что в конце концов ведет к созданию среды разработки. Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени.

В-третьих, использование объектной модели приводит к **построению систем на основе стабильных промежуточных описаний**, что упрощает

процесс внесения в них изменений. Это дает системе возможность развиваться постепенно и не приводить к полной ее переработке даже в случае существенных изменений исходных требований.

В-четвертых, объектная модель **уменьшает риск разработки** сложных систем, прежде всего потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

И, наконец, **в-пятых**, объектная модель ориентирована на человеческое восприятие мира

Использование объектного подхода

Возможность применения объектного подхода доказана для задач самого разного характера. В настоящее время объектно-ориентированное проектирование - единственная методология, позволяющая справиться со сложностью, присущей очень большим системам. Однако, следует заметить, что иногда применение OOD может оказаться нецелесообразным, например, из-за неподготовленности персонала или отсутствия подходящих средств разработки.

3. КЛАССЫ И ОБЪЕКТЫ

Природа объекта

Объект – это конкретный опознаваемый предмет, единица или сущность (реальная или абстрактная), имеющий четко определенное функциональное назначение в данной предметной области.

Объект обладает ***состоянием***, ***поведением*** и ***идентичностью***; структура и поведение схожих объектов определяет общий для них класс; термины "экземпляр класса" и "объект" взаимозаменяемы.

Состояние

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Перечень свойств объекта является, как правило, ста-

тическим, поскольку эти свойства составляют неизменяемую основу объекта. Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект.

Тот факт что всякий объект имеет состояние, означает, что он занимает определенное пространство (физически или в памяти компьютера).

Поведение

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение - это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Иными словами, поведение объекта - это его наблюдаемая и проверяемая извне деятельность.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию.

Выделяют три наиболее распространенные операции:

- Модификатор — Операция, которая изменяет состояние объекта.
- Селектор — Операция, считывающая состояние объекта, но не меняющая состояния.
- Итератор — Операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности

Кроме этого, в любом объекте существуют две универсальные операции; они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса:

- Конструктор — Операция создания объекта и/или его инициализации
- Деструктор — Операция, освобождающая состояние объекта и/или разрушающая сам объект.

Идентичность

Идентичность - это такое свойство объекта, которое отличает его от всех других объектов.

Так, например, если имеются два экземпляра класса TMyLine

```
var p1,p2:TMyLine;

p1:= TmyLine.Create;
...
p2:= TmyLine.Create;
...
```

которые создаются и заполняются значениями, то они всегда будут различными объектами, даже если будут иметь совершенно одинаковые атрибуты. Если же записать следующим образом:

```
var p1,p2:TMyLine;

p1:= TmyLine.Create;
...
p2:=p1;
```

то в этом случае объекты p1 и p2 будут полностью идентичными, так как они ссылаются на одну и ту же область памяти.

Отношения между объектами

Сами по себе объекты не представляют никакого интереса: только в процессе взаимодействия объектов реализуется система. Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять и об ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа иерархических соотношений объектов: **связи и агрегация**

Связи

Связь - это специфическое сопоставление, через которое объект-клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

Участвуя в связи, объект может выполнять одну из следующих трех ролей

- Актер — Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов
- Сервер — Объект может только подвергаться воздействию со стороны других объектов, но он никогда не

	выступает в роли воздействующего объекта
• Агент	Объект может выступать как в активной, так и в Пассивной роли; как правило, объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или агента.

Видимость

Предположим, что существуют два объекта А и В и связь между ними. Чтобы объект А мог послать сообщение объекту В, необходимо, чтобы данный объект был в каком-то смысле видим для А.

Существуют четыре способа обеспечить видимость:

- Сервер глобален по отношению к клиенту.
- Сервер (или указатель на него) передан клиенту в качестве параметра операции.
- Сервер является частью клиента.
- Сервер локально порождается клиентом в ходе выполнения какой-либо операции.

Какой именно из этих способов выбрать - зависит от тактики проектирования.

Агрегация

В то время, как связи обозначают равноправные или "клиент-серверные" отношения между объектами, агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем придти к; его частям (атрибутам). В этом смысле агрегация - специализированный частный случай ассоциации.

Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.

Выбирая одно из двух - связь или агрегацию - надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целое. Иногда наоборот предпочтительнее связи, поскольку они менее ограниченные.

Природа классов

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает

конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.

Класс - это некое множество объектов, имеющих общую структуру и общее поведение.

Любой конкретный объект является просто экземпляром класса.

Отношения между классами

Известны три основных типа отношений между классами:

1. Отношение "**обобщение/специализация**" (общее и частное).
2. Отношение "**целое/часть**", известное как "part of".
3. **Семантические, смысловые отношения**, ассоциации.

Языки программирования выработали несколько общих подходов к выражению отношений этих трех типов. В частности, большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:

- ассоциация;
- наследование
- агрегация
- использование
- инстанцирование
- метакласс.

Далее рассматриваются данные типы отношений более подробно.

Ассоциация

Пример. Предположим, что необходимо разработать программу по учету успеваемости студентов. На этапе разработки структуры классов можно выделить две абстракции – студенты и способы оперативной связи с ними. К данным способам, например, можно отнести список адресов электронной почты. При разработке системы можно предусмотреть общий случай, согласно которому у каждого студента может быть несколько e-mail адресов. Соответствующая структура классов на языке Object Pascal имеет следующий вид:

```
type
  TStudent=class;

  TEmail=class
```

```
    AddressName:string;
    Student:TStudent;
  end;

  TStudent=class
    ...
    EMailArray:array of TEmail
  end;
```

Это ассоциация вида "**один-ко-многим**": каждый конкретный адрес электронной почты относится только к одному студенту, в то время как каждый студент может указывать на совокупность e-mail адресов.

Как показывает этот пример, **ассоциация - смысловая связь**. По умолчанию, она ***не имеет направления*** (если не оговорено противное, ассоциация, как в данном примере, подразумевает двухстороннюю связь) и не объясняет, как классы общаются друг с другом. Однако именно это требуется на ранней стадии анализа. В этом случае фиксируются участники, их роли и мощность отношения.

Мощность. В предыдущем примере была приведена ассоциация "один ко многим". Тем самым обознается ее мощность (то есть, грубо говоря, количество участников).

На практике важно различать три случая мощности ассоциации:

- "один-к-одному"
- "один-ко-многим"
- "многие-ко-многим".

Отношение "один-к-одному" обозначает очень узкую ассоциацию. Отношение "многие-ко-многим" на практике встречается, пожалуй, наиболее часто. Так, например, студент, обучаясь в какой-нибудь группе, на протяжении обучения прослушивает курс лекций по определенным научным дисциплинам. Однако эти же научные дисциплины (все или некоторые из них), могут читаться и другим группам. Обычно соотношение "многие-ко-многим" осуществляется с помощью совокупности трех классов. В данном случае это будут классы Группы, Научные дисциплины и класс, осуществляющий связь между конкретной группой и соответствующим набором научных дисциплин.

```
type
  TGroups=class
    GroupName:string;
  end;

  TLessons=class
    LessonName:string;
  end;
```

```
TGroupsLessons=class
  Groups: TGroups;
  Lessons: TLessons
end;
```

Таким образом, если где-то в программе хранится список всех элементов типа TGroupsLessons, то всегда можно будет ответить на вопрос каким группам читается тот или иной предмет и какие предметы читаются той или иной группе.

Наследование

Наследование - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Класс, структура и поведение которого наследуются, называется суперклассом.

Самый общий класс в иерархии классов называется **базовым**. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие абстракции предметной области. На самом деле, хорошо сделанная структура классов - это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем. В некоторых языках программирования определен базовый класс самого верхнего уровня, который является единым суперклассом для всех остальных классов. В языке Object Pascal эту роль играет класс TObject.

Полиморфизм

В сочетании с механизмом наследования полиморфизм представляет широкие возможности при разработки программ с использованием объектного подхода. Так, ранее был приведен фрагмент кода программы графического редактора, в котором была описана структура классов. Тип TMyRectangle наследует поля Point1 и Point2 из своего непосредственного родителя TMyLine, а также поля и методы из типа TMyBaseGraphClass, который также считается (косвенным) предком для TMyRectangle. Длина такой цепочки наследования в языке никак не ограничивается. По правилу наследования тип TMyRectangle имеет в своём составе методы объекта-предка TMyLine. Однако, легко видеть, что метод Draw не подходит для рисования прямоугольника. Поэтому, полное описание типа TMyRectangle должно содержать также собственные методы для его рисования. Самым простым решением было бы ввести в новый тип такие методы, дав им некоторые новые имена.

Но объектно-ориентированный подход позволяет определить новые методы **СО СТАРЫМИ** именами, **ПЕРЕОПРЕДЕЛИВ** тем самым методы типа-родителя.

Следует отметить, что **переопределять можно только методы**; поля, указанные в родительском типе, безусловно наследуются типом-потомком и не могут быть в нём переопределены (то есть имена полей потомка не должны совпадать с именами полей типа-предка). Кроме того, новый метод в типе потомке может иметь совершенно другие параметры. нежсли одноимённый метод из типа-предка.

Возможность иметь в одной программе несколько подпрограмм-методов для разных объектов называется в ООП **полиморфизмом**, и является одним из основных свойств ООП.

Агрегация

Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами.

Использование

Отношение **использования** между классами соответствует равноправной связи между их экземплярами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера).

На самом деле, один класс может использовать другой по-разному. Это может происходить через интерфейсную функцию. Однако в типичном случае такое отношение использования проявляет себя, если в какой-либо операции происходит объявление локального объекта используемого класса.

Инстанцирование

Понятие инстанцирования классов характерно для таких объектно-ориентированных языков программирования как С++. В этом языке введено понятие **шаблон**.

Шаблон – это механизм генерирования типов. Из определения шаблона класса и набора аргументов шаблона компилятор генерирует специализацию класса и его используемых методов. Этот процесс называется ***инстанцированием***.

Таким образом, определив тип вектор с использованием шаблонов, в программе появляется возможность задавать вектора целых чисел, вещественных, строк и т.п. В языке Object Pascal такой возможности нет, но эта проблема частично решается с помощью механизма наследования.

Метаклассы

Метакласс - это класс, экземпляры которого являются классом. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках. Соответственно, они есть в Smalltalk и CLOS, но не в C++ и Object Pascal.

Взаимосвязь классов и объектов

Как уже говорилось ранее, классы и объекты - это отдельные, но тесно связанные между собой понятия. В частности, каждый объект является экземпляром какого-либо класса; класс может порождать любое число объектов. В большинстве практических случаев классы статичны, то есть все их особенности и содержание определены в процессе компиляции программы. Из этого следует, что любой созданный объект относится к строго фиксированному классу. Сами объекты, напротив, в процессе выполнения программы создаются и уничтожаются.

В качестве примера рассмотрим классы и объекты задачи по графическому редактору. Наиболее важные абстракции в этой сфере -- графические примитивы, а также контейнер, который управляет их отрисовкой. Трактовка этих классов объектов по самому их определению достаточно статична. Объекты же этих классов, напротив, динамичны. Набор графических примитивов меняется не очень часто. Существенно быстрее изменяется множество объектов на окне рисования, их цвет и взаимное расположение друг относительно друга.

На этапе анализа и ранних стадиях проектирования решаются две основные задачи:

- Выявление классов и объектов, составляющих словарь предметной области.
- Построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

В первом случае говорят о ключевых абстракциях задачи (совокупность классов и объектов), во втором - о механизмах реализации (совокупность структур).

Измерение качества абстракции

Как показывает практика, для построения системы должен использоваться минимальный набор неизменяемых компонент; сами компоненты должны быть по возможности стандартизованы и рассматриваться в рамках единой модели. Применительно к объектно-ориентированному проектированию такими компонентами являются классы и объекты, отражающие ключевые абстракции системы, а единство обеспечивается соответствующими механизмами реализации.

Опыт показывает, что процесс выделения классов и объектов является последовательным и итеративным. За исключением самых простых задач с первого раза не удается окончательно выделить и описать классы. Очень важно с самого начала по возможности приблизиться к правильным решениям, чтобы сократить число последующих шагов приближения к истине. Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев:

- зацепление
- связность
- достаточность
- полнота
- примитивность.

Зацепление – это степень глубины связей между отдельными модулями. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать.

Связность - это степень взаимодействия между элементами отдельного модуля (а для OOD еще и отдельного класса или объекта), характеристика его насыщенности.

Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели.

К идеям зацепления и связности тесно примыкают понятия **достаточности, полноты и примитивности**.

Под **достаточностью** подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Иначе говоря, компоненты должны быть полностью пригодны к использованию. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию.

Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все для взаимодействия с пользователями.

Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, вынося наверх такие операции, которые можно реализовать на более низком уровне.

Из этого вытекает требование примитивности.

Примитивными являются только такие операции, которые требуют доступа к внутренней реализации абстракции.

Таким образом, операция, которая требует прямого доступа к структуре данных, примитивна по определению. Операция, которая может быть описана в терминах существующих примитивных операций, но ценой значительно больших вычислительных затрат, также является кандидатом на включение в разряд примитивных.

4. ОСНОВЫ ЯЗЫКА UML

Объектно-ориентированные языки моделирования появились в период с середины 70-х до конца 80-х годов, когда исследователи, поставленные перед необходимостью учитывать новые возможности объектно-ориентированных языков программирования и требования, предъявляемые все более сложными приложениями, вынуждены были начать разработку различных альтернативных подходов к анализу и проектированию. С 1989 по 1994 год число различных объектно-ориентированных методов возросло с десяти более чем до пятидесяти. Тем не менее многие пользователи испытывали затруднения при выборе языка моделирования, который бы полностью соответствовал их потребностям, что послужило причиной так называемой "войны методов". В результате этих войн появилось новое поколение методов, среди которых особое значение приобрели языки **Booch**, созданный Грейди Бучем (Grady Booch), **OOSE** (Object-Oriented Software Engineering), разработанный Айваром Джекобсоном (Ivar Jacobson) и **OMT** (Object Modeling Technique), автором которого является Джеймс Рамбо (James Rumbaugh). Каждый из этих методов можно считать вполне целостным и законченным, хотя любой из них имеет не только сильные, но и слабые стороны. Выразительные возможности метода Буча особенно важны на этапах проектирования и конструирования модели. OOSE великолепно приспособлен для анализа и формулирования требований, а также для высокоуровневого проектирования. OMT оказался особенно полезным для анализа и разработки информационных систем, ориентированных на обработку больших объемов данных.

Критическая масса новых идей формируется к середине 90-х годов, когда Грейди Буч (компания Rational Software Corporation), Айвар Джекобсон (Objectory) и Джеймс Рамбо (General Electric) предприняли попытку объединить свои методы, уже получившие мировое признание как наиболее перспективные в данной области.

Начав унификацию, авторы поставили перед собой **три главные цели**:

1. моделировать системы целиком, от концепции до исполняемого артефакта, с помощью объектно-ориентированных методов;
2. решить проблему масштабируемости, которая присуща сложным системам, предназначенным для выполнения ответственных задач;
3. создать такой язык моделирования, который может использоваться не только людьми, но и компьютерами.

Официально создание UML началось в октябре 1994 года, когда Рамбо перешел в компанию Rational Software, где работал Буч. Первоначальной целью было объединение методов Буча и OMT. Первая пробная версия 0.8 Унифицированного Метода (Unified Method), как его тогда называли, появилась в октябре 1995 года. Приблизительно в это же время в компанию Rational перешел Джекобсон, и проект UML был расширен с целью включить в него язык OOSE. В результате совместных усилий в июне 1996 года вышла версия 0.9 языка UML. На протяжении всего года создатели занимались сбором отзывов от основных компаний, работающих в области конструирования программного обеспечения. За это время стало ясно" что большинство таких компаний сочло UML языком, имеющим стратегическое значение для их бизнеса. В результате был основан консорциум UML, в который вошли организации, изъявившие желание предоставить ресурсы для работы, направленной на создание полного определения UML.

Версия 1.0 языка появилась в результате совместных усилий компаний Digital Equipment Corporation, Hewlett Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments и Unisys. UML 1.0 оказался хорошо определенным, выразительным, мощным языком, применимым для решения большого количества разнообразных задач. В январе 1997 года он был представлен Группе по управлению объектами (Object Management Group, OMG) на конкурс по созданию стандартного языка моделирования.

Между январем и июнем 1997 года консорциум UML расширился, в него вошли практически все компании, откликнувшиеся на призыв OMG, а именно: Andersen Consulting, Ericsson, ObjecTime Limited, Platinum Technology, Ptech, Reich Technologies, Softeam, Sterling Software и Taskon. Чтобы формализовать спецификации UML и координировать работу с другими группами, занимающимися стандартизацией, под руководством Криса Кобрина (Cris Kobryn) из компании MCI Systemhouse и Эда Эйкхолта (Ed Eykholt) из Rational была организована семантическая группа. Пересмотренная версия UML (1.1) была снова представлена на рассмотрение OMG в июле 1997 года. В сентябре версия была утверждена на заседаниях Группы по анализу и проектированию и Комитета по архитектуре OMG, а 14 ноября 1997 года принята в качестве стандарта на общем собрании всех членов OMG.

Дальнейшая работа по развитию UML проводилась Группой по усовершенствованию (Revision Task Force, RTF) OMG под руководством Криса Кобрина. В июне 1998 года вышла версия UML 1.2, а осенью 1998 - UML 1.3.

Принципы моделирования

Моделирование имеет богатую историю во всех инженерных дисциплинах. Длительный опыт его использования позволил сформулировать четыре основных принципа.

Во-первых, ***выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение.*** Правильно выбранная модель высветит самые большие проблемы разработки и позволит проникнуть в самую суть задачи, что при ином подходе было бы попросту невозможно.

Второй принцип формулируется так: ***каждая модель может быть воплощена с разной степенью абстракции.*** Лучшей моделью будет та, которая позволяет выбрать уровень детализации в зависимости от того, кто и с какой целью на нее смотрит. Для аналитика или конечного пользователя наибольший интерес представляет вопрос «что», а для разработчика - вопрос «как». В обоих случаях необходима возможность рассматривать систему на разных уровнях детализации в разное время.

Третий принцип: ***лучшие модели - те, что ближе к реальности.*** Логично полагать, что лучшей моделью будет та, которая точнее соотносится с реальностью. Однако поскольку модель всегда упрощает реальность, основная задача состоит в том, чтобы это упрощение не повлекло за собой какие-либо существенные потери.

Четвертый принцип заключается в том, что ***нельзя ограничиваться созданием только одной модели.*** Наилучший подход при разработке любой нетривиальной системы - использовать совокупность нескольких моделей, почти независимых друг от друга. Для понимания архитектуры сложной системы требуется несколько взаимодополняющих видов: вид с точки зрения прецедентов, или вариантов использования (чтобы выявить требования к системе), с точки зрения проектирования (чтобы построить словарь предметной области и области решения) и т.д. Каждый из перечисленных видов имеет множество структурных и поведенческих аспектов, которые в своей совокупности составляют детальный чертеж программной системы.

Введение в язык UML

Унифицированный язык моделирования (UML) является стандартным инструментом для создания "чертежей" программного обеспечения. С помощью UML можно визуализировать, специфицировать, конструировать и документировать артефакты программных систем.

UML пригоден для моделирования любых систем: от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени. Это достаточно выразительный язык, позволяющий рассмотреть систему со всех точек зрения, имею-

щих отношение к ее разработке и последующему развертыванию. Несмотря на обилие выразительных возможностей, этот язык прост для понимания и использования.

Несмотря на свои достоинства, UML - это всего лишь язык; он является одной из составляющих процесса разработки программного обеспечения, и не более того. Хотя UML не зависит от моделируемой реальности, лучше всего его применять, когда процесс моделирования, основанный на рассмотрении прецедентов использования, является итеративным и пошаговым, а сама система имеет четко выраженную архитектуру.

Концептуальная модель UML

Для понимания UML необходимо рассмотреть его концептуальную модель, которая включает в себя три составные части; основные строительные блоки языка, правила их сочетания и некоторые общие для всего языка механизмы.

Строительные блоки UML

Словарь языка UML включает три вида строительных блоков:

- сущности;
- отношения;
- диаграммы.

Сущности - это абстракции, являющиеся основными элементами модели. Отношения связывают различные сущности; диаграммы группируют представляющие интерес совокупности сущностей.

В UML имеется четыре типа сущностей:

- структурные;
- поведенческие;
- группирующие;
- аннотационные.

Сущности являются основными объектно-ориентированными блоками языка. С их помощью можно создавать корректные модели.

