

*Министерство науки и высшего образования РФ  
Федеральное государственное бюджетное образовательное  
учреждение высшего образования  
Алтайский государственный технический университет  
им. И.И. Ползунова*

**Методические указания к выполнению лабораторного  
практикума по дисциплине “Проектирование  
программного обеспечения ”**

Для студентов направления 09.03.04 Программная инженерия

Барнаул 2020

Разработчики:

доцент Ананьев П.И.

Методические указания разработаны на основании Федерального государственного образовательного стандарта высшего образования - бакалавриат по направлению подготовки 09.03.04 Программная инженерия.

Методические указания рекомендованы к печати методической комиссией  
кафедры ПМ АлтГТУ

## Оглавление

<b>1</b>	<b>Цель и задачи лабораторного практикума</b>	<b>4</b>
<b>2</b>	<b>Организация выполнения лабораторного практикума</b>	<b>4</b>
<b>3</b>	<b>Содержание лабораторного практикума</b>	<b>5</b>
<b>3.1</b>	<b>Лабораторная работа №1</b>	<b>5</b>
	<i>Варианты заданий:</i>	<b>11</b>
<b>3.2</b>	<b>Лабораторная работа №2</b>	<b>13</b>
3.2.1	Определение понятия требования	13
3.2.2	Классификация требований	13
3.2.3	Сбор требований	15
3.2.4	Определение основных профилей пользователей	17
3.2.5	Сбор пользовательских историй	17
3.2.6	Анализ требований	20
<b>3.3</b>	<b>Лабораторная работа №3</b>	<b>21</b>
3.3.1	Что такое UML	21
3.3.2	Способы применения UML	22
3.3.3	Концептуальные области UML	24
3.3.4	Представления UML	28
3.3.5	Представление Use Case	30
<b>3.4</b>	<b>Лабораторная работа №4</b>	<b>35</b>
3.4.1	Объектная модель	35
3.4.2	Классы и объекты	35
3.4.3	Роль классов и объектов в анализе и проектировании	38
3.4.4	Объектно-ориентированный анализ	38
3.4.5	Выполнение объектно-ориентированного анализа	40
<b>3.5</b>	<b>Лабораторная работа №5</b>	<b>41</b>
3.4.1	Спецификации и проектирование	41
3.4.2	Процедурная абстракция	41
3.4.3	Абстракция данных	49
3.4.4	Исключительные ситуации	58
3.4.5	Абстракция итерации	64
	<b>Литература</b>	<b>67</b>

## **1 Цель и задачи лабораторного практикума**

Лабораторная работа является одной из форм подготовки специалистов ВУЗов. Лабораторный практикум – это самостоятельная работа студентов, подготовительная ступень к написанию курсовой и дипломных работ; он способствует приобретению практического опыта и соответствующих компетенций, необходимых программисту.

**Цель** лабораторного практикума по дисциплине «Проектирование программного обеспечения» — закрепление теоретических знаний и приобретение практических навыков по выявлению требований заказчика, проектированию баз данных и их реализации.

Для достижения указанной цели студентам необходимо решить следующие **задачи**:

- сформировать группу из трех человек для выполнения заданий;
- изучить выбранную предметную область и создать концепцию будущего продукта;
- выполнить сбор и анализ требований;
- построить диаграмму вариантов использования;
- построить объектную модель предметной области;
- разработать спецификации модулей будущей программы.

## **2 Организация выполнения лабораторного практикума**

Для выполнения лабораторных работ студенты формируют группы по 3 человека. Преподаватель назначает группе из 3-х студентов номер задания из вариантов, предложенных после теоретического материала первой лабораторной работы.

Выполнение каждой лабораторной работы состоит из следующих этапов:

1. Выполнение задания.
2. Создание отчета.
3. Проверка задания преподавателем.
4. Защита работы.

К сдаче зачета по дисциплине «Проектирование программного обеспечения» допускаются студенты, которые имеют положительные оценки по всем лабораторным работам.

### **3 Содержание лабораторного практикума**

#### **3.1 Лабораторная работа №1**

Тема: Разработка концепции продукта.

Задание: Для заданной предметной области разработать концепцию программного продукта, позволяющего автоматизировать протекающие в ней процессы. Выполнить задание в составе группы из трех человек в соответствии с вариантом.

##### Теоретический материал

Разработка концепции продукта заключается в анализе организационной системы предприятия и технологии достижения требуемых целей в этой системе. Можно выделить следующие этапы этого процесса:

- **Понять предметную область**
- **Выделить проблемы для решения**
- **Найти возможные варианты решения**
- **Выбрать подходящий вариант решения**
- **Определить роль компьютерной программы в решении проблемы**
- **Определить пределы возможностей программы**
- **Оценить влияние программы на существующую систему**
- **Определить способ внедрения программы в существующую систему предприятия**
- **Оценить экономический эффект от внедрения программы**
- **Разработать функциональное описание для следующей стадии - системного анализа**

Разработка концепции продукта начинается с первых слов разговора с клиентом или с начальством. Работа аналитика начинается с понимания перспектив той системы предприятия, в которой работает его клиент. Обсуждение дел происходит именно из-за того, что в существующей системе имеются проблемы или неиспользованные возможности.