Структурные сущности

Структурные сущности - это имена существительные в моделях на языке UML. Как правило, они представляют собой статические части модели, соответствующие концептуальным или физическим элементам системы. Существует семь основных разновидностей структурных сущностей: класс, интерфейс, кооперация, прецедент, активный класс, компонент и узел.

Класс (Class) - это описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Класс реализует один или несколько интерфейсов.

Графически класс изображается в виде прямоугольника, в котором обычно записаны его имя, атрибуты и операции, как показано на рис. 1.

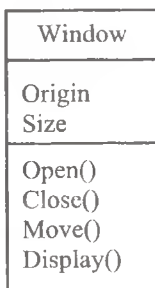

Рис. 1 Классы

Интерфейс (Interface) - это совокупность операций, которые определяют сервис (набор услуг), предоставляемый классом или компонентом. Таким образом, интерфейс описывает видимое извне поведение элемента. Интерфейс может представлять поведение класса или компонента полностью или частично; он определяет только спецификации операций (сигнатуры), опуская реализацию. Графически интерфейс изображается в виде круга, под которым пишется его имя. Интерфейс редко существует сам по себе - обычно он присоединяется к реализующему его классу или компоненту.

Кооперация (Collaboration) определяет взаимодействие; она представляет собой совокупность ролей и других элементов, которые, работая совместно, производят некоторый кооперативный эффект, не сводящийся к простой сумме слагаемых.

Графически кооперация изображается в виде эллипса, ограниченного пунктирной линией, в который обычно заключено только имя, как показано на рис. 2.

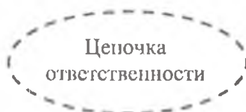

Рис. 2 Кооперация

Прецедент (Use case) - это описание последовательности выполняемых системой действий, которая производит наблюдаемый результат значимый для какого-то определенного актера (Actor).

Прецедент применяется для структурирования поведенческих сущностей модели. Прецеденты реализуются посредством кооперации. Графически прецедент изображается в виде ограниченного непрерывной линией эллипса, обычно содержащего только его имя, как показано на рис. 3.

Рис. 3 Прецедент

Три другие сущности - **активные классы, компоненты и узлы** - подобны классам: они описывают совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Тем не менее, они в достаточной степени отличаются друг от друга и от классов и, учитывая их важность при моделировании определенных аспектов объектно-ориентированных систем, заслуживают специального рассмотрения.

Активным классом (Active class) называется класс, объекты которого вовлечены в один или несколько процессов, или нитей (Threads), и поэтому могут инициировать управляющее воздействие. Активный класс во всем подобен обычному классу, за исключением того, что его объекты представляют собой элементы, деятельность которых осуществляется одновременно с деятельностью других элементов. Графически активный класс изображается так же, как простой класс, но ограничивающий прямоугольник рисуется жирной линией и обычно включает имя, атрибуты и операции.

Компонент (Component) - это физическая заменяемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает его реализацию. В системе можно встретить различные виды устанавливаемых компонентов, такие как COM+ или Java Beans, а также компоненты, являющиеся артефактами процесса разработки, например файлы исходного кода. Компонент, как правило, представляет собой физическую упаковку логических элементов, таких как классы, интерфейсы и кооперации. Гра-

фически компонент изображается в виде прямоугольника с вкладками, содержащего обычно только имя, как показано на рис. 4.

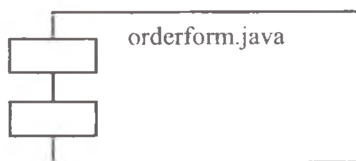

Рис. 4 Компонент

Узел (Node) - это элемент реальной (физической) системы, который существует во время функционирования программного комплекса и представляет собой вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а часто еще и способностью обработки. Совокупность компонентов может размещаться в узле, а также мигрировать с одного узла на другой. Графически узел изображается в виде куба, обычно содержащего только имя, как показано на рис. 5.

Рис. 5 Узлы

Поведенческие сущности

Поведенческие сущности (Behavioral things) являются динамическими составляющими модели UML. Это глаголы языка: они описывают поведение модели во времени и пространстве. Существует всего два основных типа поведенческих сущностей: взаимодействие и автомат.

Взаимодействие (Interaction) - это поведение, суть которого заключается в обмене сообщениями между объектами в рамках конкретного контекста для достижения определенной цели. С помощью взаимодействия можно описать как отдельную операцию, так и поведение совокупности объектов. Взаимодействие предполагает ряд других элементов, таких как сообщения, последовательности действий (поведение, инициированное сообщением) и связи (между объектами). Графически сообщения изобража-

ются в виде стрелки, над которой почти всегда пишется имя соответствующей операции, как показано на рис. 6.

Рис. 6 Сообщения

Автомат (State machine) - это алгоритм поведения, определяющий последовательность состояний, через которые объект или взаимодействие проходят на протяжении своего жизненного цикла в ответ на различные события, а также реакции на эти события. С помощью автомата можно описать поведений отдельного класса или кооперации классов. С автоматом связан ряд других элементов: состояния, переходы (из одного состояния в другое), события (сущности инициирующие переходы) и виды действий (реакция на переход). Графически состояние изображается в виде прямоугольника с закругленными углами (см. рис. 7).

Рис. 7 Автомат

Группирующие сущности

Группирующие сущности являются организующими частями модели UML. Это блоки, на которые можно разложить модель. Есть только одна первичная группирующая сущность, а именно пакет.

Пакеты (Packages) представляют собой универсальный механизм организации элементов в группы. В пакет можно поместить структурные, поведенческие и даже другие группирующие сущности. В отличие от компонентов, существующих во время работы программы, пакеты носят чисто концептуальный характер, то есть существуют только во время разработки. Изображается пакет в виде папки с закладкой, содержащей, как правило, только имя и иногда - содержимое (см. рис. 8).

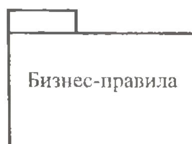

Рис. 8 Пакеты

Аннотационные сущности

Аннотационные сущности - пояснительные части модели UML. Имеется только один базовый тип аннотационных элементов - примечание (Note).

Примечание - это просто символ для изображения комментариев или ограничений, присоединенных к элементу или группе элементов. Графически примечание изображается в виде прямоугольника с загнутым краем, содержащим текстовый или графический комментарий, как показано на рис. 9.

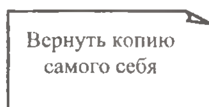

Рис. 9 Примечания

Чаще всего примечания используются, чтобы снабдить диаграммы комментариями или ограничениями, которые можно выразить в виде неформального или формального текста.

В языке UML определены четыре типа отношений:

- зависимость;
- ассоциация;
- обобщение;
- реализация.

Эти отношения являются основными связующими строительными блоками в UML и применяются для создания корректных моделей.

Зависимость (Dependency) - это семантическое отношение между двумя сущностями, при котором изменение одной из них, независимой, может повлиять на семантику другой, зависимой. Графически зависимость изображается в виде прямой пунктирной линии, часто со стрелкой, которая может содержать метку (см. рис. 10).

Рис. 10 Зависимости

Ассоциация (Association) - структурное отношение, описывающее совокупность связей; связь - это соединение между объектами.

Разновидностью ассоциации является **агрегирование** (Aggregation) - так называют структурное отношение между целым и его частями. Графически ассоциация изображается в виде прямой линии (иногда завершающейся стрелкой или содержащей метку), рядом с которой может присутствовать дополнительные обозначения, например кратность и имена ролей.

Обобщение (Generalization) - это отношение "специализация/обобщение", при котором объект специализированного элемента (потомок) может быть подставлен вместо объекта обобщенного элемента (родителя или предка)

Таким образом, потомок (Child) наследует структуру и поведение своего родителя (Parent). Графически отношение обобщения изображается в виде линии с незакрашенной стрелкой, указывающей на родителя.

Реализация (Realization) - это семантическое отношение между классификаторами, при котором один классификатор определяет “контракта”, а другой гарантирует его выполнение. Отношения реализации встречаются в двух случаях: во-первых, между интерфейсами и реализующими их классами или компонентами, а во-вторых, между прецедентами и реализующими их кооперациями. Отношение реализации изображается в виде пунктирной линии с не закрашенной стрелкой, как нечто среднее между отношениями обобщения и зависимости.

Диаграмма в UML - это графическое представление набора элементов, изображаемое чаще всего в виде связанного графа с вершинами (сущностями) и ребрами (отношениями). Диаграммы рисуют для визуализации системы с разных точек зрения. Диаграмма - в некотором смысле одна из проекций системы. Как правило, за исключением наиболее тривиальных случаев, диаграммы дают свернутое представление элементов, из которых составлена система. Один и тот элемент может присутствовать во всех диаграммах, или только в нескольких (самый распространенный вариант), или не присутствовать ни в одной (очень редкий). Теоретически диаграммы могут содержать любые комбинации сущностей и отношений. На практике, однако, применяется сравнительно небольшое количество типовых комбинаций, соответствующих наиболее употребительным видам, которые составляют архитектуру программной системы.

Основы структурного моделирования

Классы

Классы - это самые важные строительные блоки любой объектно-ориентированной системы. Они представляют собой описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Класс реализует один или несколько интерфейсов.

Классы используются для составления словаря разрабатываемой системы. Это могут быть абстракции, являющиеся частью предметной области, либо классы, на которые опирается реализация. С их помощью описывают программные, аппаратные или чисто концептуальные сущности.

Моделирование системы предполагает идентификацию сущностей, важных с той или иной точки зрения. Эти сущности составляют словарь моделируемой системы. Каждая из сущностей отличается от других и характеризуется собственным набором свойств.

У каждого класса должно быть имя, отличающее его от других классов. Имя класса - это текстовая строка. Взятое само по себе, оно называется простым именем; к составному имени спереди добавлено имя пакета, куда входит класс. Имя класса может состоять из любого числа букв, цифр и ряда знаков препинания (за исключением таких, например, как двоеточие, которое применяется для отделения имени класса от имени объемлющего пакета). Имя может занимать несколько строк. На практике для именования класса используют одно или несколько коротких существительных, взятых из словаря моделируемой системы. Обычно каждое слово в имени класса пишется с заглавной буквы, например: Customer (Клиент), Wall (Стена), TemperatureSensor (ДатчикТемпературы).

Атрибут - это именованное свойство класса, включающее описание множества значений, которые могут принимать экземпляры этого свойства.

Класс может иметь любое число атрибутов или не иметь их вовсе. Атрибут представляет некоторое свойство моделируемой сущности, общее для всех объектов данного класса. Таким образом, атрибут является абстракцией данных объекта или его состояния. В каждый момент времени любой атрибут объекта, принадлежащего ному классу, обладает вполне определенным значением. Атрибуты представлены в разделе, который расположен под именем класса; при этом, как правило, указываются только их имена.

Имя атрибута, как и имя класса, может быть произвольной текстовой строкой. На практике для именования атрибута используют одно или несколько коротких существительных, соответствующих некоторому свойству объемлющего класса. Каждое слово в имени атрибута, кроме самого первого, обычно пишется с заглавной буквы.

При описании атрибута можно явным образом указывать его класс и начальное значение, принимаемое по умолчанию.

Операцией называется реализация услуги, которую можно запросить у любого объекта класса для воздействия на поведение.

Иными словами, операция - это абстракция того, что позволено делать с объектом. У всех объектов класса имеется общий набор операций. Класс может содержать любое число операций или не содержать их вовсе. Операции класса изображаются в разделе, расположенном ниже раздела с атрибутами. При этом можно ограничиться только именами. Имя операции,

как и имя класса, может быть произвольной текстовой строкой. На практике для именования операций используют короткий глагол или глагольный оборот, соответствующий определенному поведению объемлющего класса. Каждое слово в имени операции, кроме самого первого, обычно пишут с заглавной буквы, например move или isEmpty.

При изображении класса необязательно сразу показывать все его атрибуты и операции. Как правило, это попросту невозможно - их чересчур много для одного рисунка, - да и не требуется (поскольку для данного представления системы лишь небольшое подмножество атрибутов и, операций имеет значение). По этим причинам класс обычно сворачивают, то есть изображают лишь некоторые из имеющихся атрибутов и операций, а то и вовсе опускают их. Таким образом, пустой раздел в соответствующем месте прямоугольника может означать не отсутствие атрибутов или операций, а только то, что их не сочли нужным изобразить. Явным образом наличие дополнительных атрибутов или операций можно обозначить, поставив в конце списка многоточие.

Обязанности (Responsibilities) класса - это своего рода контракт, которому он должен подчиняться.

Определяя класс, вы постулируете, что все его объекты имеют однотипное состояние и ведут себя одинаково. Выражаясь абстрактно, соответствующие атрибуты и операции как раз и являются теми свойствами, посредством которых выполняются обязанности класса. Моделирование классов лучше всего начинать с определения обязанности сущностей, которые входят в словарь системы. В-принципе, число обязанностей класса может быть произвольным, но на практике хорошо структурированный класс имеет, по меньшей мере, одну обязанность; с другой стороны, их не должно быть и слишком много. При уточнении модели обязанности класса преобразуются в совокупность атрибутов и операций, которые должны наилучшим образом обеспечить их выполнение.

Распределение обязанностей в системе

Если в ваш проект входит нечто большее, нежели пара несложных классов, предстоит позаботиться о сбалансированном распределении обязанностей. Это значит, что надо избегать слишком больших или, наоборот, чересчур маленьких классов. Каждый класс должен хорошо делать что-то одно. Если ваши абстрактные классы слишком велики, то модель будет трудно модифицировать и повторно использовать. Если же они слишком малы, то придется иметь дело с таким большим количеством абстракций, что ни понять, ни управлять ими будет невозможно. Язык UML способен помочь в визуализации и специфицировании баланса обязанностей.

Моделирование распределения обязанностей в системе включает в себя следующие этапы:

1. Идентифицируйте совокупность классов, совместно отвечающих за некоторое поведение.
2. Определите обязанности каждого класса.
3. Взгляните на полученные классы как на единое целое и разбейте те из них, у которых слишком много обязанностей, на меньшие - и наоборот, крошечные классы с элементарными обязанностями объедините в более крупные.
4. Перераспределите обязанности так, чтобы каждая абстракция стала в разумной степени автономной.
5. Рассмотрите, как классы кооперируются друг с другом и перераспределите обязанности с таким расчетом, чтобы ни один класс в рамках кооперации не делал слишком много или слишком мало.

Ассоциации

Ассоциацией (Association) называется структурное отношение, показывающее, что объекты одного типа неким образом связаны с объектами другого типа.

Если между двумя классами определена ассоциация, то можно перемещаться от объектов одного класса к объектам другого. Вполне допустимы случаи, когда оба конца ассоциации относятся к одному и тому же классу. Это означает, что с объектом некоторого класса позволительно связать другие объекты из того же класса. Ассоциация, связывающая два класса, называется **бинарной**. Графически ассоциация изображается в виде линии, соединяющей класс сам с собой или с другими классами. Ассоциацию следует использовать для того, чтобы показать структурные отношения. Ассоциации и зависимости, в отличие от отношений обобщения, могут быть рефлексивными.

Помимо описанной базовой формы существует четыре дополнения, применимых к ассоциациям.

Ассоциации может быть присвоено имя, описывающее природу отношения. Чтобы избежать возможных двусмысленностей в понимании имени, достаточно с помощью черного треугольника указать направление, в котором оно должно читаться, как показано на рис. 11.

Рис. 11 Имена ассоциаций

Класс, участвующий в ассоциации, играет в ней некоторую роль. По существу, это "лицо", которым класс, находящийся на одной стороне ассоциации, обращен к классу с другой ее стороны. Можно явно обозначить роль, которую класс играет в ассоциации. На рис. 12 показано, что класс Человек, играющий роль работника, ассоциирован с классом Компания, играющим роль работодателя. Роли тесно связаны с семантикой интерфейсов.

Рис. 12 Роли

Один класс может играть в разных ассоциациях как одну и ту же роль, так и различные.

Ассоциации отражают структурные отношения между объектами. Часто при моделировании бывает важно указать, сколько объектов может быть связано посредством одного экземпляра ассоциации (то есть одной связи). Это число называется **кратностью роли ассоциации** и записывается либо как выражение, значением которого является диапазон значений, либо в явном виде. Указывая кратность на одном конце ассоциации, вы тем самым говорите, что на этом конце именно столько объектов должно соответствовать каждому объекту на противоположном конце. Кратность можно задать равной единице (1), можно указать диапазон: "ноль или единица" (0..1), "много" (0..*), "единица или больше" (1..*). Разрешается также указывать определенное число (например, 3). С помощью списка можно задать и более сложные кратности, например 0..1, 3..4, 6..*, что означает любое число объектов, кроме 2 и 5.

Агрегирование

Простая ассоциация между двумя классами отражает структурное отношение между равноправными сущностями, когда оба класса находятся на одном концептуальном уровне и ни один не является более важным, чем другой. Но иногда приходится моделировать отношение типа "часть/целое", в котором один из классов имеет более высокий ранг (целое) и состоит из нескольких меньших по рангу частей. Отношение такого типа называют **агрегированием**. Агрегирование является частным случаем ассоциации и изображается в виде простой ассоциации с незакрашенным ромбом:

со стороны "целого", как показано на рис. 13.

Рис. 13 Агрегирование

Существует много разновидностей данного отношения. Не закрашенный ромб отличает целое от "части" - и только. Эта простая форма агрегирования является чисто концептуальной; она не влияет на результат навигации по ассоциации между целым и его частями и не подразумевает наличия между ними какой-либо зависимости по времени жизни.

Типичные приемы моделирования

Моделирование отношений наследования осуществляется в таком порядке:

1. Найдите атрибуты, операции и обязанности, общие для двух или более классов из данной совокупности.
2. Вынесите эти элементы в некоторый общий класс (если надо, создайте новый, но следите, чтобы уровней не оказалось слишком много).
3. Отметьте в модели, что более специализированные классы наследуют в общем, включив отношение обобщения, направленное от каждого потомка к его родителю.

На рис. 14 изображено несколько классов, взятых из приложения по организации работы трейдеров. Здесь показано отношение обобщения, которое от четырех классов – РасчетныйСчет, Акция, Облигация и Собственность – направлена к более общему классу ЦенныеБумаги. Он является родителем, а остальные его потомками. Каждый специализированный класс – это частный случай класса ЦенныеБумаги. Обратите внимание, что в классе ЦенныеБумаги есть две операции - presentValue (текущаяСтоимость) и history (история). Это значит, что все его потомки наследуют данные операции, а заодно и все остальные атрибуты и операции родителя, которые могут не изображаться на рисунке.

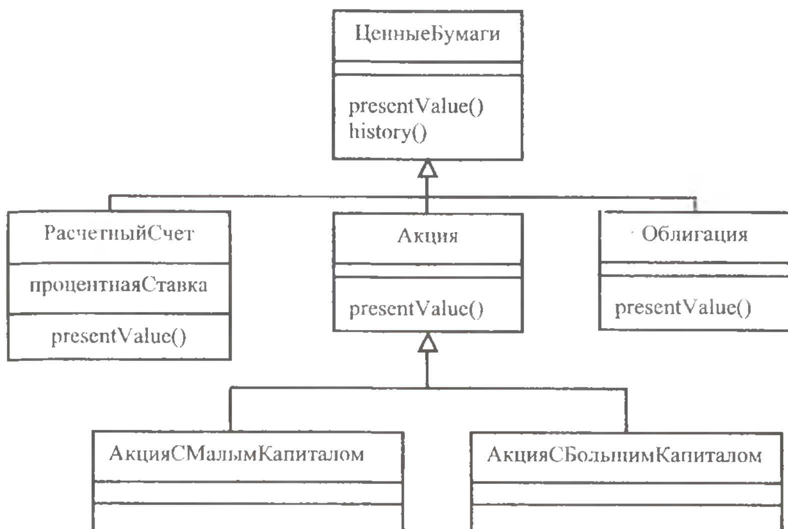

Рис. 14 Отношения наследования

Иерархия "обобщение/специализация" не обязательно ограничивается двумя уровнями. Как видно из рисунка, вполне допустимо существование более двух уровней наследования. АкцияСМалымКапиталом и АкцияСБольшимКапиталом - потомки класса Акция, который, в свою очередь, является потомком класса ЦенныеБумаги. Последний является базовым классом, поскольку не имеет родителей. Классы же АкцияСМалымКапиталом и АкцияСБольшимКапиталом - листовые, поскольку не имеют потомков. Наконец, класс Акция имеет как родителей, так и потомков, а следовательно, не является ни листовым, ни базовым.

Моделирование структурных отношений производится следующим образом:

1. Определите ассоциацию для каждой пары классов, между объектами которых надо будет осуществлять навигацию. Это взгляд на ассоциации с точки зрения данных.
2. Если объекты одного класса должны будут взаимодействовать с объектами другого иначе, чем в качестве параметров операции, следует определить между этими классами ассоциацию. Это взгляд на ассоциации с точки зрений поведения.

3. Для каждой из определенных ассоциаций задайте кратность (особенно если она не равна *, то есть значению по умолчанию) и имена ролей (особенно если это помогает объяснить модель).
4. Если один из классов ассоциации структурно или организационно представляет собой целое в отношении классов на другом конце ассоциации, выглядящих как его части, пометьте такую ассоциацию как агрегирование.

Механизмы дополнения и расширения

Самой важной разновидностью **дополнений** являются **примечания**, которые представляют собой графические символы для изображения ограничений или комментариев, присоединяемых к одному элементу или их совокупности. Примечания используются для включения в модель дополнительной информации, например требований, наблюдений, обзоров и объяснений.

Механизмы **расширения** UML позволяют расширять его возможности контролируемым образом. В их число входят **стереотипы**, **помеченные значения** и **ограничения**. Стереотипы расширяют словарь UML, позволяя создавать новые виды строительных блоков, производные от существующих, но при этом более полно соответствующие задаче. Помеченные значения расширяют свойства строительных блоков UML, разрешая вносить новую информацию в спецификацию элемента. Ограничения расширяют семантику строительных блоков UML, с их помощью можно вводить новые или изменять существующие правила. Перечисленные механизмы используют, чтобы настроить язык на конкретные требования предметной области и применяемой разработчиком методики проектирования.

Моделирование есть не что иное, как **способ обмена информацией**. UML представляет все необходимые инструменты для визуализации, специфицирования, конструирования и документирования артефактов разнообразных программных систем, однако при некоторых обстоятельствах его возможности требуется расширить или модифицировать. Однако всегда следует помнить, что он предназначен для обмена информацией между разработчиками и пользователями, и потому стоит придерживаться его основных правил и принципов, пока не возникает острая необходимость отклониться от них. Но даже в этом случае неформальную информацию следует представлять контролируемым образом, иначе никто не сможет понять, о чем идет речь.

Примечания позволяют давать произвольные комментарии и ограничения для пояснения созданной модели. Они могут указывать на артефакты, играющие важную роль в жизненном цикле разработки программного

обеспечения - такие, как требования - или же содержать наблюдения, обзоры или пояснения в свободной форме.

Стереотипы

Стереотип - это не то же самое, что родительский класс в отношении обобщения "родитель/потомок". Точнее было бы охарактеризовать его как некоторый метатип, поскольку каждый стереотип создает эквивалент нового класса в мета-модели UML. Например, при моделировании бизнес-процесса может потребоваться ввести в модель элементы, представляющие работников, документы и стратегии. Здесь и возникает необходимость в стереотипах.

В самом простом случае стереотип изображается в виде имени в кавычках (например, "name"), расположенного над именем другого элемента. Для наглядности стереотипу можно назначить пиктограмму, разместив ее справа от имени или применяя как базовый символ для стереотипной сущности.

Создавая пиктограмму для стереотипа, обычно используют цвет в качестве удобного визуального идентификатора (хотя злоупотреблять этой возможностью не следует). UML позволяет оформлять пиктограммы любым способом и, если позволяет реализация, они могут войти в набор примитивных инструментов, пополнив таким образом палитру средств, при помощи которых пользователи работают с данной предметной областью.

Помеченные значения

У каждой сущности в UML есть фиксированный набор свойств: классы имеют имена, атрибуты и операции; ассоциации - имена и концевые точки, (каждая со своими свойствами) и т.д. **Стереотипы позволяют добавлять новые сущности** в UML, а **помеченные значения - новые свойства**.

Метки можно определять для существующих элементов UML; можно также определить метки, применимые к отдельным стереотипам.

Помеченное значение - не то же самое, что атрибут класса. Скорее, оно может быть представлено как метаданные, поскольку его значение применяется к самому элементу, а не к его экземплярам. В простейшем варианте помеченное значение изображается в виде строки в фигурных скобках, расположенной под именем другого элемента.

Чаще всего с помощью помеченных значений описывают свойства, относящиеся к генерации кода или к управлению конфигурацией. Например, ими можно воспользоваться для указания языка программирования, на котором будет кодироваться конкретный класс. Можно также приме-

нить помеченные значения для указания имени автора и номера версии компонента.

Ограничения

Ограничение определяет условия, которые должны выполняться, чтобы модель была хорошо оформлена. Например, на рис. 15 показано, как отразить тот факт, что информация, передаваемая вдоль данной ассоциации, зашифрована. Аналогично можно специфицировать, что из совокупности ассоциаций только одна проявляется в данный момент времени. При моделировании систем реального времени часто используются временные и пространственные ограничения.

Рис. 15 Ограничения

Ограничение изображается в виде строки в фигурных скобках, расположенной рядом с соответствующим элементом. Эта нотация также используется как дополнение к базовой нотации элемента для визуального представления тех частей его спецификации, которые не имеют графического образа.

Типичные приемы моделирования

Комментарии

Как правило, примечания используются для того, чтобы в свободной форме записывать наблюдения, обзоры и пояснения. Включая такие комментарии в модель, вы можете превратить ее в хранилище всевозможных артефактов, создаваемых в процессе разработки. Кроме того, посредством

комментариев можно визуализировать требования к системе и явно показать, как они связаны с теми или иными частями модели.

Моделирование комментария производится следующим образом:

1. Включите его в виде текста в примечание и расположите рядом с элементом, к которому он относится. Можно показать отношение более явно, соединив примечание с соответствующими элементами при помощи зависимости.
2. Помните, что элементы модели можно скрыть или сделать видимыми. Это означает, что не обязательно делать комментарий видимым всегда, когда видны элементы, к которым он присоединен. Лучше раскрывать комментарии в диаграммах только тогда, когда это необходимо для передачи какой-либо информации.
3. Если комментарий длинный или содержит нечто более сложное, чем обычный текст, подумайте о том, чтобы вынести его во внешний документ, на который можно поставить ссылку, или встроить его в примечание, присоединенное к модели.
4. По мере продвижения процесса моделирования убирайте все комментарии, кроме тех, что содержат важные решения, которые не могут быть выведены из самой модели, либо тех, что представляют исторический интерес.

Моделирование новых строительных блоков производится в следующем порядке:

1. Убедитесь, что с помощью базового набора средств UML невозможно воплотить ваш замысел. Если вы столкнулись с типичной задачей моделирования, то, скорее всего, уже существует стандартный стереотип для ее решения.
2. Если вы убеждены, что никакого другого способа выразить соответствующую семантику не существует, подыщите примитив UML, более всего похожий на то, что вы собираетесь моделировать (например, класс, интерфейс, компонент, узел, ассоциацию и т.д.) и определите для него новый стереотип. Помните, что вы можете определить иерархию стереотипов, в которой есть как общие, так и специализированные элементы; однако не увлекайтесь этим чрезмерно.
3. Задайте общие свойства и семантику, выходящие за пределы базового элемента, определив множество помеченных значений и ограничений для стереотипа.
4. Если вы хотите связать со стереотипными элементами специальный визуальный образ, определите для каждого из них новую пиктограмму.

Диаграммы

В процессе моделирования человек упрощает реальность, чтобы лучше понять проектируемую систему. Используя UML, вы можете строить модели из базовых блоков, таких как классы, интерфейсы, кооперации, компоненты, узлы, зависимости, обобщения и ассоциации. Диаграммы позволяют обозревать эти строительные блоки в удобной для понимания форме.

Диаграмма - это графическое представление совокупности элементов, чаще всего изображаемое в виде связного графа, состоящего из вершин (сущностей) и ребер (отношений). С помощью диаграмм можно визуализировать систему с различных точек зрения. Поскольку сложное целое нельзя понять, глядя на него лишь с одной стороны, в UML определено много разных диаграмм, которые позволяют сосредоточиться на различных аспектах моделируемой системы.

Хорошие диаграммы облегчают понимание модели. Продуманный выбор диаграмм при моделировании системы позволяет задавать правильные вопросы о ней, помогает грамотной постановке задачи и проясняет последствия принимаемых решений.

При рассмотрении статических частей системы используются диаграммы следующих типов:

- диаграммы классов;
- диаграммы объектов;
- диаграммы компонентов;
- диаграммы развертывания.

Для работы с динамическими частями системы применяются пять типов диаграмм, перечисленных ниже:

- диаграммы прецедентов;
- диаграммы последовательности;
- диаграммы кооперации;
- диаграммы состояний;
- диаграммы деятельности.

Диаграммы классов

Диаграммы классов при моделировании объектно-ориентированных систем встречаются чаще других. На таких диаграммах показывается множество классов, интерфейсов, коопераций и отношений между ними.

Диаграммы классов используются для моделирования статического вида системы с точки зрения проектирования. Сюда по большей части относится моделирование словаря системы, коопераций и схем. Кроме того, диаграммы классов составляют основу еще двух диаграмм - компонентов и развертывания.

Диаграммы классов важны не только для визуализации, специфицирования и документирования структурных моделей, но также для прямого и обратного проектирования исполняемых систем.

Диаграммой классов (Class diagram) называют диаграмму, на которой показано множество классов, интерфейсов, коопераций и отношений между ними. Ее изображают в виде множества вершин и дуг.

Типичные примеры применения

Обычно диаграммы классов используются в следующих целях:

- для моделирования словаря системы. Моделирование словаря системы предполагает принятие решения о том, какие абстракции являются частью системы, а какие - нет. С помощью диаграмм классов вы можете определить эти абстракции и их обязанности;
- для моделирования простых коопераций. Кооперация - это сообщество классов, интерфейсов и других элементов, работающих совместно для обеспечения некоторого кооперативного поведения, более значимого, чем сумма составляющих его элементов.
- для моделирования логической схемы базы данных. Логическую схему можно представлять себе как чертеж концептуального проекта базы данных. Во многих сферах деятельности требуется хранить устойчивую (persistent) информацию в реляционной или объектно-ориентированной базе данных. Моделировать схемы также можно с помощью диаграмм классов.

Прямое и обратное проектирование

Моделирование, конечно, важная вещь, но следует помнить, что основным результатом деятельности группы разработчиков являются не диаграммы, а программное обеспечение. Все усилия по созданию моделей направлены только на то, чтобы в оговоренные сроки написать программу, которая отвечает изменяющимся потребностям пользователей и бизнеса. Поэтому очень важно, чтобы ваши модели и основанные на них реализации соответствовали друг другу, причем с минимальными затратами по поддержанию синхронизации между ними.

Хотя UML не определяет конкретного способа отображения на какой-либо объектно-ориентированный язык, он проектировался с. учетом этого требования. В наибольшей степени это относится к диаграммам классов, содержание которых без труда отображается на такие известные объектно-ориентированные языки программирования, как Java, C++, Smalltalk, Eiffel, Ada, Object Pascal и Forte.

Выделяют два типа проектирования – прямое и обратное.

Прямым проектированием (Forward engineering) называется процесс преобразования модели в код путем отображения на некоторый язык реализации.

Процесс прямого проектирования приводит к потере информации, поскольку написанные: языке UML модели семантически богаче любого из существующих объектно-ориентированных языков. Именно поэтому кроме программного кода очень важно иметь модель разрабатываемой системы.

Прямое проектирование диаграммы классов производится следующим образом:

1. Идентифицируйте правила отображения модели на один или несколько языков программирования по вашему выбору. Это допустимо делать как при работе над одним конкретным проектом, так и для всей организации в целом
2. В зависимости от семантики выбранных языков, вероятно, придется отказаться от использования некоторых возможностей UML Например, язык UML допускает множественное наследование, а язык программирования Object Pascal - только одиночное.
3. Применяйте помеченные значения для специфицирования языка программирования. Это можно сделать как для индивидуальных классов, так и для пакетов или коопераций.
4. Пользуйтесь инструментальными средствами для прямого проектирования моделей.

Обратным проектированием (Reverse engineering) называется процесс преобразования в модель кода, записанного на каком-либо языке программирования.

В результате этого процесса получается огромный объем информации, часть которой находится на более низком уровне детализации, чем необходимо для построения полезных моделей. В то же время обратное проектирование никогда не бывает полным.

Обратное проектирование диаграммы классов осуществляется следующим образом:

1. Идентифицируйте правила для преобразования из выбранного вами языка реализации. Это можно сделать на уровне проекта или организации в целом.
2. С помощью инструментального средства укажите код, который вы хотите подвергнуть обратному проектированию. Воспользуйтесь этим средством для создания новой модели или для модификации ранее созданной.
3. Пользуясь инструментальными средствами, создайте диаграмму классов путем опроса полученной модели.

Создавая диаграмму классов на языке UML, помните, что это всего лишь графическое изображение статического вида системы с точки зрения проектирования. Взятая в отрыве от остальных, ни одна диаграмма классов не может и не должна охватывать этот вид целиком. Диаграммы классов описывают его исчерпывающе, но каждая в отдельности - лишь один из его аспектов.

Хорошо структурированная диаграмма классов обладает следующими свойствами:

- заостряет внимание только на одном аспекте статического вида системы с точки зрения проектирования;
- содержит лишь элементы, существенные для понимания данного аспекта;
- показывает детали, соответствующие требуемому уровню абстракции, опуская те, без которых можно обойтись; и не настолько лаконична, чтобы ввести читателя в заблуждение относительно важных аспектов семантики.

При изображении диаграммы классов необходимо руководствоваться следующими правилами:

- дайте диаграмме имя, связанное с ее назначением;
- располагайте элементы так, чтобы свести к минимуму число пересекающихся линий;
- пространственно организуйте элементы так, чтобы семантически близкие сущности располагались рядом;
- чтобы привлечь внимание к важным особенностям диаграммы, используйте примечания и цвет;

- старайтесь не показывать слишком много разных видов отношений; как правило, в каждой диаграмме классов должны доминировать отношения какого-либо одного вида.

Диаграммы объектов

Диаграммой объектов называют диаграмму, на которой показано множество объектов и отношений между ними в некоторый момент времени. Графически диаграмму объектов представляют в виде графа, состоящего из вершин и ребер.

Диаграммы объектов применяют при моделировании статических видов системы с точки зрения проектирования и процессов. При этом моделируется "**снимок**" системы в данный момент времени и изображается множество объектов, состояний и отношений между ними. Пример диаграммы объектов приводится на рисунке 16.

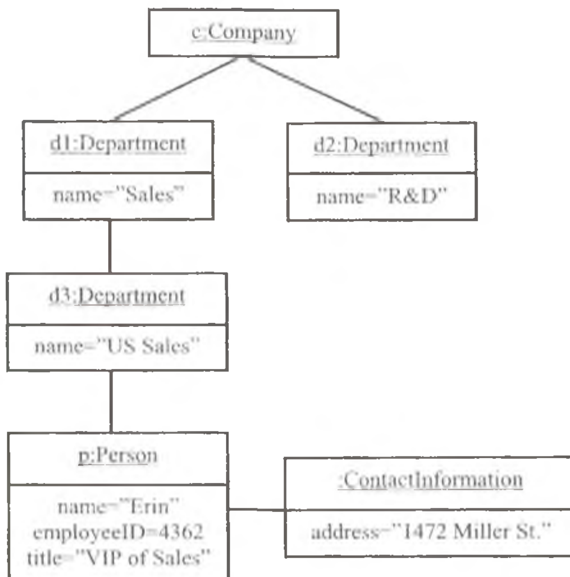

Рис. 16 Диаграмма объектов

Моделирование объектной структуры осуществляется следующим образом:

1. Идентифицируйте механизм, который собираетесь моделировать. Механизм представляет собой некоторую функцию или поведение части моделируемой системы, являющееся результатом взаимодействия сообщества классов, интерфейсов и других сущностей.
2. Для каждого обнаруженного механизма идентифицируйте классы, интерфейсы и другие элементы, участвующие в кооперации, а также отношения между ними.
3. Рассмотрите один из сценариев использования работы механизма. "Заморозьте" этот сценарий в некоторый момент времени и изобразите все объекты, участвующие в механизме.
4. Покажите состояние и значения атрибутов каждого такого объекта, если это необходимо для понимания сценария.
5. Покажите также связи между этими объектами, которые представляют экземпляры существующих ассоциаций.

Следует отдавать себе отчет в том, что во всех системах, кроме самых тривиальных, существуют сотни, а то и тысячи объектов, большая часть которых анонимна. Полностью специфицировать все объекты системы и все способы, которыми они могут быть ассоциированы, невозможно. Следовательно, диаграммы объектов должны отражать только некоторые конкретные объекты или прототипы, входящие в состав работающей системы.

Диаграммы прецедентов

Диаграммой прецедентов, или использования (Use case diagram), называется диаграмма, на которой показана совокупность прецедентов и актеров, а также отношения между ними.

Актеры и прецеденты связываются между собой **только отношениями ассоциации**, между тем между прецедентами допускается использование операций обобщения, включения и расширения.

Отношение обобщения между прецедентами аналогично отношениям обобщения между классами. Это означает, что прецедент-потомок наследует поведение и семантику своего родителя, может замещать его или дополнять его поведение.

Отношение включения между прецедентами означает, что в некоторый базовый прецедент инкорпорировано поведение другого прецедента. Включаемый прецедент никогда не существует автономно. Благодаря наличию отношений включения удается избежать многократного описания одного и того же потока событий, поскольку общее поведение можно опи-

сать в виде самостоятельного прецедента. Отношения включения изображаются в виде зависимостей со стереотипом "include".

Отношение расширения применяют для моделирования таких частей прецедента, которые пользователь воспринимает как необязательное поведение системы. Тем самым можно разделить обязательное и необязательное поведение. Отношение расширения изображают в виде зависимости со стереотипом "extend". Ниже на рисунке 17 представлены три типа отношений между прецедентами.

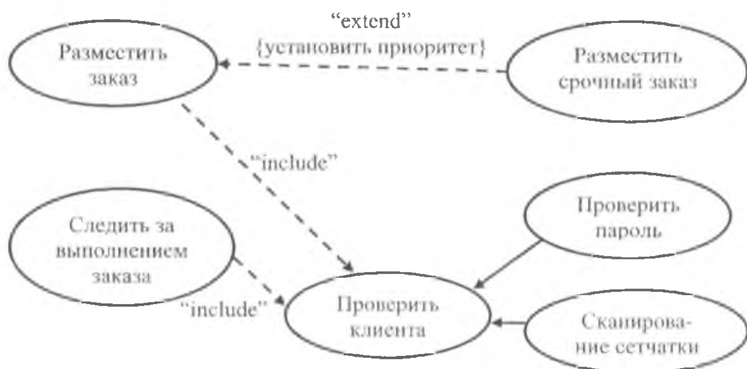

Рис. 17 Обобщения, включения и расширения

Отношения между актерами могут включать в себя операцию обобщения. Внешний вид диаграммы прецедентов представлен на рисунке 18.

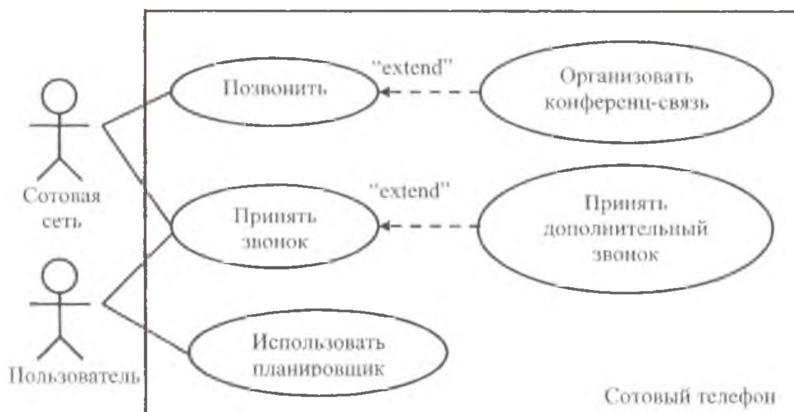

Рис. 18 Диаграмма прецедентов

Типичные примеры применения

Диаграммы прецедентов, или использования, применяют для моделирования статического вида системы с точки зрения прецедентов. Этот вид охватывает главным образом поведение системы, то есть видимые извне сервисы, предоставляемые системой в контексте ее окружения.

При моделировании статического вида системы с точки зрения прецедентов диаграммы использования обычно применяются двумя способами:

- ***для моделирования контекста системы***. Моделирование контекста подразумевает, что мы обводим систему воображаемой линией и выявляем актеров, которые находятся за этой линией и взаимодействуют с системой. Диаграммы прецедентов нужны на этом этапе для идентификации актеров и семантики их ролей;
- ***для моделирования требований к системе***. Моделирование требований к системе предполагает указание на то, что система должна делать (с точки зрения внешнего наблюдателя), независимо от того, как она должна это делать. Диаграммы прецедентов нужны здесь для специфицирования желаемого поведения системы. Они позволяют рассматривать всю систему как "черный ящик": вы видите все, что находится вне нее, наблюдаете за ее реакцией на события, но ничего не знаете о ее внутреннем устройстве.