Программист-аналитик использует в своей работе знания основ работы предприятия, а также по возможности специальные знания по соответствующей отрасли производства. Его основным инструментом является способность задавать компетентные вопросы. В ходе обсуждения выясняются потребности и возможные пути решения проблем. Целью оценки системы является нахождение возможных путей решения проблемы. Этап начинается с анализа потребностей и имеющихся ресурсов, затем выясняются возможные подходы и, наконец, выбирается наиболее подходящий для дальнейшей разработки вариант. Если

обнаруживаются несколько примерно равноценных вариантов, то всю процедуру можно повторить заново. Повторный анализ может потребоваться и в других случаях, например, когда выбранный вначале путь решения не обеспечивает желаемого результата, или оказывается слишком дорогостоящим, или обнаруживается какие-то нежелательные побочные эффекты, и т.д..

Для лучшего понимания рассматриваемой системы желательно составить подробное описание того как она функционирует. В этом описании должна содержаться полная информация о процессах, которые протекают в организации, с указанием людей, в них участвующих. Также в описании должны быть отмечены документы, которые фигурируют в рассматриваемых процессах. Особенно важно разделить сотрудников организации по должностям. Это помогает при построении программы определить функционал, который будет доступен тому или иному сотруднику.

При оценке системы может применяться такой метод, как мозговой штурм. Он позволяет рассмотреть многие варианты в виде "А что, если сделать вот так?" Обычно предлагается усовершенствовать текущую систему, либо компьютеризировать ее. Желательно провести усовершенствование системы, не изменяя ее радикально и не нарушая функционирования. Конечно, при проведении только поверхностных изменений могут остаться более глубоко лежащие проблемы.

Варианты решения проблем могут содержать, как подходы, связанные с автоматизацией, так и организационные методы, которые могут помочь оптимизировать потоки документов в организации и устранить дублирование функций. При формулировании вариантов решения необходимо учитывать то, что они должны быть представлены людям далеким от программирования и не должны содержать специфические термины, понятные только разработчикам программного обеспечения.

Предлагая варианты реализации программы, бывает уместно остановиться и посмотреть на программу "издалека", попытавшись определить, на что она похожа. Часто разрабатываемая программа попадает в одну из специфических категорий. В подобном случае может оказаться выгоднее купить готовый программный продукт, а не тратить средства на разработку собственного.

После формулирования возможных способов решения проблем необходимо проанализировать плюсы и минусы каждого варианта. В списке не должны присутствовать варианты, которые не дают каких-то плюсов. Т.к. использование таких подходов изначально не принесет ничего хорошего.

Предлагая варианты реализации программы, бывает уместно остановиться и

посмотреть на программу "издалека", попытавшись определить, на что она похожа. Часто разрабатываемая программа попадает в одну из специфических категорий. В подобном случае может оказаться выгоднее купить готовый программный продукт, а не тратить средства на разработку собственного.

В результате проведенного анализа должно быть принято решение об использовании наиболее подходящего варианта.

Если решено создавать свою программу, то наличие четкого описания ее возможностей помогает предотвратить непредвиденное усложнение программы. Это усложнение может проявиться двояко - программа может либо оказаться более сложной сама по себе, либо начать выполнять большее число функций. Заранее составленное описание возможностей (что программа должна делать, а что - нет) позволяет ограничить такой рост функциональности.

Одним из важнейших этапов описания будущей программы является перечисление тех проблем, которые поможет решить реализуемый проект. Возможно не все проблемы будут решены, возможно какая-то их часть будет решаться организационными методами. Все это должно найти свое отражение при оценке системы.

При работе над большими проектами описание возможностей превращается в функциональную спецификацию. При отсутствии же детальной спецификации нужно хотя бы определить тип программы. Четко определенный тип программы помогает при создании и обслуживании данных - если изменение модели данных приводит к изменению типа программы, то такое изменение недопустимо. Такие изменения обычно выражены во внезапном появлении новых объектов данных.

Программы разрабатываются разными способами. Много очень полезных приложений было разработано "за одну ночь". Конечно, при разработке приложения можно не составлять никаких описаний. В такой ситуации программист должен удостовериться, что обе стороны не нарушат своих обязательств. Дело в том, что если пользователь не определил требуемых ему возможностей программы, то программист не может сказать, сколько времени уйдет на ее написание.

Любая программа оказывает воздействие на аппаратуру, локальные сети и все прочее, а кроме того, наибольшее воздействие, пусть косвенное, будет оказано на людей.

После создания функционального описания программы проясняется и ее воздействие на существующую систему предприятия. Компьютерная программа оказывает влияние на две стороны системы: на существующее аппаратное обеспечение и на людей.

Влияние на аппаратуру является наиболее очевидным, и оказывает воздействие на

полную стоимость системы. Система должна поддерживать современные приложения и работать в локальной сети. Наличие любого отклонения от этого условия увеличивает стоимость проекта. Представление о реальном эффекте от внедрения системы можно получить, рассматривая его не как "улучшения", а как "изменения". На этом этапе надо проанализировать какая техника уже есть на предприятии и ее тип, есть ли на предприятии локальная сеть, если выход в глобальную, существует ли web-сервер организации и где он размещен. После этого можно сформулировать, как повлияет внедрение программы на существующую систему. Какую технику придется купить, какую модернизировать, придется устанавливать сервер или нет, протягивать сеть или нет.