Моделирование контекста системы состоит из следующих шагов:

1. Идентифицируйте окружающих систему актеров. Для этого нужно найти группы, которым участие системы требуется для выполнения их задач; группы, которые необходимы для осуществления системой своих функций
2. Организуйте похожих актеров с помощью отношений обобщения/специализации.
3. Введите стереотипы для каждого актера, если это облегчает понимание.
4. Поместите актеров на диаграмму прецедентов и определите способы их связи с прецедентами системы.

Моделирование требований к системе производится следующим образом:

1. Установите контекст системы, идентифицировав окружающих ее актеров.
2. Для каждого актера рассмотрите поведение, которого он ожидает или требует от системы

3. Поименуйте эти общие варианты поведения как прецеденты.
4. Выделите общее поведение в новые прецеденты, которые будут использоваться другими; выделите вариации поведения в новые прецеденты, расширяющие основные потоки событий.
5. Смоделируйте эти прецеденты, актеры и отношения между ними на диаграмме прецедентов.
6. Дополните прецеденты примечаниями, описывающими нефункциональные требования; некоторые из таких примечаний можно присоединить к системе в целом.

Диаграммы взаимодействий

Диаграммы последовательностей и **кооперации** (и те, и другие называются диаграммами взаимодействий) относятся к числу пяти видов диаграмм, применяемых в UML для моделирования динамических аспектов системы. На диаграммах взаимодействий показывают связи, включающие множество объектов и отношений между ними, в том числе сообщения, которыми объекты обмениваются. При этом **диаграмма последовательностей акцентирует внимание** на временной упорядоченности сообщений, а **диаграмма кооперации** - на структурной организации посылающих и принимающих сообщения объектов.

Диаграммы взаимодействий могут существовать автономно и служить для визуализации, специфицирования, конструирования и документирования динамики конкретного множества объектов, а могут использоваться для моделирования отдельного потока управления в составе прецедента.

Диаграммы взаимодействий важны не только для моделирования динамических аспектов системы, но и для создания исполняемых систем посредством прямого и обратного проектирования. Диаграмма взаимодействий описывает взаимодействия, состоящие из множества объектов и отношений между ними, включая сообщения, которыми они обмениваются.

Диаграммой последовательностей называется диаграмма взаимодействий, акцентирующая внимание на временной упорядоченности сообщений.

Графически такая диаграмма представляет собой таблицу, объекты в которой располагаются вдоль оси X, а сообщения в порядке возрастания времени - вдоль оси Y. Обычно инициирующий взаимодействие объект размещают слева, а остальные - правее (тем дальше, чем более подчиненным является объект). Затем вдоль оси Y размещаются сообщения, которые объекты посылают, причем более поздние оказываются ниже. Это дает наглядную картину, позволяющую понять развитие потока управления во времени. Внешний вид диаграммы последовательностей приводится на рисунке 19.

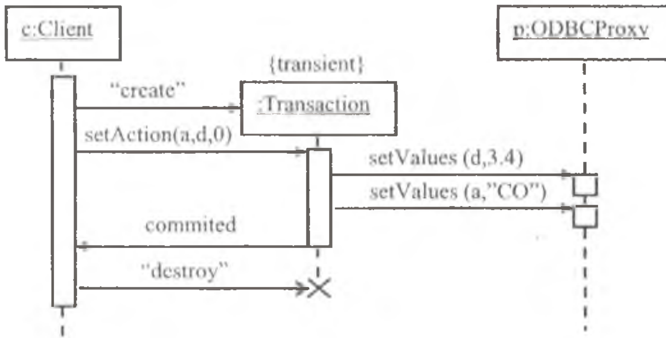

Рис. 19 Диаграмма последовательностей

Диаграммы последовательностей характеризуются двумя особенностями, отличающими их от других диаграмм.

Во-первых, на них показана **линия жизни объекта**. Это вертикальная пунктирная линия, отражающая существование объекта во времени. Большая часть объектов, представленных на диаграмме взаимодействий, существует на протяжении всего взаимодействия, поэтому их изображают в верхней части диаграммы, а линии жизни прорисованы сверху донизу. Объекты могут создаваться и во время взаимодействий. Линии жизни таких объектов начинаются с получения сообщения со стереотипом "create". Объекты могут также уничтожаться во время взаимодействий; в таком случае их линии жизни заканчиваются получением сообщении стереотипом "destroy", а в качестве визуального образа используется большая буква X, обозначающая конец жизни объекта.

Вторая особенность этих диаграмм - **фокус управления**. Он изображается в виде вытянутого прямоугольника, показывающего промежуток времени, в течение которого объект выполняет какое-либо действие, непосредственно или с помощью подчиненной процедуры. Верхняя грань прямоугольника выравнивается по временной оси с моментом начала действия, нижняя – с моментом завершения (и может быть помечена сообщением о возврате).

Диаграммой кооперации называется диаграмма взаимодействий, основное внимание в которой уделяется структурной организации объектов, принимающих и отправляющих сообщения. Графически такая диаграмма представляет собой граф из вершин и ребер.

Для создания диаграммы кооперации нужно расположить участвующие во взаимодействии объекты в виде вершин графа. Затем связи, соединяющие эти объекты, изображаются в виде дуг этого графа. Наконец, связи дополняются сообщениями, которые объекты принимают и посылают. Это

дает пользователю ясное визуальное представление о потоке управления в контексте структурной организации кооперирующихся объектов. Ниже на рисунке 20 приводится внешний вид диаграммы кооперации.

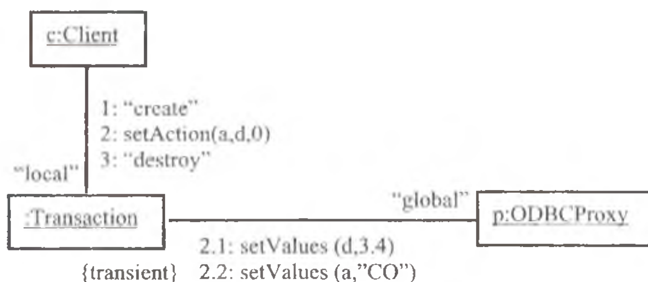

Рис. 20 Диаграмма кооперации

У диаграмм кооперации есть два свойства, которые отличают их от других диаграмм.

Первое - это **путь**. Для описания связи одного объекта с другим к дальней концевой точке этой связи можно присоединить стереотип пути (например, "local", показывающий, что помеченный объект является локальным по отношению к отправителю сообщения).

Второе свойство - это **порядковый номер сообщения**. Для обозначения временной последовательности перед сообщением можно поставить номер (номер начинается с единицы), который должен постепенно возрастать для каждого нового сообщения (2, 3 и.т.д.). Для обозначения вложенности используется десятичная нотация (1 - первое сообщение; 1.1- первое сообщение, вложенное в сообщение 1; 1.2- второе сообщение, вложенное в сообщение 1 и т.д.). Уровень вложенности не ограничен. Для каждой связи можно показать несколько сообщений (вероятно, посылаемых разными отправителями), и каждое из них должно иметь уникальный порядковый номер.

Поскольку диаграммы последовательностей и кооперации используют одну и ту же информацию из метамодели UML, они семантически эквивалентны. Это означает, что можно преобразовать диаграмму одного типа в другой без какой-либо потери информации, что и было показано на двух предыдущих рисунках. Это не означает, однако, что на обеих диаграммах представлена в точности одна и та же информация. Так, на упомянутых рисунках диаграмма кооперации показывает, как связаны объекты (обратите внимание на стереотипы "local" и "global"), а соответствующая диаграмма последовательностей - нет. С другой стороны, на диаграмме последовательностей могут быть показаны возвращаемые сообщений (сообщение "committed"), а на соответствующей диаграмме кооперации они отсут-

ствуют. Таким образом, можно сказать, что диаграммы обоих типов используют одну модель, но визуализируют разные ее особенности.

Диаграммы деятельности

Диаграммы деятельности - это один из пяти видов диаграмм, применяемых для моделирования динамических аспектов поведения системы. Как правило, они применяются, чтобы промоделировать последовательные (а иногда и параллельные) шаги вычислительного процесса. С помощью диаграмм деятельности можно также моделировать жизнь объекта, когда он переходит из одного состояния в другое в разных точках управления. Диаграммы деятельности могут использоваться самостоятельно для визуализации, специфицирования, конструирования и документирования динамики совокупности объектов, но они пригодны также и для моделирования потока управления при выполнении некоторой операции. Если в диаграммах взаимодействий акцент делается на переходах потока управления от объекта к объекту, то диаграммы деятельности описывают переходы от одной деятельности к другой.

Деятельность (Activity) - это некоторый относительно продолжительный отрезок выполнения в автомате. В конечном итоге деятельность сводится к некоторой последовательности действий, которая составлена из атомарных вычислений приводящих к изменению состояния системы или возврату значения. Действие может заключаться в вызове другого действия, посылке сигнала, создании или уничтожении объекта либо в простом вычислении.

Диаграммы деятельности важны не только для моделирования динамических аспектов поведения системы, но и для построения выполняемых систем посредством прямого и обратного проектирования.

Диаграмма деятельности - это диаграмма, показывающая поток переходов от одной деятельности к другой.

Внешний вид диаграммы деятельности представлен на рисунке 21.

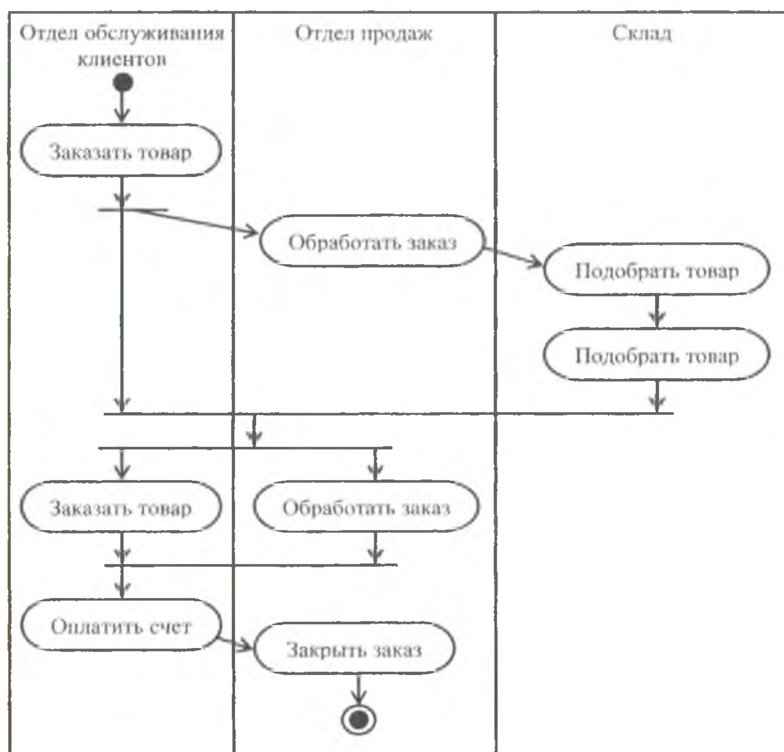

Рис. 21 Диаграмма деятельности

В потоке управления, моделируемом диаграммой деятельности, происходят различные события. Вы можете вычислить выражение, в результате чего изменяется значение некоторого атрибута или возвращается некоторое значение. Также, например, можно выполнить операцию над объектом, послать ему сигнал или даже создать его или уничтожить. Все эти выполняемые атомарные вычисления называются **состояниями действия**, поскольку каждое из них есть состояние системы, представляющее собой выполнение некоторого действия. Состояния действия изображаются прямоугольниками с закругленными краями. Внутри такого символа можно записывать произвольное выражение. Состояния действия не могут быть подвергнуты декомпозиции. Кроме того, они атомарны. Это значит, что внутри них могут происходить различные события, но выполняемая в состоянии действия работа не может быть прервана. И наконец, обычно предполагается, что длительность одного состояния действия занимает неощутимо малое время.

В противоположность этому, **состояния деятельности** могут быть подвержены дальнейшей декомпозиции. Состояния деятельности не являются атомарными, то есть могут быть прерваны. Предполагается, что для их завершения требуется заметное время. Можно считать, что состояние действия - это частный вид состояния деятельности, а конкретнее - такое состояние, которое не может быть подвергнуто дальнейшей декомпозиции. А состояние деятельности можно представлять себе как составное состояние, поток управления которого включает только другие состояния деятельности и действий. Состояния деятельности и действий обозначаются одинаково, с тем отличием, что у первого могут быть дополнительные части, такие как действия входа и выхода.

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 в другое. Переход представляется простой линией со стрелкой. Такие переходы еще называются **переходами по завершению** или **нетриггерными**.

Простые последовательные переходы встречаются наиболее часто, но их одних недостаточно для моделирования любого потока управления. Как и в блок-схеме, вы можете включить в модель ветвление, которое описывает различные пути выполнения в зависимости от значения некоторого булевского выражения. Точка ветвления представляется ромбом. В точку ветвления может входить ровно один переход, а выходить - два или более. Для каждого исходящего перехода задается булевское выражение, которое вычисляется только один раз при входе в точку ветвления. Ни для каких двух исходящих переходов эти сторожевые условия не должны одновременно принимать значение "истина". Но эти условия должны покрывать все возможные варианты, иначе поток остановится.

Для удобства разрешается использовать ключевое слово “else” для пометки того из исходящих переходов, который должен быть выбран в случае, если условия, заданные для всех остальных переходов, не выполнены.

Простые и ветвящиеся последовательные переходы в диаграммах деятельности используются чаще всего. Однако можно встретить и параллельные потоки, что особенно характерно для моделирования бизнес-процессов. В UML для обозначения разделения и слияния таких параллельных потоков выполнения использую синхронизационная черта, которая рисуется в виде жирной вертикальной или горизонтальной линии. Каждый из параллельно выполняющихся потоков управления существует в контексте независимого от активного объекта, который, как правило, моделируется либо процессом, либо вычислительной нитью.

Из рисунка также видно, что точка слияния представляет собой механизм синхронизации нескольких параллельных потоков выполнения. В эту

точку входят два или более перехода, а выходит ровно один. Выше точки слияния деятельности, ассоциированные с приходящими в нее путями, выполняются параллельно. В точке слияния параллельные потоки синхронизируются, то есть каждый из них ждет, пока все остальные достигнут этой точки, после чего выполнение продолжается в рамках одного потока.

При моделировании течения бизнес-процессов иногда бывает полезно разбивать состояния деятельности на диаграммах деятельности на группы, каждая из которых представляет отдел компании, отвечающий за ту или иную работу. В UML такие группы называются **дорожками**, поскольку визуально каждая группа отделяется от соседних вертикальной чертой, как плавательные дорожки в бассейне (см. рис. 21).

Каждой присутствующей на диаграмме дорожке присваивается уникальное имя. Никакой глубокой семантики дорожка не несет, разве что может отражать некоторую сущность реального мира. Каждая дорожка представляет сферу ответственности за часть всей работы, изображенной на диаграмме, и в конечном счете может быть реализована одним или несколькими классами. На диаграмме деятельности, разбитой на дорожки, каждая деятельность принадлежит **ровно одной дорожке**, но переходы могут пересекать границы дорожек.

Типичные примеры применения

Диаграммы деятельности используются для моделирования динамических аспектов системы. Эти динамические аспекты могут предполагать деятельность любого уровня абстракции любого вида системной архитектуры, включая классы (в том числе активные, интерфейсы, компоненты и узлы).

Использовать диаграмму деятельности для моделирования некоторого динамического аспекта системы можно в контексте практически любого элемента модели. Но чаще всего они рассматриваются в контексте системы в целом, подсистемы, операции или класса.

При моделировании динамических аспектов системы диаграммы деятельности применяются в основном двумя способами:

- **для моделирования рабочего процесса**. Здесь внимание фокусируется на деятельности с точки зрения актеров, которые сотрудничают с системой. Рабочие процессы часто оказываются с внешней, обращенной к пользователю стороны программной системы. Для такого применения диаграмм деятельности моделирование траекторий объектов имеет особенно важное значение;
- **для моделирования операции**. В этом случае диаграммы деятельности используются как блок-схемы для моделирования деталей вы-

числений. Для такого применения особенно важно моделирование точек ветвления, разделения и слияния. При этом контексте диаграммы деятельности включает параметры операции и ее локальные объекты.

Диаграмма состояний

Диаграмма состояний показывает автомат, фокусируя внимание на потоке управления от состояния к состоянию.

Автомат - это описание последовательности состояний, через которые проходит объект на протяжении своего жизненного цикла, реагируя на события, в том числе описание реакций на эти события.

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

Событие – это спецификация существенного факта, который происходит во времени и пространстве. В контексте автоматов событие - это стимул, способный вызвать срабатывание перехода.

Переход - 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.

Ниже на рисунке 22 приведен внешний вид диаграммы состояний.

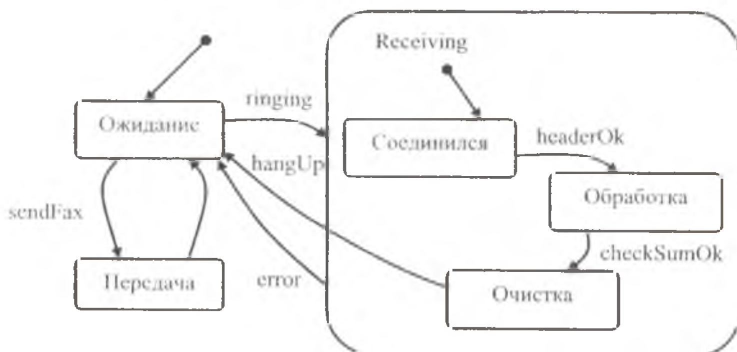

Рис. 22 Диаграмма состояний

Типичные примеры использования

Диаграммы состояний применяются для моделирования динамических аспектов системы. Использовать диаграммы состояний для моделирования некоторого динамического аспекта системы можно в контексте практически любого элемента модели. Обычно, однако, диаграммы состояний применяются в контексте системы в целом, подсистемы или класса. При моделировании динамических аспектов системы, класса или прецедента диаграммы состояний обычно используются только с целью моделирования ***реактивных объектов***.

Реактивный объект - это такой объект, поведение которого лучше всего характеризовать его реакцией на внешние события.

Как правило, реактивный объект находится в состоянии ожидания, пока не получит событие, а когда это случается, его реакция зависит от предшествующих событий. После того как объект отреагирует на событие, он снова переходит в состояние ожидания следующего события. Для таких объектов интерес представляют прежде всего устойчивые состояния, события, инициирующие переходы из одного состояния в другое, и действия, выполняемые при смене состояния.

В то время как взаимодействия применяются для моделирования поведения сообщества объектов, совместно решающих некоторую задачу, диаграммы состояний предназначены для моделирования поведения одного объекта на протяжении его жизненного цикла. Если диаграммы деятельности моделируют поток управления от деятельности к деятельности, то диаграммы состояний - поток управления от события к событию.

При моделировании поведения реактивного объекта нужно специфицировать главным образом три вещи:

- устойчивые состояния, в которых может находиться объект,
- события, которые инициируют переходы из одного состояния в другое,
- действия, выполняемые при каждой смене состояния.

Моделирование реактивного объекта подразумевает моделирование всего его жизненного цикла, начиная с момента создания и вплоть до уничтожения, с особым акцентом на устойчивые состояния, в которых может находиться объект.

Устойчивое состояние - такое, в котором объект может находиться неопределенно долгое время.

Диаграмма компонентов

Диаграмма компонентов показывает набор компонентов и отношения между ними.

Диаграммы компонентов обычно включают в себя: компоненты, интерфейсы, отношения зависимости, обобщения, ассоциации и реализации. Диаграммы компонентов могут также содержать пакеты и подсистемы. Внешний вид диаграммы компонентов приведен на рисунке 23.

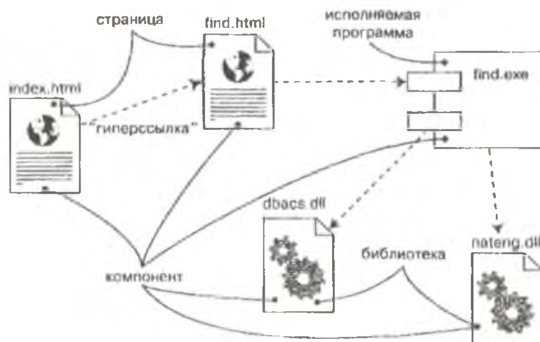

Рис. 23 Диаграмма компонентов

Диаграммы компонентов используются для моделирования статического вида системы с точки зрения реализации и как правило, используются в следующих случаях:

- моделирование исходного кода
- моделирование исполняемых версий.
- моделирование физических баз данных.