Воздействие на людей является более утонченным. Люди смотрят на компьютеры как на решение всех проблем, делающее жизнь лучше. Точнее, компьютеры делают жизнь другой. И люди реагируют на эти изменения по-разному.

Внедрение программы изменит образ работы многих людей. В этой ситуации некоторые могут почувствовать себя лишними, подумать, что им грозит увольнение, и потерять уверенность в завтрашнем дне.

Управляющий персонал должен заранее подготовиться к такому явлению. Ненормальное психологическое состояние работников может свести на нет все преимущества компьютеризации, которая на самом деле призвана освободить людей от шаблонной работы, предоставив им больше творчества.

Если решено заменить старую систему новой, нужно в течение некоторого времени использовать эти системы вместе.

Данный этап оценки системы служит для заключения предыдущих этапов в твердые рамки по времени. Зная необходимые для создания новой системы затраты сил и средств, нужно определить, сколько это займет времени. Очевидно, что ответ на этот вопрос сильно зависит от конкретной обстановки. Ниже приводится список ключевых вопросов:

- Какова последовательность действий при внедрении системы?
- Когда нужно запустить программное обеспечение? Когда должны завершаться остальные этапы внедрения?
- В какой момент времени нужно установить требуемую или доработать имеющуюся аппаратуру?
- Как организовать обучение персонала?
- Если требуется преобразование данных, то когда его нужно произвести?
- Как долго придется проверять программное обеспечение, прежде чем оно станет



основной системой?

Ответы на все эти вопросы должны быть сформулированы и приведены в описании.

После этого можно переходить к последнему этапу оценки системы, который покажет реальную жизнеспособность проекта.

Способ определения экономической эффективности зависит от отношений программиста-аналитика с заказчиком. Если программист приглашен извне, он может отвечать не только за создание программы, но и за ее внедрение и эксплуатацию, выступая в качестве консультанта. Во многих остальных случаях программист отвечает только за написание текста программы, оставляя решение других проблем компетентным лицам.

Затраты делятся на прямые и косвенные. Прямые включают в себя программирование, усовершенствование аппаратуры, покупку обучающих программ и т.д. Косвенные расходы идут на обучение персонала и подобные нужды.

Определить экономическую эффективность - значит сравнить получаемую прибыль с расходами. Обычно это делается неформально - "внедрение программы окупит все затраченные средства", "все пользуются компьютерными системами, и нам она тоже нужна" и т.п. Более формально экономический эффект рассчитывается путем детального анализа преимуществ от внедрения программы. При этом следует ответить на ряд вопросов, касающихся выполнимости проекта. Способна ли программа работать? Ответ на этот вопрос может быть отрицательным по ряду причин. Программа может не решить нужные проблемы или вызвать слишком много побочных эффектов. Или же для успешной эксплуатации программы может не хватить денег или персонала.

Результатом выполнения оценки системы служит функциональное описание, пункты которого приведены в таблице 1. Оно служит для формулировки взаимодействия проектируемой системы с внешним миром, а также входных и выходных данных для этой системы.

Этот документ, на составление которого тратится много времени, становится основой для всей дальнейшей деятельности, включая системный анализ.

Аналитик получает информацию разными способами. Важно все время помнить об основах текущей системы, тогда беседы с текущими и потенциальными пользователями системы позволят выяснить, какие данные вводятся в систему и что ожидается у нее на выходе. Неоценимую услугу здесь могут оказать существующие формы и отчеты, дающие нужную информацию в сжатой форме.

Таблица 1. Функциональное описание.

<b>Область рассмотрения</b>	<b>Характеристика приложения</b>
Аппаратные средства	Указывается аппаратная платформа, на которой будет функционировать программа
Метод ввода	Указывается тип интерфейса и используемые технические средства
Метод вывода	Указывается тип интерфейса и используемые технические средства
Возможности	Описание возможностей программы
Входные данные	Перечисление входных данных
Выходные данные	Перечисление выходных данных и отчетов
Специальные особенности	Особенности программы, которые не являются общепринятыми (например поддержка интерфейса на разных языках, шифрование данных и т.д.)
Пользователи	Список пользователей системы (из предметной области)
Предполагаемый объем	Предполагаемый объем данных (в записях), который будет заполняться за год работы системы. Этот и следующий пункты необходимы, чтобы можно было принять решение о том, как будут храниться данные. (С использованием СУБД или нет, и можно было бы определить класс СУБД)
Предполагаемые темпы	Указывается (в %) как будет увеличиваться объем данных за год.
Время	Время, необходимое для реализации проекта.

Информация высокого уровня - экраны отчетов, формы, таблицы и т.д. дает возможность довольно просто описать требуемые элементы приложения. Например, наличие списка данных позволяет определить примерное количество таблиц, отношения между ними, число экранов поддержки, а также примерные затраты на их разработку. На этом этапе можно также выяснить, не потребуется ли для обработки данных каких-либо дополнительных аппаратных средств. Так, интерактивная система, работающая в сети, требует очень быстрого обмена по этой сети.

Если программист-аналитик имеет опыт в решении подобных проблем, то уже на этом этапе он сможет оценить затраты на разработку программы, несмотря на то, что еще не вся информация известна. Насколько сильно следует детализировать функциональное описание? Настолько, сколько требуется для определения затрат на разработку. Если вам уже приходилось решать аналогичную проблему и у вас есть опыт, то количество деталей можно уменьшить.

Целью функционального описания является определение взаимодействия программы с аппаратурой и пользователями, а также число и сложность структур данных. Подробности вроде конфликтных ситуаций также важны, но ими лучше заниматься при проведении

анализа системы.

На этом этапе вы даете оценку только необходимой работе, а затраты времени еще предстоит уточнить. И для этого вам будет необходимо знать уровень подготовки тех людей, которые будут непосредственно писать программу, а также о том, была ли ранее написана подобная программа, и т.д. Такая информация станет известной после проведения анализа системы.

***Варианты заданий:***

1. Информационная система для обеспечения деятельности депо по ремонту пассажирских вагонов.
2. Информационная система для обеспечения деятельности судоходной компании.
3. Информационная система для обеспечения деятельности отдела гарантийного ремонта товаров.
4. Информационная система для обеспечения деятельности отдела учета налогообложения физических лиц городской налоговой инспекции.
5. Информационная система для обеспечения деятельности отдела заселения муниципальных общежитий администрации города.
6. Информационная система для обеспечения деятельности Государственной автомобильной инспекции по безопасности дорожного движения города.
7. Информационная система для ведения реестра имущества университетского городка.
8. Информационная система для обеспечения деятельности рекламного агентства.
9. Информационная система для обеспечения деятельности отдела вневедомственной охраны квартир.
10. Информационная система для обеспечения деятельности отдела аренды.
11. Информационная система для обеспечения деятельности гостиницы.
12. Информационная система для обеспечения деятельности предприятия по учету платы за потребленную электроэнергию.
13. Информационная система обработки результатов экзаменационной сессии.

14. Информационная система обслуживания клиентов в отделении банка.
15. Информационная система для обслуживания читателей в библиотеке.
16. Информационная система для автоматизации работы автопарка.
17. Информационная система «Исполнительская дисциплина».
18. Информационная система «Деканат».
19. Информационная система «Абитуриент».
20. Информационная система обслуживания больных в поликлинике.
21. Информационная система «Расписание».
22. Информационная система «Библиотеки города».
23. Информационная система для обслуживания клиентов санатория.
24. Информационная система «Отдел кадров».
25. Информационная система «Метеослужба».

## 3.2 Лабораторная работа №2

Тема: Сбор и анализ требований.

Задание к работе: Для предметной области из лабораторной № 1 выполнить сбор и анализ бизнес-требований, пользовательских и функциональных требований. Определить основные профили пользователей. Собрать пользовательские истории.

Теоретический материал

### 3.2.1 Определение понятия требования

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

Введем еще одно определение. **Требования - это исходные данные, на основании которых проектируются и создаются автоматизированные информационные системы.** Первичные данные поступают из различных источников, характеризуются противоречивостью, неполнотой, нечеткостью, изменчивостью. Требования нужны в частности для того, чтобы Разработчик мог определить и согласовать с Заказчиком временные и финансовые перспективы проекта автоматизации. Поэтому значительная часть требований должна быть собрана и обработана на ранних этапах создания АИС. Однако собрать на ранних стадиях все данные, необходимые для реализации АИС, удастся только в исключительных случаях. На практике процесс сбора, анализа и обработки растянут во времени на протяжении всего жизненного цикла АИС, зачастую нетривиален и содержит множество подводных камней.

### 3.2.2 Классификация требований

**Требования к продукту.** В своей основе требования - это то, что формулирует заказчик. Цель, которую он преследует - получить хороший конечный продукт: функциональный и удобный в использовании. Поэтому требования к продукту являются основополагающим классом требований. Более подробно требования к продукту детализируются в следующих ниже классификациях.

**Требования к проекту.** Вопросы формулирования требований к проекту, т.е. к тому, как Разработчик будет выполнять работы по созданию целевой системы, казалось бы, не лежат в компетенции Заказчика. Без регламентации процесса Заказчиком легко можно было бы обойтись, если бы все проекты всегда выполнялись точно и в срок. Однако, к сожалению, мировая статистика результатов программных проектов говорит об обратном. Заказчик, вступая в договорные отношения с Разработчиком, несет различные риски, главными из

которых является риск получить продукт с опозданием, либо ненадлежащего качества. Основные мероприятия по контролю и снижению риска - регламентация процесса создания программного обеспечения и его аудит.

Насколько подробно Заказчику следует регламентировать требования к проекту - вопрос риторический. Ответ на него зависит от множества факторов, таких, как ценность конечного продукта для Заказчика, степень доверия Заказчика к Разработчику, сумма подписанного контракта, увязка срока сдачи продукта в эксплуатацию с бизнес-планами Заказчика и т.д. Однако, со всей определенностью можно сказать следующее:

1. регламентация процесса Заказчиком позволяет снизить его риски;
2. мероприятия Заказчика по регламентации процесса приводят к дополнительным накладным расходам. Требуется найти разумный компромисс между степенью контроля рисков и величиной расходов.

В качестве *требований к проекту* может быть внесен *регламент отчетов Разработчика, совместных семинаров по оценке промежуточных результатов, определены характеристики компетенций участников рабочей группы, исполняющих проект, их количество, указана методология управления проектом.*

Внедрение ИС на предприятии всегда преследует конкретные бизнес-цели - такие, как, например, повышение прозрачности бизнеса, сокращение сроков обработки информации, экономия накладных расходов и т.д. Современные информационные системы - это крупные программные системы, содержащие в себе множество модулей, функциональных, интерфейсных элементов, отчетов и т.д. Поэтому существует необходимость разделения требований по уровням. Уровни требований связаны, с одной стороны, с уровнем абстракции системы, с другой - с уровнем управления на предприятии. Обычно выделяют три уровня требований.