Диаграмма развертывания

Диаграммы развертывания, или применения, - это один из двух видов диаграмм, используемых при моделировании физических аспектов объектно-ориентированной системы. Такая диаграмма показывает конфигурацию узлов, где производится обработка информации, и то, какие компоненты размещены на каждом узле.

Ниже на рисунке 24 приведен внешний вид диаграммы развертывания.

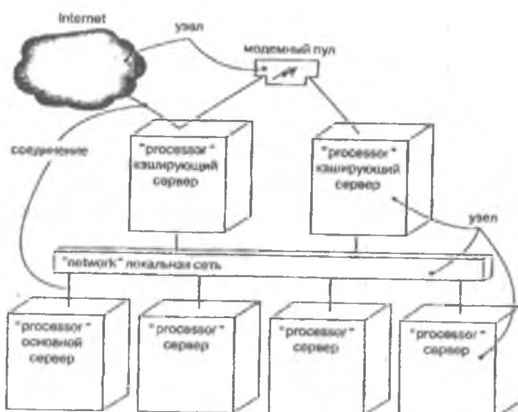

Рис. 24 Диаграмма развертывания

Диаграммы развертывания используются для моделирования статического вида системы с точки зрения развертывания. Это представление в первую очередь обращено на распределение, поставку и установку частей, из которых состоит физическая система.

Есть несколько типов систем, для которых диаграммы развертывания не нужны. Если вы разрабатываете программу, исполняемую на одной машине и обращающуюся только к стандартным устройствам на этой же машине, управление которыми целиком возложено на операционную систему (возьмем для примера клавиатуру, дисплей и модем персонального компьютера), то диаграммы развертывания можно игнорировать. Но если разрабатываемая программа обращается к устройствам, которыми операционная система обычно не управляет, или эта программа физически размещена на разных процессорах, то диаграмма развертывания поможет выявить отношения между программными и аппаратными средствами.

При моделировании статического вида системы с точки зрения развертывания диаграммы развертывания используются, как правило, в трех случаях:

- моделирование встроенных систем. **Встроенной системой** называется аппаратный комплекс, взаимодействующий с физическим миром, в котором велика роль программного обеспечения. Встроенные системы управляют двигателями, приводами и дисплеями, а сами управляются внешними стимулами, например датчиками температуры и перемещения. Диаграмму развертывания можно использовать

для моделирования устройств и процессоров, из которых состоит встроенная система;

- моделирование клиент-серверных (client/server) систем. Клиент-серверная система- это типичный пример архитектуры, где основное внимание уделяется четкому разделению обязанностей между интерфейсом пользователя, существующим на клиенте, и хранимыми данными системы, существующими на сервере.
- моделирование полностью распределенных систем.

5. ВВЕДЕНИЕ В RUP

Сегодня трудно найти книгу или статью, посвященную технологии создания больших программных систем, которая не цитировала бы убийственную характеристику современной ситуации в области разработки программных систем: “По нашим оценкам только 26% проектов создания ИС заканчиваются успешно”.

Большинство авторов приблизительно одинаково оценивают главную причину такого положения вещей. Это - возросшая сложность создаваемых систем. Проявлениями этой сложности являются не только увеличение размеров и функциональности. Не меньше, если не больше, проблем порождается частой сменой потребностей пользователей и ростом требований к качеству систем.

Традиционные способы разрешения проблемы сложности, такие как увеличение коллективов разработчиков, их специализация, распределение работ в чистом виде, ведут к еще большим трудностям согласования результатов и сборки готовых систем.

Серьезным шагом к победе над сложностью явилось появление объектно-ориентированного программирования (ООП). Но только шагом. Появление ООП не устранило проблем недостаточного взаимопонимания разработчиков и пользователей, неэффективного управления разработкой в условиях изменяющихся требований, неконтролируемости изменений в процессе выполнения работ, субъективности в оценке качества продуктов разработки и т.д.

И, тем не менее, 26% проектов заканчиваются успешно! Почему же все остальные снова и снова наступают на те же грабли? Почему они настойчиво продолжают учиться только на своих ошибках? Ведь решение лежит на поверхности: нужно сформировать и документировать набор проверенных на практике принципов, методов и процессов качественной и производительной работы над проектами по созданию программного обеспечения. Этому и будет посвящен настоящий раздел.

Что такое Rational Unified Process?

Rational Unified Process - это процесс разработки программного обеспечения.

Процесс - частично упорядоченный набор шагов, которые нужно проделать для достижения цели; при разработке программного обеспечения цель состоит в формировании или расширении существующего программного изделия.

Когда программная система разрабатывается на пустом месте, ее разработка - это процесс создания системы из требований. Но как только система приняла какую-то форму, т.е. прошла начальный цикл развития, то любая доработка - это процесс приспособления системы к новым или изменившимся требованиям. Разработка и все последующие доработки составляют **жизненный цикл системы**.

Rational Unified Process обеспечивает строгий подход к назначению задач и ответственности в пределах группы разработки. Его цель состоит в том, чтобы гарантировать высокое качество программного продукта, отвечающего потребностям конечных пользователей, в пределах предсказуемого временного графика и бюджета.

Архитектура процесса

Два измерения

Rational Unified Process представляет процесс разработки программной системы в двух измерениях:

- По содержанию действий участников групп разработки (по основным потокам работ)
- Во времени (по стадиям жизненного цикла разрабатываемой системы).

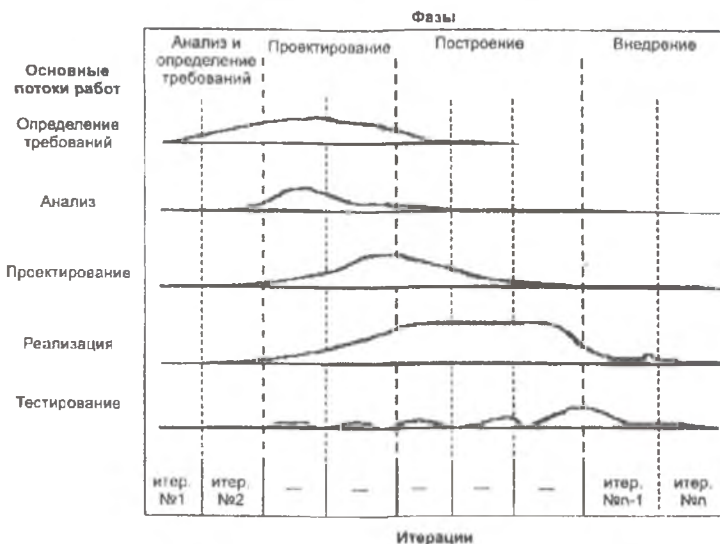

Рис. 25 Организация процесса по содержанию (основные потоки работ) и во времени (стадии)

Первое измерение представляет статический аспект процесса: оно описано в терминах основных потоков работ (исполнители, действия, их последовательность и так далее).

Второе измерение представляет динамический аспект процесса, поскольку оно выражено в терминах циклов, стадий, итераций и этапов.

Статическое содержание процесса

Статическое содержание процесса организовано в основных потоках работ. Rational Unified Process включает девять основных потоков работ; шесть потоков работ процесса разработки:

- Деловое моделирование
- Требования
- Анализ и проектирование
- Выполнение
- Испытание
- Развертывание

и три потока работ поддержки:

- Управление конфигурацией и изменением
- Управление проектом
- Среда

Основные потоки работ описаны в терминах работников, действий и артефактов.

- Работник определяет поведение и ответственности индивидуума или нескольких индивидуумов, работающих вместе как группа. Это - важное отличие, потому что обычно о работнике думают как о конкретном человеке или группе. В Rational Unified Process, работник - больше роль, которая определяет, как индивидуумы должны выполнять работу.

С каждым работником ассоциируется набор связанных действий; связанность означает, что действия будут лучше выполнены этим индивидуумом. Ответственности каждого работника обычно определяются относительно некоторых артефактов, например документов. Примеры работников: аналитик делового процесса, проектировщик, архитектор или инженер-технолог.

Структура жизненного цикла

Когда мы рассматриваем динамическую организацию процесса во времени, жизненный цикл программы разбивается на циклы, каждый из которых работает над новым поколением изделия.

Rational Unified Process делит один цикл развития на четыре последовательных стадии:

- Начало
- Уточнение
- Конструирование
- Переход

Каждая стадия заканчивается четко определенной вехой - временной точкой, в которой должны быть приняты некоторые критические решения, а поэтому ключевые цели должны быть достигнуты.

На **начальной стадии** необходимо установить деловые применения системы и определить рамки проекта. Чтобы сделать это, следует идентифицировать все внешние объекты, с которыми взаимодействует система (субъекты), и определить характер этого взаимодействия на высоком уровне. Эта работа включает идентификацию всех прецедентов и описание нескольких наиболее существенных. Деловое применение включает критерии успеха, оценку риска, оценку необходимых ресурсов и укрупненный план с указанием дат завершения главных этапов. В конце начальной стадии необходимо исследовать требования жизненного цикла проекта и решить, следует ли продолжать разработку.

Стадия уточнения Цели стадии уточнения состоят в том, чтобы проанализировать прикладную область, создать нормальную архитектурную основу, разработать план проекта и устранить самые высокие элементы риска проекта. Архитектурные решения должны приниматься с пониманием целостной системы. Это подразумевает, что должны быть описаны большинство прецедентов и приняты во внимание некоторые из связей. В конце стадии уточнения происходит детальное исследование цели и контекста системы, архитектурных решений и способы разрешения главных рисков.

Стадия конструирования. На данной стадии происходит итерационная разработка законченного изделия, которое должно быть готово к передаче пользователям. Это подразумевает описание остающихся прецедентов, изложение деталей конструкции, завершение выполнения и проверку программного обеспечения. В конце стадии конструирования необходимо решить, все ли программное обеспечение, рабочие места и пользователи готовы и работоспособны.

Стадия перехода. В процессе стадии перехода происходит передача программного обеспечения пользователям. Как только изделие попадает в руки конечных пользователей, часто возникают новые проблемы, которые требуют дополнительной разработки по корректировке системы, исправлению необнаруженных ранее проблем или завершению реализации некоторых возможностей, которые, возможно, были отложены. Эта стадия обычно начинается с выпуска “бета-версии” системы. В конце переходной стадии необходимо решить, достигнуты ли цели жизненного цикла, и возможно, приступить к запуску другого цикла разработки.

Итерации

Каждая стадия Rational Unified Process может быть в свою очередь разбита на итерации. **Итерация** - это законченный цикл разработки, приводящий к выпуску выполнимого изделия (внутренней или внешней версии) или подмножества конечного продукта, которое возрастает с приращением от итерации к итерации, чтобы стать законечной системой.

Каждая итерация в пределах стадии приводит к выпуску выполнимой системы. Каждая итерация содержит все аспекты разработки программного обеспечения и повторяет все основные потоки работ. Но акценты на основных потоках работ различны, в зависимости от стадии разработки.

Итеративная разработка программного обеспечения

Процесс “водопада”

До недавнего времени программные системы строились по так называемому принципу “водопада”, согласно которому процесс разработки сводится к следующей последовательности действий:

1. Исследуйте проблему, которая должна быть решена, все требования и их связи. Зафиксируйте их в виде задания на разработку и заставьте все заинтересованные стороны согласиться, что это именно то, что они хотят получить.
2. Разработайте проектные решения, удовлетворяющие всем требованиям и связям. Тщательно исследуйте эту конструкцию и удостоверьтесь, что все заинтересованные стороны согласны с правильностью Ваших решений.
3. Осуществите проектные решения, используя лучшие методы кодирования.
4. Убедитесь, что все заявленные требования выполнены и ваша работа удовлетворяет заказчика.
5. Передайте заказчику готовый продукт.

Именно так были построены небоскребы и космические корабли. Этот способ очень хорош, но только потому, что прикладная область давно известна: за плечами инженеров сотни лет проектирования и конструирования.

Специалисты по программному обеспечению не имели такой возможности. Не имея своего, они не могли не воспользоваться положительным опытом специалистов из смежных областей. Несколько лет назад многие из разработчиков и представить себе не могли, что процесс “водопада” не является единственным разумным подходом. А если бы и смогли, это не изменило бы ситуацию, т.к. метод “водопада” заложен во все ГОСТы и методики проектирования программных систем. Если процесс “водопада” идеален, тогда почему большинство проектов, особенно проектов крупных систем, заканчиваются неудачей?

Тому есть несколько причин:

- Контекст программирования существенно отличается от такового других технических дисциплин.
- Мы не сумели учесть некоторые факторы, связанные с человеком (изменяющиеся требования).
- Мы пытаемся использовать подход, четко работающий в определенных обстоятельствах, вне этих обстоятельств.

Преимущества итераций

Итерационный подход превосходит принцип "водопада" по следующим причинам:

- Он позволяет принимать во внимание изменяющиеся требования. Действительность состоит в том, что требования изменяются. Изменение требований и их "ползучесть" всегда были первоисточниками неудач проектов; они ведут к опозданиям поставок, нарушениям планов, неудовлетворенности заказчиков и бесполезности разработки.
- Элементы интегрируются постепенно. Итерационный подход - почти непрерывная интеграция.
- Он позволяет смягчать риски уже на начальных стадиях проекта, выявляя многие аспекты проекта: пригодность инструментальных средств и имеющегося в наличии программного обеспечения, навыки людей и т.д. Некоторые предполагаемые риски могут не проявиться, и могут появляться новые, не предполагавшиеся ранее.
- Он облегчает многократное использование компонентов - проще идентифицировать общие части, когда они частично разработаны или реализованы, чем идентифицировать все общие части с самого начала. Идентификация и разработка многократно используемых частей трудна. Критический анализ проектных решений на начальных итерациях позволяет архитекторам идентифицировать потенциальное многократное использование и разрабатывать и совершенствовать общий код в последующих итерациях.
- Он приводит к более устойчивой архитектуре - в нескольких итерациях возможно исправление большего количества ошибок. Слабые места обнаруживаются даже в ранних итерациях. Одновременно обнаруживаются и еще могут быть легко исправлены критические параметры эффективности, отпадает необходимость делать это накануне развертывания.
- Разработчики учатся по ходу и более полно используют различные навыки и специальные знания в течение всего жизненного цикла. Испытатели раньше запускают тесты, технические специалисты раньше делают свои записи и т.д.

Организаторы проекта часто сопротивляются итерационному подходу, видя в нем своего рода бесконечное "перемалывание" сделанного. В Rational Unified Process интерактивный подход очень управляем; итерации планируются по числу, продолжительности и цели. Задачи и ответственности участников определены. Объективные критерии продвижения зафик-

сированы. Некоторая доработка от одной итерации до другой имеет место, но она также тщательно управляется.

Управление требованиями

Управление требованиями - это систематический подход к обнаружению, документированию, организации и сопровождению изменяющихся требований к системе.

Требование - это условие или возможность, которой должна соответствовать система.

Сбор требований может показаться довольно тривиальной задачей. Однако в реальных проектах разработчики неизбежно сталкиваются с существенными трудностями потому, что:

- Требования не всегда очевидны и могут исходить из разных источников
- Требования не всегда удается ясно выразить словами.
- Число требований может стать неуправляемым, если ими не управлять.
- Требования имеют уникальные свойства или значения свойств. Например, они не являются ни одинаково важными, ни одинаково простыми для согласования.
- Требования изменяются.

Какими навыками нужно обладать, чтобы справиться с этими трудностями? Наиболее важные из них это умение:

- Анализировать проблемы
- Определять потребности совладельцев
- Управлять контекстом проекта (приоритеты)
- Управлять изменением требований

Управление изменением требований

Независимо от того, насколько Вы осторожны при определении ваших требований, всегда будут вещи, которые потребуют изменения. Что делает изменяющиеся требования сложными для управления? Изменение требования означает не только то, что должно быть потрачено больше или меньше времени на реализацию новой конкретной возможности, но также и то, что изменение в одном требовании может вступить в конфликт с другими требованиями. Вы должны удостовериться, что Вы придаете вашим требованиям структуру, которая гибка к изменениям, и что Вы используете

ссылки трассируемости для представления зависимостей между требованиями и другими артефактами разработки.

Управление изменением включает действия подобные базированию, определению зависимостей, которые важно проследить, устанавливая трассируемость между связанными предметами и управлением изменением.

Архитектура приложения

Rational Unified Process от начала до конца жизненного цикла управляется прецедентами, но действия проектирования сосредоточены вокруг понятия архитектуры - архитектуры системы или архитектуры приложения. Основной упор на ранних итерациях процесса - главным образом в стадии уточнения - делается на производство и проверку правильности архитектуры приложения. В начальном цикле развития архитектурные решения материализуются в форме выполняемого архитектурного прототипа, который постепенно развивается, чтобы на более поздних итерациях стать законченной системой.

Что такое архитектура приложения?

Архитектура приложения - это понятие, которое интуитивно чувствует большинство проектировщиков, особенно опытных, и которое достаточно сложно точно определить. В частности, трудно провести четкую границу между проектом и архитектурой. Архитектура - один из аспектов проекта, который концентрируется на некоторых определенных особенностях.

В Rational Unified Process **архитектура системы** - это организация или структура существенных компонентов системы, взаимодействующих через интерфейсы с компонентами, составленными из последовательно меньших компонентов и интерфейсов.

Почему так нужен проект архитектуры?

- Наличие проекта архитектуры позволяет Вам иметь интеллектуальный контроль над проектом, управлять его сложностью и поддерживать целостность системы.

Сложная система - это нечто большее, чем сумма ее частей, чем последовательность маленьких независимых тактических решений. Она должна иметь некоторую объединяющую структуру, позволяющую организовать эти части и обеспечивать точные правила роста. Архитектура закладывает основу для понимания всего проекта, устанавливая общий набор справочников, общий словарь и т.д.

- Архитектура является эффективной базой для крупномасштабного многократного использования.

Ясно показывая крупные компоненты и критические интерфейсы между ними, архитектура позволяет Вам рассуждать о многократном использовании: о внутреннем многократном использовании - идентифицировать общие части, и о внешнем многократном использовании - объединять готовые изделия и имеющиеся в наличии компоненты. Она позволяет также многократное использование в большем масштабе: многократное использование самой архитектуры в потоке производства приложений, - т.е. в производстве, которое объединяет различные функциональные возможности в одной системе.

- Архитектура является базой для руководства проектом.

Планирование и персонал организуются по линиям крупных компонентов. Фундаментальные структурные решения принимаются небольшой централизованной архитектурной группой. Разработка разбивается на разделы среди маленьких групп, каждая из которых отвечает за одну или несколько частей системы.

Описание архитектуры

Архитектура приложения описывается множеством ее представлений. Каждое представление архитектуры отражает некоторый аспект, интересующий группу совладельцев проекта: конечных пользователей, проектировщиков, администраторов, системотехников, эксплуатационщиков и т.д.

Представления фиксируют главные решения проектирования структуры, показывая, как приложение разделено на компоненты, и как связаны эти компоненты для получения полезной конфигурации. Эти решения должны вытекать из требований функциональности и других факторов. С другой стороны, эти решения устанавливают дальнейшие ограничения на требования и на будущие проектные решения нижнего уровня.

Представления архитектуры по существу являются выжимками, иллюстрирующими “архитектурно существенные” элементы моделей. В Rational Unified Process все начинается с типичного набора представлений, называемого моделью представления “4+1”.

Этот набор включает:

- **представление прецедентов**, которое содержит прецеденты и сценарии, охватывающие архитектурно существенное поведение, клас-

сы или технические риски. Оно является подмножеством модели прецедентов

- **логическое представление**, которое содержит наиболее важные классы проекта, их организацию в пакеты и подсистемы различных уровней. Здесь содержится уже некоторая реализация прецедента. Это представление является подмножеством модели проекта
- **представление выполнения**, которое содержит краткий обзор модели выполнения в терминах модулей пакетов и уровней. Здесь описывается также распределение пакстов и классов (из логического представления) в пакетах и модулях представления выполнения. Это представление является подмножеством модели выполнения
- **представление процесса**, которое содержит описание задач (процессов и нитей), их взаимодействия и конфигурации и распределения объектов и классов по задачам. Потребность этого представления возникает только, если система имеет существенную степень параллелизма. В Rational Unified Process 5.1 представление процесса - это подмножество модели проекта
- представление развертывания, которое содержит описание различных физических узлов для наиболее типичных конфигураций платформы, и расположение задач (из представления процесса) по физическим узлам. Потребность этого представления возникает для распределенных систем.