- На *верхнем уровне* представлены так называемые *бизнес-требования* (business requirements). Примеры бизнес-требования: система должна сократить срок оборачиваемости обрабатываемых на предприятии заказов в три раза. Бизнес-требования обычно формулируются топ-менеджерами, либо акционерами предприятия.
- *Следующий уровень - уровень требований пользователей* (user requirements). Пример требования пользователя: система должна представлять диалоговые средства для ввода исчерпывающей информации о заказе, последующей фиксации информации в базе данных и маршрутизации информации о заказе к сотруднику, отвечающему за его планирование и исполнение. Требования пользователей часто бывают плохо

структурированными, дублирующимися, противоречивыми. Поэтому для создания системы важен третий уровень, в котором осуществляется формализация требований.

- **Третий уровень - функциональный** (functional requirements). Пример функциональных требований (или просто функций) по работе с электронным заказом: заказ может быть создан, отредактирован, удален и перемещен с участка на участок.

*Существуют объективные противоречия между требованиями различных уровней.* Так, очевидным бизнес-требованием является требование о полноте информации, собираемой на рабочих местах пользователей в единую базу данных. Чем полнее информация - тем глубже база для анализа деятельности и принятия решений. С другой стороны, конкретному пользователю системы вполне может быть достаточно использования только той части информации, которая влияет на выполнение его основных функций.

Важные правила внедрения и использования АИС на предприятии - **"Одна точка сбора"**, "Данные собираются там, где они появляются". Использование этих правил позволяет избежать затрат на необоснованное дублирование информации и, что важнее - потерь от ошибок учета, неизбежно возникающих при дублировании точек ввода.

Внедрение АИС на предприятии приводит к необходимости оснащения всех точек ввода информации автоматизированными рабочими местами (АРМ), обучению персонала и, зачастую, оптимизации и повышению уровня формализации рабочих процессов, выполняемых персоналом. Поэтому **внедрение АИС - непростой процесс**, часто требующий "перекройки человеческого материала" и встречающий сопротивление со стороны пользователей, которые не готовы, либо не хотят работать по-новому.

### 3.2.3 Сбор требований

#### *Сбор и анализ бизнес требований*

Первым и самым важным этапом в разработке продукта является **сбор бизнес требований**. Цель этой работы — определить основные требования бизнеса (исходные данные, истинные цели, которым должен служить продукт и проблемы, которые нужно преодолеть).

Для продуктов под заказ и продуктов для открытого рынка процесс сбора бизнес требований существенно различается.

- **Продукт под заказ** — заказчик определен с самого начала, ему известны начальные предпосылки (стимулы) для инициации проекта и проблемы, которые продукт должен решать. В связи с этим, сбор требований начинается с определения исходных предпосылок, целей продукта и описания сценариев работы пользователей

с будущим продуктом. Анализ конкурентных продуктов, которые могут удовлетворять схожие сценарии, делается в самом конце.

- **Продукт для открытого рынка** — сектор рынка с самого начала может быть не определен, а цели продукта основываются на конкурентном анализе.

В результате, сразу за определением исходных предпосылок (стимулов) идет обзор конкурентов, далее идет определение целевого сегмента рынка и потребностей его заказчиков и только после этого определяются цели продукта и критерии его успеха.

Таблица 2. Последовательность задач, выполняемых на этапе сбора бизнес требований.

<b>Продукт под заказ</b>	<b>Продукт для открытого рынка</b>
<i>Определение исходных стимулов</i>	<i>Определение исходных стимулов</i>
<i>Определение целей продукта и критериев успеха</i>	<i>Обзор конкурентов</i>
<i>Определение потребностей клиента</i>	<i>Определение целевого сегмента рынка</i>
<i>Обзор конкурентов</i>	<i>Определение потребностей клиента</i>
	<i>Определение целей продукта и критериев успеха</i>

### **Определение стимулов**

На данном этапе указываются основные причины, которые стимулируют принятие решения о создании этого продукта.

Как правило, причиной создания продукта может служить одна или несколько из нижеперечисленных проблем, благоприятных возможностей или требований бизнеса:

- Потребность рынка (например, банк авторизует проект миграции клиентского ПО на платформу Mac OS в ответ на возросшую рыночную долю этой ОС);
- Производственная необходимость (например, R&D отдел авторизует проект интеграции системы контроля версий с сервером статистики для повышения уровня контроля за достижениями сотрудников);
- Потребность заказчика (например, крупный сборщик компьютеров авторизует проект кастомизации существующего ПО под компьютеры собственного производства для достижения большей совместимости);
- Технический прогресс (например, производитель ПО авторизует проект разработки новой версии продукта, использующей новые возможности только что вышедшей ОС);
- Юридические ограничения или нормы (например, банк может авторизовать проект перехода продукта на новый метод шифрования, соответствующий критериям современных норм безопасности, для работы с государственными заказчиками).



### ***Сбор пользовательских требований.***

Уже известны основные сценарии использования будущего продукта, определены бизнес требования и высокоуровневые возможности. Теперь нужно детализировать требования к продукту. Основная работа на этом этапе ведется с будущими пользователями продукта.