Внедрение Rational Unified Process

Для успешного внедрения нового процесса работы над проектом по разработке программного обеспечения (или над всеми проектами в организации), необходимо рассмотреть следующие проблемы:

- Поддержка со стороны администрации. Без соответствующей поддержки администрации трудно вводить новый процесс.
- Обучение участников проекта. Каждый участник проекта должен понимать новый процесс.
- Конфигурирование процесса для удовлетворения потребностей проекта. При конфигурировании необходимо рассмотреть такие факторы как прикладная область, разновидность приложения, культура компании и инструментальные средства поддержки.
- Инструментальные средства поддержки. Процесс разработки программного обеспечения будет гораздо эффективнее, если все действия жизненного цикла системы будут поддержаны соответствующими инструментальными средствами. Необходимо выбрать набор инструментальных средств поддержки и приспособить их для работы друг с другом в пределах процесса.

Важно, чтобы введение нового процесса и, возможно, приобретение нового набора инструментальных средств поддерживала администрация. Руководство должно знать все, что требуется, чтобы ввести новый процесс:

- Введение нового процесса всегда требует определенных капитальных затрат.
- Большинство людей стремится делать дело способами, с которыми они уже знакомы. Поэтому необходимо стимулировать членов группы придерживаться процесса. Самым лучшим способом внедрения RUP является создание, по крайней мере, одного полного проекта с последующей его оценкой на предмет того, что получилось хорошо и что может быть улучшено.
- Когда вводится новый процесс, всегда есть определенный риск. Один из способов уменьшить риск состоит в том, чтобы обеспечить соответствующее обучение и выполнить пробный проект. Другой способ снижения риска состоит в привлечении квалифицированных консультантов. Они должны быть способны идентифицировать и устранить некоторые из проблем прежде, чем они станут дорогостоящими проблемами.

Инструментальные средства поддержки

Процесс разработки программного обеспечения требует, чтобы инструментальные средства поддерживали все действия жизненного цикла системы. Итерационный процесс развития предъявляет специальные требования к используемым инструментальным средствам, типа их интеграции друг с другом и прямой и обратной разработки моделей и кодов. Rational Unified Process можно использовать совместно с рядом инструментальных средств фирмы Rational или других продавцов. Однако Rational предоставляет много хорошо интегрированных инструментальных средств, которые эффективно поддерживают Rational Unified Process. Ниже перечислены инструментальные средства, которые требуются при разработке.

- **Инструмент моделирования** для разработки различных моделей, типа модели прецедентов или модели проекта. Инструмент должен иметь возможность прямой и обратной разработки.

Пример: Rational Rose

- **Инструмент управления требованиями** для фиксации, организации, расположения по приоритетам и прослеживания всех требований.

Пример: Rational Requisite Pro

- **Инструмент документирования** для поддержки проектной документации.

Пример: Rational SoDA

- **Средства программирования**, которые должны быть интегрированы со средой моделирования и условиями проведения испытаний.

Примеры: Rational Apex/Ada, Rational Apex/C++ (Java ready)

- Инструментальные средства поддержки руководителя проекта при планировании и управлении проектом.

Пример: Microsoft Project

- **Инструмент управления задачами** может помочь организаторам проекта знать, какие задачи находятся в работе, привести их исполнение в соответствие некоторому процессу, и в то же самое время автоматизировать утомительные задачи, связанные с управлением конфигурацией. Он может также помочь руководству проекта непрерывно контролировать его продвижение.

Примеры: Rational ClearQuest

- **Инструмент управления конфигурацией** может помочь следить за всеми произведенными артефактами и их различными версиями. Очень важна интеграция среды кодирования, инструментальных средств моделирования и инструментальных средств управления конфигурацией.

Пример: ClearCase.

- **Инструментальные средства тестирования.** Важно использовать средства тестирования для автоматизации испытаний так, чтобы было возможно легко повторять испытания кода (возвратное тестирование) при минимизации затрат и увеличении качества. Более специализированные инструментальные средства позволяют выполнять испытания загрузки.

Примеры: Rational SQA Suite, Rational TestMate, Rational Visual Test

Поток работ “Управление проектом”

Руководство проектом программного обеспечения - искусство балансирования конкурирующих целей, управления рисками и преодоления ограничений ради поставки изделия, отвечающего потребностям обеих категорий заказчиков - оплачивающих счета и пользователей. Тот факт, что бесспорно успешных проектов так немного, объясняется чрезвычайной сложностью задачи.

Цель потока работ управления проектом программного обеспечения в Rational Unified Process состоит в том, чтобы упростить задачу, предоставляя некоторый контекст руководства проектом. Он не является панацеей для успеха, но он обеспечивает тот подход к управлению проектом, который заметно увеличит вероятность поставки удачного программного обеспечения.

С этой целью поток работ управления проектом Rational Unified Process:

- Предлагает структуру управления сложными (преимущественно программными) проектами;
- Предлагает практические рекомендации по планированию, укомплектованию персоналом, выполнению и мониторингу проектов.
- Предлагает структуру управления рисками.

Однако этот поток работ не пытается охватить все аспекты руководства проектом. Например, он не включает проблемы типа:

- Руководство людьми: наем, обучение, тренировка
- Управление бюджетом: определение, распределение и т.д.
- Управление контрактами с поставщиками и заказчиками.

Этот поток работ нацелен, главным образом, на самые важные аспекты итерационного процесса разработки:

- Управление рисками
- Планирование итерационного проекта: всего жизненного цикла и конкретной итерации
- Мониторинг продвижения итерационного проекта, метрики.

Краткий обзор действий

Работник: Руководитель проекта

Работник "Руководитель проекта" является ключевым действующим лицом этого потока работ. Он распределяет ресурсы, формирует приоритеты, координирует взаимодействия с заказчиками и пользователями и вообще руководит проектной группой, направляя ее действия в правильное русло. Кроме того, руководитель проекта устанавливает порядок действий, который гарантирует целостность и качество артефактов проекта, а также отвечает за эффективное выполнение процесса проверки изменения изделия.

Рис. 26 Диаграмма краткого обзора действий потока работ

Действие: Изложение деловых обстоятельств

Описание деловых обстоятельств документирует экономическую эффективность изделия. Это инструмент, посредством которого обосновывается размер капиталовложений в проект. Плохое документирование деловых обстоятельств может испортить лучшие проектные идеи, в то время как ыграмотное документирование деловых обстоятельств может гарантировать соответствующее финансирование достойных изделий.

Действие “Изложение деловых обстоятельств” включает следующие шаги:

- Описание изделия.
- Определение контекста предметной области.
- Определение цели создания изделия.
- Разработка финансового прогноза.
- Описание проектных ограничений.

Действие выполняется один раз в каждой итерации. Результатом действия является артефакт “Деловые обстоятельства”.

Действие: Идентификация рисков

Целью этого действия является идентификация рисков и прогнозирование их воздействия на проект. Концепция рисков является одной из важнейших для управления итеративным проектом.

Действие “Идентификация рисков” включает следующие шаги:

- Идентификация потенциальных рисков
- Группировка и сортировка рисков
- Идентификация стратегий предотвращения риска
- Идентификация стратегий уменьшения риска
- Идентификация стратегий локализации риска

Действие выполняется один раз в каждой стадии. Результатом действия является артефакт “Список рисков”.

Действие: Разработка плана проекта

В этом действии производится крупномодульный план проекта, фокусирующийся на главных вехах и ключевых выпусках в течение всего жизненного цикла.

Действие “Разработка плана проекта” включает следующие шаги:

- Определение вех проекта
- Определение целей вех
- Определение количества и длины итераций в пределах стадий
- Уточнение сроков вех и контекста
- Определение плана измерений

Действие выполняется один раз в проекте. Результатами действия являются артефакты “План проекта” и “План измерений”.

Действие: Формирование проектной группы

Цели этого действия состоят в планировании человеческих ресурсов по наборам навыков,
необходимых для проекта, и в группировке доступных ресурсов в относительно независимые, но сотрудничающие группы.

Действие “Формирование проектной группы” включает следующие шаги:

- Рассмотрение потребностей в персонале по стадиям
- Распределение работы среди работников
- Формирование групп
- Обучение персонала

Rational Unified Process описывает потоки работ в терминах работников и действий.

Связь индивидуума с работником через какое-то время может измениться под влиянием стадии жизненного цикла проекта и выполняемой работы.

- Индивидуум может действовать как несколько различных работников в течние одного и того же дня.
- Индивидуум может действовать как несколько работников одновременно.
- Несколько личностей могут действовать группой как один работник для совместного выполнения некоторых действий.

Необходимо распределить обязанности так, чтобы минимизировать передачу ответственности за артефакты от одного индивидуума к другому.

Профессиональный и количественный состав участников проектных групп зависит от стадии разработки. Если над проектом работает больше 5-7 человек, то необходимо разделить их на группы. За каждый артефакт (прецедент, подсистему, компонент) должен отвечать один человек или группа. Группы должны состоять минимально из двух и максимально из семи человек; группы более семи человек обычно естественным путем распадаются на подгруппы, так что лучше сделать это заранее.

Результатом действия является модифицированный артефакт "План проекта", в котором указано распределение человеческих ресурсов по работам во времени.

Действие: Разработка плана итерации

Целью этого действия является разработка подробного плана одиночной итерации, состоящего из детальной схемы разбиения работы на действия, распределения ответственности, внутренних вех итерации и выпусков, оценочных критериев для итерации. Сама по себе итерация - это набор очень точно выделенных для создания выполнимой программы действий. Акцент на выполнимости программы вынуждает к почти непрерывной интеграции и позволяет рано обнаруживать технические риски и уменьшать сопутствующие риски проекта. Выполнение итерации подразумевает некоторое количество доработки и сопровождается изменениями в характере доработки.

Действие "Разработка плана итерации" включает следующие шаги:

- Определение рамок итерации
- Определение критериев оценки итерации
- Определение действий итерации
- Распределение ответственности

Действие выполняется один раз в каждой итерации. Результатом действия является артефакт "План итерации".

Действие: Выполнение плана итерации

Цель этого действия - произвести выпуск выполнимой программы. Выполнение плана итерации просто: необходимо выполнить действия, определенные непосредственно в плане итерации. Периодическая оценка продвижения и результатов в контрольных точках с большой вероятностью гарантируют, что проект идет по графику. Малые отклонения от графика можно просто не учитывать, они являются результатом неточной оценки, но накопление сдвигов ведет к большим отклонениям от графика. Необходимо контролировать риски, отслеживая наиболее разрушительные.

Результатом действия является артефакт "Модель выполнения".

Действие: Оценка итерации

Оценка итерации фиксирует результат итерации, степень достижения оценочных показателей, полученные уроки и изменения, которые должны быть сделаны. Одно из основных достоинств итерационного подхода состоит в том, что итерации обеспечивают естественные вехи для оценки прогресса и ограничения риска. В пределах итерации необходимо продолжать оценивать прогресс и риск (возможно неформально), чтобы гарантировать, что трудности не разрушат проект.

Оценка итерации заключается в определении успеха или неудачи итерации и в изучении собранных данных с целью изменения проекта или улучшения процесса его разработки.

Действие "Оценка итерации" включает следующие шаги:

- Сбор результатов измерений
- Оценка результатов итерации
- Рассмотрение внешних изменений
- Исследование критериев оценки
- Создание запросов изменений

Действие выполняется один раз в каждой итерации. Результатами действия являются артефакт <Оценка итерации> и предложения по корректировке артефактов "Список рисков", "План проекта" и "План итерации" для следующей итерации.

Действие: Ревизия списка рисков

Цель этого действия - модифицировать список рисков для отражения текущего состояния проекта. Оценка рисков - скорее один непрерывный процесс, чем некоторое разовое действие, которое происходит только в определенных ключевых точках в ходе проекта.
Как минимум необходимо осуществлять следующие действия:

- Еженедельно ревизовать ваш список, чтобы отслеживать изменения.
- Сделать верхнюю часть списка из десяти пунктов видимыми для всего проекта в целом и настойчиво выполнять все необходимые для них действия.

В конце итерации необходимо пересмотреть список рисков с точки зрения целей итерации. Особенно важно:

- Исключить риски, которые были полностью смягчены.
- Ввести новые обнаруженные риски.
- Пересмотреть важность и переупорядочить список рисков.

Риски должны устойчиво уменьшаться по мере продолжения стадии уточнения и в течение конструирования. Если этого не происходит, значит, не представляется возможным обрабатывать риски соответствующим образом, или разрабатываемая система слишком сложна.

Действие "Ревизия списка рисков" включает следующие шаги:

- Ревизия рисков в течение итерации
- Ревизия рисков в конце итерации

Действие выполняется один раз в каждой итерации. Результатом действия является модифицированный артефакт "Список рисков".

Ключевая идея в управлении риском состоит в том, чтобы не ждать пассивно, когда риск осуществится и станет проблемой или уничтожит проект, а решить заранее, что с ним делать.

Существуют три основных возможности:

- **Предотвращение риска**: реорганизовать проект так, чтобы этот риск не мог на него воздействовать.
- **Передача риска**: реорганизовать проект так, чтобы кто-то другой или что-то другое имели с ним дело (заказчик, продавец, банк, другой элемент и т.д.)
- **Принятие и локализация риска**: учесть этот риск как неизбежный. Подготовить план действий на случай проявления этого риска и контролировать его симптомы.

При принятии риска необходимо сделать две вещи:

- Снизить риск: немедленно предпринять некоторые действенные шаги к снижению вероятности или влияния риска
- Определить план действий: какой способ действий выбрать, если риск станет фактической проблемой

План проекта

План проекта - крупномодульный план, который разрабатывается для всего проекта только один раз. Он полностью определяет границы проекта для одного цикла разработки и содержит:

- Даты главных вех:
  - целей жизненного цикла (конец стадии начала, рамки проекта и его финансирование)
  - архитектуры жизненного цикла (конец стадии уточнения, законченная архитектура)
  - начальной работоспособности (конец стадии конструирования, первая бета-версия)
  - выпуска изделия (конец стадии перехода и полного цикла разработки)
- Потребности в укомплектовании персоналом: какие ресурсы требуются и к какому времени
- Даты малых вех: конец каждой итерации и ее главные цели, если они известны

План проекта создается как можно раньше в стадии начала и модифицируется по мере необходимости.

План итерации

План итерации - мелкомодульный план, который создается один раз для каждой итерации. Обычно проект имеет два активных плана итерации одновременно:

- План текущей итерации, который используется, чтобы проследить продвижение итерации
- План следующей итерации, который обычно начинают формировать во второй половине текущей итерации и который бывает готов к концу текущей итерации

План итерации формируется для определения задач и их распределения между личностями и группами с использованием традиционных методов и инструментальных средств планирования (диаграммы Ганта и т.д.). План должен содержать важные даты, типа сроков завершения компонентов, получения компонентов от других организаций, главных обзоров и т.п.

Поскольку итеративный процесс является динамичным и, как предполагается, приспособлен к изменениям в целях и в тактике, нет необходимости тратить время на детальное планирование действий, которые “скрыты за горизонтом”. Эти планы все равно быстро устареют и будут игнорироваться проектной группой. План итерации охватывает реально управляемый отрезок времени и имеет нужную степень детализации для реального планирования.

Определение количества и длины итераций в пределах стадий

Когда будут определены длины стадий проекта, должны быть определены число итераций и их длина.

Длина итерации

Рекомендации по продолжительности выполнения итераций зависят, главным образом, от количества людей, занятых в проекте.

Играют роль и другие факторы: степень знакомства организации с итерационным подходом, включая наличие устойчивой и зрелой организации, уровень автоматизации, которую группа использует для управления кодом (например, наличие средств управления конфигурацией), обмена информацией (например, внутренняя сеть) и тестирования и т.д.

В зависимости от риска, размера и сложности проекта возможно множество вариантов. В Rational Unified Process описаны четыре различных стратегии построения жизненного цикла проекта.

Пошаговый жизненный цикл

Пошаговая стратегия определяет потребности пользователя, формулирует системные требования и затем исполняет остальную часть разработки в последовательных конструкциях. Первая конструкция содержит часть запланированных возможностей, следующая конструкция прибавляет новые возможности и так далее, пока система не будет закончена.

Для пошагового жизненного цикла характерны следующие итерации:

- короткая предварительная итерация, чтобы установить контекст и видение, и определить деловые прецеденты
- одна итерация уточнения, в течение которой определяются требования и устанавливается архитектура
- несколько итераций конструирования, в течение которых реализуются прецеденты и архитектура излагается в деталях
- несколько итераций перехода для передачи программы сообществу пользователей.

Эта стратегия пригодна, если:

- прикладная область известна
- риски хорошо понятны
- проектная группа надежна.

Эволюционный жизненный цикл

Эволюционная стратегия отличается от пошаговой тем, что потребности пользователя не поняты полностью, и все требования не могут быть определены с первого взгляда, они уточняются в каждой следующей конструкции.

Характерны следующие итерации:

- короткая предварительная итерация, чтобы установить контекст и видение, и определить деловые прецеденты
- несколько итераций уточнения, в течение которых уточняются требования
- одна итерация конструирования, в течение которой реализуются прецеденты и архитектура излагается в деталях
- несколько итераций перехода для передачи программы сообществу пользователей.

Эта стратегия пригодна, если:

- прикладная область новая или незнакомая
- группа неопытна.

Жизненный цикл пошаговой поставки

Некоторые авторы могут поэтапно поставлять заказчику постепенно возрастающие функциональные возможности. Эта стратегия может быть необходима, когда поджимают сроки поставки продукта на рынок, или когда может быть выгодной ранняя готовность некоторых главных возможностей. При таком подходе стадия перехода начинается раньше и имеет больше итераций. Эта стратегия требует очень устойчивой архитектуры.

Характерны следующие итерации:

- короткая предварительная итерация, чтобы установить контекст и видение, и определить деловые прецеденты
- одна итерация уточнения, в течение которой компонуется устойчивая архитектура
- одна итерация конструирования, в течение которой реализуются прецеденты и архитектура излагается в деталях
- несколько итераций перехода для передачи программы сообществу пользователей.

Эта стратегия пригодна, если:

- прикладная область известна, а поэтому архитектура и требования могут быть рано стабилизированы
- проектная группа надежна
- постепенные выпуски с увеличивающимися функциональными возможностями важны для заказчика.

Жизненный цикл "главного проекта"

Традиционный подход водопада можно представить как вырожденный случай, в котором имеется только одна длинная итерация стадии конструирования. На практике трудно избежать дополнительных итераций в стадии перехода.

Характерны следующие итерации:

- короткая предварительная итерация, чтобы установить контекст и видение, и определить деловые прецеденты
- одна очень длинная итерация конструирования, в течение которой реализуются прецеденты и архитектура излагается в деталях
- несколько итераций перехода для передачи программы сообществу пользователей.

Эта стратегия пригодна, если:

- маленькие приращения хорошо определенной функциональности добавляются к очень устойчивой программе
- новые функциональные возможности хорошо определены и поняты
- группа имеет опыт работы в прикладной области и с существующей программой.

На практике очень немного проектов строго следуют одной из описанных выше стратегий. Обычно применяют какую-либо их комбинацию, наиболее пригодную для конкретного случая.

6. ЗАДАЧИ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

В каждом из предложенных вариантов требуется при помощи языка UML построить модель программного обеспечения. Должны быть выполнены следующие действия:
1) составление глоссария проекта;
2) создание модели вариантов использования;
3) анализ вариантов использования;
4) проектирование системы;

После выполнения третьего этапа модель должна удовлетворять перечисленным ниже требованиям. Глоссарий проекта должен иметь вид таблицы и храниться в отдельном файле. На диаграммах вариантов использования каждое действующее лицо (actor) и вариант использования должны сопровождаться описанием. Эти описания должны быть составлены на русском языке. Описание действующего лица должно коротко (в одну-две строки) сообщать о роли данного лица. Описание варианта использования должно включать в себя пояснение, предусловие, потоки событий (основной и альтернативные, если таковые есть) и постусловие. Диаграммы взаимодействия, соответствующие потокам событий вариантов использования, должны содержать необходимые пояснения.

При проектировании системы требуется:

- создать иерархию классов системы;
- разместить классы по пакетам (использовать деление: пользовательский интерфейс – управление – данные; или другое в зависимости от постановки задачи);
- связать объекты с классами, сообщения на диаграммах взаимодействия – с операциями;
- каждый класс снабдить описанием, которое должно включать в себя краткое описание (ответственность класса), описание атрибутов в виде таблицы (имя, описание, тип), таблицу с описанием операций (имя, описание, сигнатура);
- для классов указать стереотипы;
- построить диаграммы классов системы, отображающие связи между классами;
- для описания поведения экземпляров отдельных классов построить диаграммы состояний;
- разработать (если это требуется вариантом задания) схему базы данных и отобразить ее на диаграмме «сущность – связь».

При реализации системы необходимо построить диаграммы компонентов для каждого пакета и для системы в целом. Также следует разработать диаграмму размещения. В зависимости от варианта задания диаграмма размещения должна показывать расположение компонентов в распределенном приложении или связи между встроенным процессором и устройствами. Должна быть произведена проверка корректности модели и автоматическая генерация кода средствами Rational Rose.

Задача 1. Цифровой диктофон

Требуется разработать модель программного обеспечения, управляющего работой цифрового диктофона. Цифровой диктофон – это бытовое электронное устройство, предназначенное для записи и воспроизведения речи. Звуковые сообщения записываются через встроенный микрофон и сохраняются в памяти устройства. Сообщения воспроизводятся через встроенный громкоговоритель. Работа устройства осуществляется под управлением центрального процессора. Диктофон хранит до 10 звуковых сообщений. Длина каждого сообщения ограничена размером свободной памяти. Диктофон осуществляет прямой (по номеру сообщения) доступ к любому сообщению из памяти. Пользователь имеет возможность воспроизводить сообщения, хранящиеся в памяти диктофона, стирать их, записывать новые. Исполнителем должна быть разработана схема базы данных для хранения сообщений в памяти диктофона. Интерфейс с пользователем осуществляется при помощи экранного меню и управляющих кнопок на корпусе диктофона. При помощи кнопок- стрелок осуществляется навигация по пунктам меню. Кнопки «Да», «Нет» служат для подтверждения или отмены пользователем выбора той или иной опции меню (структуру меню исполнитель должен разработать самостоятельно). Имеются также кнопки «Воспроизведение», «Пауза» и «Запись» для работы со звуковыми сообщениями. Во время записи сообщения на экране отображается время, в течение которого ведется запись, при воспроизведении – длительность воспроизведенной части сообщения. Если диктофон не используется, через 30 секунд он автоматически переходит в режим сбережения энергии. В этом режиме никакие операции над звуковыми сообщениями не возможны. Энергия расходуется только на сохранение памяти диктофона в неизменном состоянии. Переход из режима сбережения энергии в обычный режим осуществляется при нажатии пользователем любой кнопки. В диктофоне имеется датчик уровня заряда батарей. При падении уровня заряда ниже установленного предела диктофон автоматически переходит в режим сбережения энергии (независимо от того используется он в данный момент или нет). Переход в обычный режим становится возможным только после восстановления нормального уровня заряда батарей.

Задача 2. Торговый автомат

Требуется разработать модель программного обеспечения встроенного процессора универсального торгового автомата. В автомате имеется пять лотков для хранения и выдачи товаров. Загрузка товаров на лотки осуществляется обслуживающим персоналом. Автомат следит за наличием товара. Если какой-либо товар распродан, автомат отправляет сообщение об этом на станцию обслуживания и информирует покупателей (зажигается красная лампочка рядом с лотком данного товара). Автомат принимает к оплате бумажные купюры и монеты. Специальный индикатор высвечивает текущую сумму денег, принятых автоматом к оплате. После ввода денег клиент нажимает на кнопку выдачи товара. Выдача товара производится только в том случае, если введенная сумма денег соответствует цене товара. Товар выдается поштучно. При нажатии на кнопку «Возврат» клиенту возвращаются все принятые от него к оплате деньги. Возврат денег не производится после выдачи товара. Автомат должен корректно работать при одновременном нажатии на кнопки выдачи товара и возврата денег. В специальном отделении автомата, закрываемом замком, есть «секретная кнопка», которая используется обслуживающим персоналом для выемки выручки. При нажатии на эту кнопку открывается доступ к ящику с деньгами. Автомат получает со станции обслуживания данные о товарах и хранит их в своей памяти. Данные включают в себя цену, наименование товара, номер лотка, на котором находится товар и количество товара на лотке. Вариант задания включает в себя разработку схемы базы данных о товарах.

Задача 3. Табло на станции метро

Требуется разработать модель программного обеспечения табло для информационной службы метрополитена. Табло расположены на каждой станции метро. Они работают под управлением единого пункта управления (ПУ) информационной службы метро. Табло отображает текущее время (часы, минуты, секунды) и время, прошедшее с момента отправления последнего поезда (минуты, секунды). Момент прибытия и отправления поезда определяется при помощи датчиков, устанавливаемых на путях. Все табло метро синхронизованы, текущее время отсчитывается и устанавливается из центральной службы времени, находящейся на ПУ. На табло высвечивается конечная станция назначения прибывающего поезда. Эти данные содержатся в расписании движения поездов, которое хранится в памяти табло и периодически обновляется с ПУ. В «бегущей строке» табло отображается рекламная информация. Память табло хранит до 10 рекламных сообщений. Сообщения отображаются друг за другом с небольшими паузами, циклически. Содержание рекламных сообщений поступает с ПУ.

Дополнительная функция табло – по запросу с ПУ оно пересылает данные о нарушениях расписания (преждевременных отправлениях поездов или опозданиях). В ходе выполнения задания должна быть создана схема базы данных для хранения рекламных сообщений, расписания и сведений о нарушении расписаний.

Пояснение: в задании требуется разработать модель ПО только для табло, но не для пункта управления информационной службы.

Задача 4. Система автоматизации для пункта проката видеокассет

Требуется разработать модель программной системы автоматизации работы пункта проката видеокассет (далее в тексте – системы). Пункт проката содержит каталог кассет, имеющихся в наличии в данный момент времени. Система поддерживает работу каталога, позволяя служащим проката добавлять новые наименования кассет, удалять старые и редактировать данные о кассетах. Клиент, обратившийся в пункт, выбирает кассету по каталогу, вносит залог и забирает ее на определенный срок. Срок проката, измеряемый в сутках, оговаривается при выдаче кассеты. Стоимость проката
вычисляется системой исходя из тарифа за сутки и срока проката. Клиент возвращает кассету и оплачивает прокат. Если кассета не повреждена, клиенту возвращается залог. Служащий пункта проката регистрирует сдачу кассеты клиенту и ее возврат в системе. Если клиент повредил кассету, то кассета удаляется из каталога, а залог остается в кассе проката. При необходимости служащий может запросить у системы следующие данные:

- имеется ли в наличии кассета с данным названием;
- когда будет возвращена какая-либо кассета из тех, что сданы в прокат;
- является ли данный клиент постоянным клиентом пункта проката (пользовался ли прокатом 5 или более раз).

Постоянным клиентам предоставляются скидки, а также от них принимаются заявки на пополнение ассортимента кассет. Заявки регистрируются в системе. По ним готовится итоговый отчет, руководствуясь которым, служащие пункта проката обновляют ассортимент кассет. Необходимо разработать схему базы данных для хранения каталога, учетных записей о прокате кассет и заявок на пополнение ассортимента.

Задача 5. Мини-АТС

Требуется разработать модель программного обеспечения встроенного микропроцессора учрежденческой мини-АТС (автоматической телефонной станции). Мини-АТС осуществляет связь между служащими учреждения. Каждый абонент подключен к ней линией связи. Мини-АТС со-

единяет линии абонентов (осуществляет коммутацию линий). Абоненты имеют номера, состоящие из трех цифр. Специальный номер 9 зарезервирован для внешней связи. Телефонное соединение абонентов производится следующим образом. Абонент поднимает трубку телефона, и мини-АТС получает сигнал «Трубка». В ответ мини-АТС посылает сигнал «Тон». Приняв этот сигнал, абонент набирает телефонный номер (посылает три сигнала «Цифра»). Мини-АТС проверяет готовность вызываемого абонента. Если абонент не готов (его линия занята), мини-АТС посылает вызывающему абоненту сигнал «Занято». Если абонент готов, мини-АТС посылает обоим абонентам сигнал «Вызов». При этом телефон вызываемого абонента начинает звонить, а вызывающий абонент слышит в трубке длинные гудки. Вызываемый абонент снимает трубку, и мини-АТС получает от него сигнал «Трубка», после чего осуществляет коммутацию линии. Абоненты обмениваются сигналами «Данные», которые мини-АТС должна передавать от одного абонента к другому. Когда один из абонентов опускает трубку, мини-АТС получает сигнал «Конец» и посылает другому абоненту сигнал «Тон». В любой момент абонент может положить трубку, при этом мини-АТС получает сигнал «Конец». После получения этого сигнала сеанс обслуживания абонента завершается. Если абонент желает соединиться с абонентом за пределами учреждения, то он набирает номер «9». Мини-АТС посылает по линии, соединяющей с внешней (городской) АТС, сигнал «Трубка» и в дальнейшем служит посредником между телефоном абонента и внешней АТС. Она принимает и передает сигналы и данные между ними, не внося никаких изменений. Единственное исключение касается завершения сеанса. Получив от городской АТС сигнал «Конец», мини-АТС посылает абоненту сигнал «Тон», и ждет сигнала «Конец» для завершения обслуживания абонента. Если же вызывавший абонент первым вешает трубку, то мини-АТС получает сигнал «Конец», передает его городской АТС и завершает сеанс. Мини-АТС может получить сигнал «Вызов» от городской АТС. Это происходит, когда нет соединений с внешними абонентами. Сигнал «Вызов» от городской АТС передается абоненту с кодом «000». Только этот абонент может отвечать на внешние звонки.

Задача 6. Телефон

Требуется разработать модель программного обеспечения встроенного микропроцессора для аппарата учрежденческой телефонной сети. Аппарат подключен к линии связи, ведущей к мини-АТС. В его задачу входит прием и передача сигналов (в том числе и голосовых данных) мини-АТС. Аппарат имеет кнопочную панель управления, экран для отображения набираемых номеров, звонок и трубку, в которую встроены микрофон и громкоговоритель. В начальном состоянии трубка телефона повешена, те-

лефон не реагирует на нажатия кнопок. Телефон реагирует только на сигнал «Вызов» от мини-АТС, при этом включается звонок. При снятии трубки на АТС подается сигнал «Трубка». При получении ответного сигнала «Тон» от АТС телефон воспроизводит звуковой тон «Готов» (длинный непрекращающийся гудок) в трубку. При получении сигнала «Занято», в трубке воспроизводится тон «Занято» (частые короткие гудки). Пользователь, слыша в трубке тон «Готов», набирает трехзначный номер. Номер может быть набран при помощи кнопок с цифрами или нажатием на специальную кнопку « # ». При нажатии на кнопку с цифрой соответствующий ей сигнал «Цифра» передается АТС. Нажатия на кнопки с цифрами после третьего игнорируются. Во время набора номера введенные цифры отображаются на экране. Последний полностью набранный номер запоминается в памяти аппарата для того, чтобы можно было его воспроизвести при нажатии на кнопку « # ». При нажатии на эту кнопку номер из памяти аппарата высвечивается на экране, и АТС передается последовательность из трех сигналов «Цифра». В ответ на набранный номер от АТС приходит либо сигнал «Занято», либо сигнал «Вызов». При получении сигнала «Вызов» телефон воспроизводит в трубку длинные гудки до того момента, когда АТС осуществит коммутацию и передаст сигнал «Данные». Телефон воспроизводит данные, передаваемые с сигналом, в трубку. Ответ пользователя воспринимается микрофоном трубки, преобразуется в сигнал «Данные» и передается АТС. Обмен данными прерывается, если повешена трубка одного из телефонов, участвующих в обмене. О том, что трубку повесил вызываемый абонент, сообщает сигнал «Занято», посылаемый АТС. После того, как трубка аппарата была повешена, телефон посылает АТС сигнал «Конец», и телефон переходит в начальное состояние.

Задание 7. Стиральная машина

Требуется разработать модель программного обеспечения встроенного микропроцессора стиральной машины. Машина предназначена для автоматической стирки белья. Машина включает в себя следующие устройства: бак для белья, клапаны для забора и слива воды, мотор, устройство подогрева воды, термометр, таймер, дверца для доступа в бак, несколько емкостей для различных моющих средств, панель управления с кнопками и индикатором. В памяти машины хранятся 5 программ стирки, заданные изготовителем. Пользователи не могут вносить в них изменения. Каждая программа определяет температуру воды, длительность стирки, используемые моющие средства (номер емкости и время подачи), скорость вращения бака во время стирки и отжима. Вариант задания предусматривает разработку схемы базы данных для хранения программ стирки в памяти машины. Для использования машины необходимо открыть дверцу, поместить белье в бак, поместить моющие средства в емкости, закрыть дверцу,

выбрать программу стирки и нажать на кнопку «Пуск». Перед тем как приступить к стирке машина открывает клапан для забора воды, набирает необходимое количество воды, после чего закрывает клапан. Далее, машина действует по выбранной пользователем программе:

1) Подогревает, если необходимо, воду до нужной температуры.
2) Включает таймер и запускает вращение бака для стирки.
3) По таймеру подает в бак моющие средства, предусмотренные программой.
4) По окончании стирки сливает воду и запускает отжим.

Во время работы машины на индикаторе высвечивается время, прошедшее с момента запуска (минуты и секунды), текущий режим работы (стирка или отжим), номер текущей программы стирки. В целях безопасности дверца бака блокируется до окончания стирки. Машина не воспринимает нажатий на кнопки, за исключением одной – пользователь имеет возможность в любой момент нажать на кнопку «Останов», чтобы принудительно остановить стирку и слить воду.

Задание 8. Таксофон

Требуется разработать модель встроенной системы управления работой таксофона городской телефонной сети. Таксофон предназначен для оказания платных услуг телефонной связи. Он подключен к линии связи. В нем имеется кнопочная панель, дисплей, трубка со встроенным микрофоном и громкоговорителем, приемник карт – устройство для считывания телефонных карт, используемых для оплаты разговора. В начальном состоянии трубка таксофона повешена, дисплей потушен, таксофон не реагирует на нажатия кнопок и какие-либо сигналы из линии. При снятии трубки таксофон выдает на дисплей сообщение «Вставьте карту» и ожидает, когда пользователь вставит карту в приемник. Дальнейшее функционирование таксофона осуществляется только при вставленной карте. Если карту вынимают, таксофон возвращается к началу и выдает сообщение о необходимости вставить карту. При попадании карты в приемник производится считывание информации с карты. Если кредит исчерпан или карта не пригодна (не удается узнать кредит), то таксофон выдает соответствующее сообщение на дисплей таксофона. Если карта может быть использована для оплаты, то на дисплей выдается количество «единиц» на карте, и на телефонную станцию (АТС) подается сигнал «Трубка». При получении ответного сигнала «Тон» из линии таксофон воспроизводит звуковой тон «Готов» (длинный непрекращающийся гудок) в трубку. При получении сигнала «Занято», в трубке воспроизводится тон «Занято» (короткие гудки). После получения от АТС сигнала «Тон» от пользователя принимаются семизначный номер вызываемого абонента, остальные нажатия на кнопки игнорируются. Когда пользователь нажимает на кнопку с цифрой соответст-

вующий ей сигнал «Цифра» передается АТС. Во время набора номера введенные цифры отображаются на дисплее. В ответ на набранный номер от АТС приходит либо сигнал «Занято», либо сигнал «Вызов». При получении сигнала «Вызов» таксофон воспроизводит в трубку длинные гудки до того момента, когда АТС осуществит коммутацию и передаст сигнал «Данные». Таксофон воспроизводит данные, передаваемые с сигналом, в трубку. При получении данных из трубки, аппарат преобразует их в сигнал «Данные» и передает их АТС. Во время разговора на дисплее ведется отсчет времени и уменьшается кредит на телефонной карте – каждые 15 секунд вычитается четверть «единицы». Обмен данными прерывается, в следующих случаях:

- исчерпан кредит;
- карта вынута из приемника;
- от АТС пришел сигнал «Занято»;
- повешена трубка таксофона.

Если трубка была повешена, аппарат посылает в линию сигнал «Конец» и выдает на дисплей сообщение «Выньте карту». После извлечения карты из приемника таксофон переходит в начальное состояние.

Задание 9. Банкомат

Требуется разработать модель программного обеспечения банкомата. Банкомат – это автомат для выдачи наличных денег по кредитным пластиковым карточкам. В его состав входят следующие устройства: дисплей, панель управления с кнопками, приемник кредитных карт, хранилище денег и лоток для их выдачи, хранилище конфискованных кредитных карт, принтер для печати справок. Банкомат подключен к линии связи для обмена данных с банковским компьютером, хранящим сведения о счетах клиентов. Обслуживание клиента начинается с момента помещения пластиковой карточки в банкомат. После распознавания типа пластиковой карточки, банкомат выдает на дисплей приглашение ввести персональный код. Персональный код представляет собой четырехзначное число. Затем банкомат проверяет правильность введенного кода. Если код указан неверно, пользователю предоставляются еще две попытки для ввода правильного кода. В случае повторных неудач карта перемещается в хранилище карт, и сеанс обслуживания заканчивается. После ввода правильного кода банкомат предлагает пользователю выбрать операцию. Клиент может либо снять наличные со счета, либо узнать остаток на его счету. При снятии наличных со счета банкомат предлагает указать сумму (10, 50, 100, 200, 500, 1000 рублей). После выбора клиентом суммы банкомат запрашивает, нужно ли печатать справку по операции. Затем банкомат посылает запрос на снятие выбранной суммы центральному компьютеру банка. В случае получения разрешения на операцию, банкомат проверяет, имеется ли требуемая сум-

ма в его хранилище денег. Если он может выдать деньги, то на дисплей выводится сообщение «Выньте карту». После удаления карточки из приемника, банкомат выдает указанную сумму в лоток выдачи. Банкомат печатает справку по произведенной операции, если она была затребована клиентом. Если клиент хочет узнать остаток на счету, то банкомат посылает запрос центральному компьютеру банка и выводит сумму на дисплей. По требованию клиента печатается и выдается соответствующая справка. В специальном отделении банкомата, закрываемом замком, есть «секретная кнопка», которая используется обслуживающим персоналом для загрузки денег. При нажатии на эту кнопку открывается доступ к хранилищу денег и конфискованным кредитным картам.

Задание 10. Холодильник

Требуется разработать модель программного обеспечения встроенного процессора холодильника. Холодильник состоит из нескольких холодильных камер для хранения продуктов. В каждой холодильной камере имеется регулятор температуры, мотор, термометр, индикатор, таймер, датчик открытия двери камеры и устройство для подачи звуковых сигналов. При помощи терморегулятора устанавливается максимально допустимая температура в данной камере. Мотор предназначен для поддержания низкой температуры. Термометр постоянно измеряет температуру внутри камеры, а индикатор температуры, расположенный на дверце, постоянно высвечивает ее значение. При повышении температуры выше предела, определяемого текущим положением регулятора, включается мотор. При снижении температуры ниже некоторого другого значения, связанного с первым, мотор отключается. Доступ в камеру осуществляется через дверцу. Если дверь холодильной камеры открыта в течение слишком долгого времени, подается звуковой сигнал. Звуковой сигнал также подается в любых нештатных ситуациях (например, при поломке мотора). Холодильник ведет электронный журнал, в котором отмечаются все происходящие события:

- изменение положения терморегулятора камеры;
- включение и отключение мотора;
- доступ в камеру;
- внештатные ситуации.

Вариантом задания предусмотрена разработка схемы базы данных для хранения журнала событий холодильника. Содержимое журнала может быть передано в компьютер, подсоединенный к специальному гнезду на корпусе холодильника.

Задание 11. Кодовый замок

Требуется разработать модель программного обеспечения встроенного микропроцессора для кодового замка, регулирующего доступ в помещение. Кодовый замок состоит из панели с кнопками (цифры «0»...«9», кнопка «Вызов», кнопка «Контроль»), цифрового дисплея, электромеханического замка, звонка. Панель с кнопками устанавливается с наружной стороны двери, замок устанавливается с внутренней стороны двери, звонок устанавливается внутри охраняемого помещения. В обычном состоянии замок закрыт. Доступ в помещение осуществляется после набора кода доступа, состоящего из четырех цифр. Во время набора кода введенные цифры отображаются на дисплея. Если код набран правильно, то замок открывается на некоторое время, после чего дверь снова закрывается. Содержимое дисплея очищается. Кнопка «Вызов» используется для подачи звукового сигнала внутри помещения. Кнопка «Контроль» используется для смены кодов. Смена кода доступа осуществляется следующим образом. При открытой двери нужно набрать код контроля, состоящий из четырех цифр, и новый код доступа. Для смены кода контроля нужно при открытой двери и нажатой кнопке «Вызов» набрать код контроля, после чего – новый код контроля.