При использовании итеративного процесса вся функциональность поделена на части (это описано ранее), поэтому сбор и анализ требований, а позднее проектирование системы ведется строго в рамках функциональности, которая должна быть реализована в текущей итерации.

### **3.2.4 Определение основных профилей пользователей**

Для того чтобы продукт был удобен пользователям и делал «то, что надо», сначала надо определить, кто же им будет пользоваться.

На практике, даже для домашних продуктов очень сложно определить «среднего пользователя», чтобы на основе его потребностей проектировать продукт. Для корпоративных продуктов это попросту невозможно: директор будет пользоваться одной функциональностью, его секретарь другой, а бухгалтер третьей. По этой причине перед началом сбора требований должны быть определены основные профили пользователей.

### **3.2.5 Сбор пользовательских историй**

Настала очередь начать общение с конечными пользователями и узнать, для достижения каких целей будет использоваться будущий продукт. Лучшим методом для достижения этой цели является сбор пользовательских историй.

***Пользовательская история — это вариант использования будущего продукта в конкретной ситуации с целью достижения измеримого результата.*** Каждая пользовательская история должна приносить пользу — не должно быть историй, которые выполняют действия ради действий.

Пользовательские истории могут содержать как сложные инструкции с ответвлениями, так и конкретные примеры. Если действия пользователей продиктованы скорее здравым смыслом, нежели инструкцией, то пользовательская история должна содержать пример реальной ситуации, а их систематизация и оптимизация должна быть оставлена на потом. Если же бизнес процессы, которые автоматизирует продукт, строго формализованы, то пользовательская история должна содержать алгоритмы действий продукта или людей, работающих с ней.

Важно помнить, что пользовательская история не описывает способ работы с конкретным продуктом и использует только термины предметной области. Она должна быть понятной и родной людям, которые знают процесс, автоматизируемый продуктом. Способ реализации всегда должен оставаться за кадром (он будет определен намного позже).

Далее указаны поля вариантов использования (полужирным шрифтом помечены обязательные):

- **Идентификатор** («Уникальный номер» плюс «Имя»).
- Источник/Автор.
- Дата создания.
- Профиль пользователя. *Если в рамках истории происходит взаимодействие между различными пользователями, то их следует указать через запятую, а в последовательности действий описать принадлежность пользователей к ним. Также удобно определить профиль по умолчанию — он будет иметься в виду, если поле не заполнено.*
- **Приоритет.**
- Частота использования (числа от 1 до 10 или текстовые значения: «почти никогда», «один раз», «редко», «время от времени», «часто», «очень часто», «каждый N минут/часов/дней»)
- **Родительское бизнес требование.**
- Предусловие. *Описание начального состояния. Может содержать причины/порядок действий, которые привели к проблеме, решаемой в пользовательской истории.*
- **Цель/ Результат.** *Цели, которые преследует пользователь в рамках пользовательской истории. По возможности следует указывать мотивы, которые побудили его (зачем это надо?) и ожидаемый результат.*
- Последовательность действий. *Может быть указан в качестве примера или содержать алгоритм взаимодействия с системой для формализованных процессов.*

Основной акцент в описании пользовательской истории должен быть сделан на цели, которую хочет достичь её автор. В качестве последовательности действий можно указать, какого поведения ожидает пользователь от продукта при достижении поставленной цели, но ни в коем случае не стоит пытаться смоделировать поведение продукта для этой ситуации, так как на этом этапе это будет напрасная трата времени и сил (исключение составляют продукты, в основе которых положен стандарт).

Каждая пользовательская история имеет одно или больше родительских бизнес требований и ее главная цель описать наиболее удачные способы их удовлетворения. Именно на основе родительского требования устанавливается приоритет истории. Если один из пользователей просит реализовать историю, которая не подчинена бизнес требованию, нужно обговорить добавление соответствующего бизнес требования в продукт с его непосредственным инвестором. В случае положительного решения, нужно отразить изменение в подходящем сценарии.

Самая главная ошибка, которую делают при сборе требований к продуктам, это формирование образа продукта на основе ответов на вопрос «Что должен делать продукт» или еще хуже «Как должен работать продукт». Целью же сбора требований, является получение ответа на вопрос — «Для чего нужен продукт». А потому, единственный вопрос, который должен задаваться пользователю — это «Зачем?».

*Пример: пользователь продукта начинает рассказывать, что ему нужно три мастера, которые бы выполняли определенные задачи, просит использовать определенные цветовые схемы и форматы данных. Как правило, такой случай заканчивается полным фиаско. В процессе ввода продукта в эксплуатацию окажется, что пользователь не учёл мнения других людей, которые будут работать с продуктом, не указал исчерпывающих требований к аппаратной платформе. И, как результат, первая версия продукта будет иметь ошибки в проектировании (например, клиентская часть продукта не будет поддерживать Mac OSXi, как следствие, сильно ограничит рынок сбыта), будет отсутствовать часть важнейшего функционала, без которого невозможно использовать её в рамках основных бизнес процессов. А значит, продукт не будет отвечать требованиям реального бизнеса.*

Нужно очень четко контролировать поступающую от заказчика информацию. Требования должны быть продиктованы бизнесом и быть нацелены на достижение реальных и измеряемых результатов. Одним словом реализация требования должна приносить пользу, а не являться самоцелью. Если пользователь говорит, что неплохо было бы сделать «мастер того-то и того-то» нужно мгновенно среагировать и задать вопрос — «зачем этот мастер ему нужен». Как правило, он с радостью расскажет о реальных мотивах, которые и интересуют!

*Пример: пользователь сказал, что ему нужна функция авто-ввода для поля «фамилия». Это требование отвечает на вопрос «что», а потому прежде чем его вносить, нужно задать вопрос: «зачем нужна эта функция?» ответом на этот вопрос, скорее всего, будет что-то вроде «Нужно максимально сильно ускорить*

*процесс ввода персональных данных оператором о клиенте». Требование будет выглядеть так:*

*Нужно максимально сильно ускорить процесс ввода персональных данных оператором о клиенте. Одним из возможных способов может быть реализация функции авто-ввода фамилии из базы уже имеющих клиентов.*

*Теперь команда проектирования пользовательского интерфейса знает реальное требование к будущему интерфейсу, а команда тестирования, знает критерии для его тестирования.*

### **3.2.6 Анализ требований**

После завершения этапа сбора требований вы **уже располагаете всей необходимой информацией** о требованиях к будущей системе, но эта информация не систематизирована и часто дублируется. Для небольшого продукта это неважно, и он может обойтись без большинства стадий анализа, быстро перейдя к проектированию, но для крупного продукта это приведет к большому количеству ошибок в процессе проектирования и, как следствие, к увеличению бюджета и срока разработки. Чем больше продукт, тем более важной является стадия анализа.

Основной целью анализа требований является их систематизация и избавление от дублируемых данных. Это достигается за счет разделения пользовательских историй на отдельные пакеты по функциональному признаку и их иерархической структуризации.

Главная цель формирования пакетов — упростить доступ к нужным данным, за счет того, что все варианты использования относящихся к определенной функциональности можно будет увидеть на одной странице.

Пакеты формируются из пользовательских историй, которые описывают схожую деятельность или способ достижения схожего результата. Как правило, всего они подчинены одному бизнес-требованию.

### 3.3 Лабораторная работа №3

Тема: Разработка вариантов использования.

Задание: Для изученной в лабораторных работах 1-2 предметной области построить диаграмму Use Case с использованием любого CASE-средства.

Теоретический материал

#### 3.3.1 Что такое UML

Унифицированный язык моделирования (UML) — это язык визуального моделирования для решения задач общего характера, который используется при определении, визуализации, конструировании и документировании артефактов программной системы. С помощью языка UML можно фиксировать решения, принятые при создании различных систем. Он используется для того, чтобы лучше понимать, проектировать, поддерживать и контролировать эти системы. UML можно использовать со всеми методами разработки, во всех предметных областях и на всех этапах жизненного цикла программы. Этот язык призван объединить в единый стандартный подход весь опыт, который был накоплен в процессе использования старых способов моделирования, а также все лучшее из современных методов создания программного обеспечения (ПО). Он включает в себя семантические концепции, нотацию и руководящие указания. UML состоит из четырех частей, описывающих различные аспекты системы: статические, динамические, организационные и относящиеся к окружению. Предполагается, что язык UML будет поддерживаться средствами для интерактивного визуального моделирования, включающими в себя генераторы кода и отчетов. Спецификация UML не определяет конкретный процесс разработки, однако использовать этот язык моделирования удобнее всего в итеративном процессе. Впрочем, его можно применять в большинстве существующих объектно-ориентированных процессов разработки.

UML позволяет отображать и статическую структуру, и динамическое поведение системы. Система моделируется как группа дискретных объектов, которые взаимодействуют друг с другом таким образом, чтобы удовлетворить требования пользователя. В статической структуре задаются типы объектов, значимые для системы и ее реализации, а также отношения между этими объектами. Динамическое поведение определяет историю объектов и их взаимодействие для достижения конечной цели. Наиболее полного и разностороннего понимания системы можно достичь при моделировании с различных, но взаимосвязанных точек зрения.

В UML также есть конструкции для распределения моделей по пакетам, с помощью которых разработчики могут разделять систему на составные части. В сложных программных разработках эти конструкции позволяют управлять различными версиями пакетов, а также описывать и контролировать зависимости между ними. В языке UML есть также структуры, позволяющие отображать решения по реализации системы и помещать выполняемые блоки программы в компоненты.

UML не является языком программирования. С его помощью можно писать программы, но в нем отсутствуют свойственные большинству языков программирования синтаксические и семантические соглашения, облегчающие работу программиста. С помощью инструментальных средств, поддерживающих UML и содержащих генераторы кода, на основе созданной модели можно получить программный код на различных языках и наоборот, по исходному коду уже существующих программ можно восстановить их UML-модели.

UML — не формальный язык для доказательства теорем. Такие языки существуют, но их довольно трудно освоить и применить для решения большинства общих задач. UML же является языком моделирования общего назначения. В более специфических случаях, например, для разработки пользовательских интерфейсов, проектирования схем со сверхвысоким уровнем интеграции или (Создания систем, обладающих искусственным интеллектом, целесообразно использовать более специализированные инструменты и языки моделирования. - UML — язык дискретного моделирования. Он не предназначен для разработки непрерывных систем, встречающихся в физике и механике. UML создавался как язык моделирования общего назначения для применения в таких дискретных системах, как программное обеспечение, аппаратные средства или цифровая логика.