Задание 12. Турникет метро

Требуется разработать модель программного обеспечения встроенного процессора турникета для метрополитена. При помощи турникета контролируется проход пассажиров в метро и взимается входная плата. Турникет имеет приемник карт, устройство для перекрывания доступа, таймер, три оптических датчика для определения прохода пассажира, устройство подачи звуковых сигналов, индикаторы «Проход» и «Стоп». В начальном состоянии турникета зажжен индикатор «Стоп», индикатор «Проход» потушен. Если один из датчиков посылает сигнал, то проход через турникет сразу же перекрывается, и подается предупредительный звуковой сигнал. Для прохода пассажир должен поместить карту в приемник карт. Турникет считывает с нее данные: срок годности карты и количество «единиц» на ней. Если данные не удается считать, или карта просрочена, или заблокирована, то карта возвращается пассажиру, и турникет остается в исходном состоянии. В другом случае с карты списывается одна «единица», карта возвращается из приемника, индикатор «Стоп» гаснет, зажигается индикатор «Проход», и пассажир может пройти через турникет. Получив от одного из датчиков сигнал, турникет ожидает время, отведенное на проход пассажира (5 секунд), после чего он возвращается в начальное состояние. Наличие трех датчиков в турникете гарантирует, что при проходе пассажира хотя бы один из них подаст сигнал (датчики невозможно перешагнуть, пе-

репрыгнуть и т.д.). Во время прохода пассажира возможна ситуация, когда все три датчика посылают сигналы. В этом случае принимается только первый сигнал и от момента его приема отсчитывается положенное время. Остальные сигналы игнорируются. Турникет заносит в свою память время всех оплаченных проходов. В конце рабочего дня он передает всю информацию, накопленную за день, в АСУ метрополитена. В ходе выполнения этого варианта задания должна быть разработана схема базы данных о проходах через турникет.

Задание 13. Система учета товаров

Требуется разработать модель системы поддержки заказа и учета товаров в бакалейной лавке. В бакалейной лавке для каждого товара фиксируется место хранения (определенная полка), количество товара и его поставщик. Система поддержки заказа и учета товаров должна обеспечивать добавление информации о новом товаре, изменение или удаление информации об имеющемся товаре, хранение (добавление, изменение и удаление) информации о поставщиках, включающей в себя название фирмы, ее адрес и телефон. При помощи системы составляются заказы поставщикам. Каждый заказ может содержать несколько позиций, в каждой позиции указываются наименование товара и его количество в заказе. Система учета по требованию пользователя формирует и выдает на печать следующую справочную информацию:

- список всех товаров;
- список товаров, имеющихся в наличии;
- список товаров, количество которых необходимо пополнить;
- список товаров, поставляемых данным поставщиком.

В ходе выполнения этого варианта задания должна быть разработана схема базы данных, хранящей информацию о товарах, заказах и поставщиках.

Задание 14. Библиотечная система

Требуется разработать модель системы автоматизирующей деятельность библиотеки. Система поддержки управления библиотекой должна обеспечивать операции (добавление, удаление и изменение) над данными о читателях. В регистрационном списке читателей хранятся следующие сведения: фамилия, имя и отчество читателя; номер его читательского билета и дата выдачи билета. Наряду с регистрационным списком системой должен поддерживаться каталог библиотеки, где хранится информация о книгах: название, список авторов, библиотечный шифр, год и место издания, название издательства, общее количество экземпляров книги в библиотеке и количество экземпляров, доступных в текущий момент. Система обеспечивает добавление, удаление и изменение данных

чивает добавление, удаление и изменение данных каталога, а также поиск книг в каталоге на основании введенного шифра или названия книги. В системе осуществляется регистрация взятых и возвращенных читателем книг. Про каждую выданную книгу хранится запись о том, кому и когда была выдана книга, и когда она будет возвращена. При возврате книги в записи делается соответствующая помстка, а сама запись не удаляется из системы. Система должна выдавать следующую справочную информацию:

- какие книги были выданы за данный промежуток времени;
- какие книги были возвращены за данный промежуток времени;
- какие книги находятся у данного читателя;
- имеется ли в наличии некоторая книга.

Вариант задания предусматривает разработку схемы базы данных, хранящей список читателей, каталог книг и записи о выдаче книг.

Задание 15. Интернет-магазин

Требуется разработать модель программного обеспечения Интернет-магазина. Интернет-магазин позволяет делать покупки с доставкой на дом. Клиенты магазина при помощи программы-браузера имеют доступ к каталогу продаваемых товаров, поддержку которого осуществляет Интернет-магазин. В каталоге товары распределены по разделам. О каждом товаре доступна полная информация (название, вес, цена, изображение, дата изготовления и срок годности) Для удобства клиентов предусмотрена система поиска товаров в каталоге. Заполнение каталога информацией происходит автоматически в начале рабочего дня, информация берется из системы автоматизации торговли. При отборе клиентами товаров поддерживается виртуальная «торговая корзина». Любое наименование товара может быть добавлено в «корзину» или изъято в любой момент по желанию покупателя с последующим пересчетом общей стоимости покупки. Текущее содержимое «корзины» постоянно показывается клиенту. По окончании выбора товаров производится оформление заказа и регистрация покупателя. Клиент указывает в регистрационной форме свою фамилию, имя и отчество, адрес доставки заказа и телефон, по которому с ним можно связаться для подтверждения сделанного заказа. Заказы передаются для обработки в систему автоматизации торговли. Проверка наличия товаров на складе и их резервирование Интернет-магазином не производятся. Дополнительно требуется разработать схему базы данных, хранящей заказы. Следует определиться, по какому архитектурному шаблону будет строиться Web-приложение («тонкий клиент» или «толстый клиент»). В соответствии с выбранным шаблоном следует построить модели клиентской части магазина и серверной части, промоделировать связи между частями приложения. Для Web-приложений типичными являются следующие *классы*:

- клиентская Web-страница;

- серверная Web-страница (например, CGI-скрипт);
- HTML-форма;
- объект JavaScript.

Дополнительные *связи* между классами Web-приложений:

- link – ссылка с одной страницы на другую;
- build – связь между CGI-скриптом и клиентской страницей, генерируемой при его выполнении;
- submit – связь между формой и серверной Web-страницей,принимающей данные из формы.

Типичные *компоненты*:

- Web-страница (HTML-файл),
- Active Server Page (ASP),
- Java Server Page (JSP),
- сервлет,
- библиотека скриптов (например, подключаемый файл с Javascript-функциями).

Задание 16. WWW-конференция

Требуется разработать модель программного обеспечения WWW-конференции. WWW-конференция представляет собой хранилище сообщений в сети Интернет, доступ к которому осуществляется при помощи браузера. Для каждого сообщения конференции хранятся значения следующих полей: номер сообщения, автор, тема, текст сообщения, дата добавления сообщения, ссылка на родительское сообщение. Начальной страницей конференции является иерархический список сообщений. Верхний уровень иерархии составляют сообщения, открывающие новые темы, а подуровни составляют сообщения, полученные в ответ на сообщения верхнего уровня. Сообщение-ответ всегда имеет ссылку на исходное сообщение. В списке отображаются только темы сообщений, их авторы и даты добавления. Просматривая список, пользователь выбирает сообщение и по гиперссылке открывает страницу с текстом сообщения. Помимо текста на этой странице отображается список (иерархический) сообщений являющихся ответами, ответами на ответы и т.д. Для удобства пользователей необходимо предусмотреть поиск сообщений по автору или по ключевым словам в теме или тексте сообщения. Сообщения добавляются в конференцию зарегистрированными пользователями, которые при отправке сообщения должны указать своё имя и пароль. Регистрирует новых пользователей модератор конференции – её ведущий. При регистрации пользователь заполняет специальную форму, содержимое которой затем пересылается модератору и запоминается в базе пользователей. Модератор решает, реги-

стрировать пользователя или нет, и отправляет свой ответ. При добавлении сообщений пользователь имеет возможность начать новую тему или ответить на ранее добавленные сообщения. После добавления сообщения оно доступно для чтения всем пользователям (даже незарегистрированным), и список сообщений обновляется. Модератор имеет право по тем или иным причинам удалять сообщения любых авторов. Он также может наказывать пользователей, нарушающих правила поведения в конференции, лишая на некоторое время пользователя возможности добавлять и редактировать сообщения. Вариант задания включает в себя разработку схемы базы данных для хранения сообщений конференции и информации об её участниках. Выполняющим это задание полезно ознакомиться с заключительным замечанием к варианту «Интернет-магазин». Наиболее подходящей архитектурой для WWW-конференции является «тонкий клиент», поскольку клиентская часть практически не содержит «бизнес-логики». Единственным её элементом, который может выполняться на стороне клиента, является проверка правильного заполнения полей формы, перед отправкой её содержимого на сервер.

Задание 17. Каталог ресурсов Интернет

Требуется разработать модель программного обеспечения каталога ресурсов сети Интернет. В каталоге хранится следующая информация о ресурсах: название ресурса, уникальный локатор ресурса (URL), раздел каталога, в котором содержится ресурс, список ключевых слов, краткое описание, дата последнего обновления, контактная информация. Доступ пользователей к каталогу осуществляется при помощи браузера. Пользователи каталога могут добавлять новые ресурсы, информация о которых не была внесена ранее. Ресурсы в каталоге классифицируются по разделам. Полный список ресурсов каждого раздела должен быть доступен пользователям. Пользователям каталога должны быть предоставлены возможности по поиску ресурсов. Поиск осуществляется по ключевым словам. Если пользователь не доволен результатами поиска, он может уточнить запрос (осуществить поиск среди результатов предыдущего поиска). Должна быть возможность выдавать результаты поиска в разной форме (вывод всей информации о ресурсах или частичной). Пользователь может отсортировать список ресурсов по релевантности (соответствию ключевым словам из запроса) или по дате обновления. Поскольку содержание ресурсов Интернет со временем изменяется необходимо следить за датой последнего обновления, периодически опрашивая Web-сайты, URL которых хранятся в каталоге. Вариант задания включает в себя разработку схемы базы данных для хранения сообщений конференции и информации об её участниках. Выполняющим это задание полезно ознакомиться с заключительным замечанием к варианту «Интернет-магазин». Как и в варианте «WWW-

конференция» самой подходящей архитектурой для каталога является «тонкий клиент», поскольку клиентская часть практически не включает в себя функций «бизнес-логики» кроме проверки содержимого форм перед пересылкой на сервер.

Задание 18. Будильник

Требуется разработать модель программного обеспечения встроенного микропроцессора для будильника. На экране будильника постоянно отображается текущее время (часы и минуты, например: *12 : 00*), двоеточие между числом часов и числом минут зажигается и гаснет с интервалом в полсекунды. Управление будильником осуществляется следующими кнопками:

- кнопкой режима установки времени,
- кнопкой режима установки времени срабатывания,
- двумя отдельными кнопками для установки часов и минут,
- кнопкой сброса сигнала «СБРОС».

На будильнике имеется переключатель режима работы со следующими положениями: «ВЫКЛ», «ВКЛ», «РАДИО» и «ТАЙМЕР». Для установки текущего времени нужно нажать на кнопку режима установки и, при нажатой кнопке, нажимать на кнопки установки часов и минут. При каждом нажатии на кнопки, устанавливаемое значение увеличивается на одну единицу (один час или одну минуту соответственно). При достижении максимального значения производится сброс. Для установки времени срабатывания будильника нужно нажать на кнопку режима установки времени срабатывания и, держа кнопку нажатой, нажимать на кнопки установки часов и минут. Когда переключатель режима работы находится в положении «ВКЛ», при достижении времени срабатывания происходит подача звукового сигнала в течение одной минуты. Сигнал можно прервать, нажав на кнопку «СБРОС». При этом сигнал должен быть возобновлен через пять минут. При установке переключателя в положение «ВЫКЛ» звуковой сигнал не подается. Когда переключатель находится в положении «РАДИО» работает радиоприемник. При переводе переключателя в положение «ТАЙМЕР» включается радиоприемник на тридцать минут, а затем часы переходят в состояние будильника (аналогично положению «ВКЛ»). При нажатии на кнопку режима установки времени, будильник должен отображать время срабатывания.

Задание 19. Генеалогическое дерево

Требуется разработать модель системы для поддержки генеалогических деревьев. Система хранит сведения о персонах (Ф.И.О., пол, дата рождения, дата смерти, биография) и о родственных связях между ними.

Связи бывают только трех видов: «мужья-жены», «дети-родители» и «братья- сестры». Система обеспечивает возможность добавления данных о новых персонах и родственных связях, изменение введенных данных и удаление ненужных данных. Система следит за непротиворечивостью вводимых данных. Например, недопустимо, чтобы человек был собственным предком или потомком. Разработанная модель должна содержать схему базы данных для хранения генеалогических деревьев. Пользователи системы могут осуществлять поиск полезной информации по дереву:

- находить для указанного члена семьи его детей;
- находить для указанного члена семьи его родителей;
- находить для указанной персоны братьев и сестер, если таковые есть;
- получать список всех предков персоны;
- получать список всех потомков персоны;
- получать список всех родственников персоны;
- прослеживать цепочку родственных связей от одной персоны до другой (например, если Петр является шурином Ивана, то на запрос о родственных связях между Петром и Иваном выдается такой результат: «Петр – брат Ольги, Ольга – жена Ивана»).

Задание 20. Телевизор

Требуется разработать модель встроенной системы управления работой телевизора. В телевизоре имеются следующие устройства: приемник телевизионного сигнала, устройство отображения картинки, память каналов, память настроек, управляющие кнопки, пульт дистанционного управления (ДУ). Управление телевизором осуществляется при помощи кнопок на корпусе (их четыре: «ВКЛ / ВЫКЛ», « – », « + », кнопка начальной установки) и пульта ДУ. Кнопка «ВКЛ / ВЫКЛ» позволяет включать и выключать телевизор. После включения телевизора на экран отображается передача, идущая по каналу №1, при этом используются параметры изображения и значение громкости, сохраненные в памяти настроек. Память каналов телевизора хранит до 60 каналов. Каналы нумеруются, начиная с нуля. Последовательное переключение каналов осуществляется при помощи кнопок «–» и «+». Нажатие на «+» переключает телевизор на канал с номером, на единицу большим (с 59-го канала телевизор переключается на 0-ой). Нажатие на «–» переключает телевизор на канал с номером, на единицу меньшим (с 0-го канала телевизор переключается на 59-ый). При нажатии на кнопку начальной установки очищается память каналов телевизора, после чего осуществляется поиск передач и сохранение их частот в памяти каналов. Поиск начинается с нижней границы рабочего диапазона телевизора. На экран телевизора выводится «синий экран». Рабочая частота постепенно увеличивается до тех пор, пока приемник не обнаружит те-

левизионный сигнал. Найденная передача выводится на экран в течение 10 секунд. Также отображается номер, под которым найденный канал будет сохранен в памяти (начиная с 1). Затем поиск продолжается до тех пор, пока не достигнута верхняя граница диапазона, или пока не заполнена вся память каналов. Телевизор принимает управляющие сигналы с пульта ДУ. На пульте ДУ расположены следующие кнопки:

- кнопки с цифрами «0»...«9» для прямого переключения канала (по номеру);
- кнопки «П–» и «П+» для последовательного переключения каналов;
- кнопки «Г–» и «Г+» для изменения громкости;
- кнопки «МЕНЮ», « < » и « > » для доступа к экранному меню.

Для прямого переключения на нужный канал его номер набирается с помощью кнопок с цифрами. После нажатия первой цифры в течение 5 секунд ожидается нажатие второй. Если вторая цифра не была нажата, то номер канала считается состоящим из одной цифры и осуществляется переключение на него. Кнопки «П–» и «П+» на пульте имеют те же функции, что и кнопки «-» и «+» на корпусе телевизора. Кнопки «Г–» и «Г+» позволяют увеличивать или уменьшать громкость. Каждое нажатие на «Г–» уменьшает громкость на одну единицу, пока она больше нуля, «Г+» увеличивает громкость на единицу, пока не достигнуто максимальное значение. Текущее значение громкости сохраняется в памяти настроек. Кнопки «МЕНЮ», «<» и «>» позволяют устанавливать значения настоек, хранящихся в памяти телевизора. При нажатии на кнопку «МЕНЮ» внизу экрана возникает надпись «ЯРКОСТЬ» и полоса, отображающая текущее значение яркости. Кнопками «<» и «>» яркость можно уменьшить или увеличить. При работе с меню нажатия на все остальные кнопки игнорируются. После повторного нажатия на кнопку «МЕНЮ» значение яркости запоминается в памяти настроек, и осуществляется переход к настройке контрастности. Настройка контрастности и остальных параметров (четкости, цветовой гаммы) происходит аналогично. Нажатие на кнопку «МЕНЮ» по окончании настройки цветовой гаммы (последнего пункта меню) приводит к окончанию работы с меню. Выход из меню также осуществляется в том случае, если в течение 15 секунд не была нажата ни одна кнопка.

Задание 21. Система поддержки составления расписания занятий

Требуется разработать модель системы поддержки составления расписания занятий. Система обеспечивает составление расписания некоторого учебного заведения, внесение в расписание изменений, выдачу полного расписания и дополнительной информации (например, по итоговому расписанию составляется расписание указанной группы на заданный день или неделю). В расписании фиксируются время и место проведения занятия, предмет и преподаватель, проводящий занятие, а также номер группы, для

которой это занятие проводится. Расписание не должно содержать коллизий (например, разные занятия не должны пересекаться друг с другом по месту и времени их проведения, один преподаватель не может вести одновременно два разных занятия, в одно и то же время у одной и той же группы не может быть два различных занятия и т. д.). При работе над этим вариантом задания необходимо разработать схему базы данных для хранения расписания.

Задание 22. Домофон

Требуется разработать модель программного обеспечения встроенного микропроцессора домофона. Домофон регулирует доступ в подъезд многоквартирного дома. В подъезде имеется дверь с замком. С наружной стороны двери установлена внешняя панель домофона, на которой находятся кнопки для связи с каждой квартирой, микрофон и динамик. В каждой квартире находится внутренняя панель домофона с кнопками: «СВЯЗЬ», «БЛОКИРОВКА» и «ОТКРЫТЬ». Кроме того, на внутренней панели имеется микрофон и динамик. Жильцы могут открывать дверь ключом. Посетитель может нажать кнопку квартиры на внешней панели. При этом в квартире раздается звонок (если подача звонка в квартиру не заблокирована). Услышав звонок, жилец квартиры нажимает на кнопку «СВЯЗЬ» внутренней панели домофона, после чего домофон устанавливает звуковое сообщение между жильцом и посетителем. Звуки, произносимые посетителем в микрофон, установленный на внешней панели, воспроизводятся в динамике, установленном в квартире. Звуки из микрофона в квартире, передаются в динамик на внешней панели. После сеанса связи жилец может нажать на кнопку «ОТКРЫТЬ», чтобы замок на двери в подъезд открылся, и посетитель смог войти. По истечении минуты замок должен снова заблокировать вход в подъезд. Жилец, который желает, чтобы его не беспокоили, может отключить подачу звонка в свою квартиру, нажав на кнопку «БЛОКИРОВКА». Повторное нажатие на эту кнопку вновь включает подачу звонка.

ЗАКЛЮЧЕНИЕ

Потребность в сложных программных системах растет с ошеломляющей быстротой. Поэтому происходит постоянная модернизация принципов разработки программного обеспечения, описанных в настоящем пособии. Фундаментальная ценность объектно-ориентированного проектирования как технологии заключается в том, что оно позволяет превратить разработку в истинно творческий процесс на всем протяжении жизненного цикла продукта. Цель данного курса была попытка развить именно данную идею, поскольку только с использованием творческой мысли можно создавать качественные программные продукты, удовлетворяющие требованиям пользователей.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. 2-е изд. / Пер. с англ. – М.: Издательство Бином, СПб.: Невский диалект, 1999.
2. Буч Г., Рамбо Дж., Джекобсон А. Язык UML. Руководство пользователя / Пер. с англ. – М.: ДМК, 2000.
3. С.А. Трофимов. CASE-технологии: практическая работа в Rational Rose – М.: БИНОМ, 2001.
4. А. Якобсон, Г. Буч, Дж. Рамбо унифицированный процесс разработки программного обеспечения / Пер. с англ. – СПб.: Питер, 2002.

Крутов Алексей Николаевич

МЕТОДЫ ПРОГРАММИРОВАНИЯ. ООП. UML. RUP

Учебное пособие

Печатается в авторской редакции

Компьютерная верстка, макет Н.П.Бариновой

Лицензия ИД № 06178 от 01.11.01. Подписано в печать 01.09.04. Гарнитура «Times New Roman». Формат 60х84/16. Бумага офсетная. Печать оперативная. Объем 6,74 усл. печ. л., 7,25 уч. -изд. л. Тираж 150 экз. Заказ № 212.
Издательство «Самарский университет», 443011, г. Самара, ул. Ак. Павлова, д.1.
Отпечатано ООО «Универс-групп»