### **3.3.2 Способы применения UML**

Основу роли UML в разработке программного обеспечения составляют разнообразные способы использования языка, те различия, которые были перенесены из других языков графического моделирования.

Существуют три режима использования UML разработчиками: режим эскиза, режим проектирования и режим языка программирования. Безусловно, самый главный из трех - это режим использования *UML для эскизирования*. В этом режиме разработчики используют UML для обмена информацией о различных аспектах системы. В режиме проектирования можно использовать эскизы при прямой и обратной разработке. При *прямой разработке (forward-engineering)* диаграммы рисуются до написания кода, а при

*обратной разработке (reverse-engineering)* диаграммы строятся на основании исходного кода, чтобы лучше понять его.

Сущность эскизирования, или эскизного моделирования, в избирательности. В процессе прямой разработки вы делаете наброски отдельных элементов программы, которую собираетесь написать, и обычно обсуждаете их с некоторыми разработчиками из вашей команды. При этом с помощью эскизов вы хотите облегчить обмен идеями и вариантами того, что вы собираетесь делать. Вы обсуждаете не всю программу, над которой намереваетесь работать, а только самые важные ее моменты, которые вы хотите донести до коллег в первую очередь, или разделы проекта, которые вы хотите визуализировать до начала программирования. Такие совещания могут быть очень короткими: 10-минутное совещание по нескольким часам программирования или однодневное совещание, посвященное обсуждению двухнедельной итерации. При обратной разработке вы используете эскизы, чтобы объяснить, как работает некоторая часть системы. Вы показываете не все классы, а только те, которые представляют интерес и которые стоит обсудить перед тем, как погрузиться в код. Поскольку эскизирование носит неформальный и динамичный характер и вам нужно делать это быстро и совместно, то наилучшим средством отображения является доска. Эскизы полезны также и в документации, при этом главную роль играет процесс передачи информации, а не полнота. Инструментами эскизного моделирования служат облегченные средства рисования, и часто разработчика не очень придерживаются всех строгих правил UML.

Напротив, язык *UML* как средство *проектирования* нацелен на полноту. В процессе прямой разработки идея состоит в том, что проект разрабатывается дизайнером, чья работа заключается в построении детальной модели для программиста, который будет выполнять кодирование. Такая модель должна быть достаточно полной в части заложенных проектных решений, а программист должен иметь возможность следовать им прямо и не особо задумываясь. Дизайнером модели может быть тот же самый программист, но, как правило, в качестве дизайнера выступает старший программист, который разрабатывает модели для команды программистов. Причина такого подхода лежит в аналогии с другими видами инженерной деятельности, когда профессиональные инженеры создают чертежи, которые затем передаются строительным компаниям.

Проектирование может быть использовано для всех деталей системы либо дизайнер может нарисовать модель какой-то конкретной части. Общий подход состоит в том, чтобы дизайнер разработал модели проектного уровня в виде интерфейсов подсистем, а затем дал возможность разработчикам поработать над реализацией подробностей

При обратной разработке цель моделей состоит в представлении подробной информации о программе или в виде бумажных документов или в виде, пригодном для интерактивного просмотра с помощью графического браузера. В такой модели можно показать все детали класса в графическом виде, который разработчикам проще понять.

При разработке моделей требуется более сложный инструментарий, чем при составлении эскизов, так как необходимо поддерживать детальность, соответствующую требованиям поставленной задачи. Специализированные CASE-средства попадают в эту категорию. Инструменты прямой разработки поддерживают рисование диаграмм и копирование их в репозиторий с целью сохранения информации. Инструменты обратного проектирования читают исходный код, записывают его интерпретацию в репозиторий и генерируют диаграммы. Инструменты, позволяющие выполнять как прямую, так и обратную разработку, называются *двухсторонними (round-trip)*.

Некоторые средства используют исходный код в качестве репозитория, а диаграммы используют его для графического представления. Такие инструменты более тесно связаны с программированием и часто встраиваются прямо в средства редактирования исходного кода. Граница между моделями и эскизами довольно размыта, но различия остаются в том, что эскизы сознательно выполняются неполными, подчеркивая важную информацию, в то время как модели нацелены на полноту, часто имея целью свести программирование к простым и до некоторой степени механическим действиям.

Чем дольше вы работаете с UML, а программирование становится все более механическим, тем очевиднее становится необходимость перехода к автоматизированному созданию программ. Действительно многие CASE-средства так или иначе генерируют код, что позволяет автоматизировать построение значительной части системы. В конце концов, вы достигнете такой точки, когда сможете описать с помощью UML всю систему и перейдете в режим использования *UML в качестве языка программирования*. В такой среде разработчики рисуют диаграммы, которые компилируются прямо в исполняемый код, а UML становится исходным кодом. Очевидно, что такое применение UML требует особенно сложных инструментов.

### **3.3.3 Концептуальные области UML**

Все концепции и модели языка UML можно отнести к четырем концептуальным областям.

#### **Статическая структура**

В первую очередь, любая точная модель должна определять полное множество объектов, то есть ключевые концепции приложения, их внутренние свойства и отношения между собой. Эта структура и есть статическое представление системы. Концепции приложения моделируются



как классы, каждый из которых описывает тип дискретных объектов, содержащих определенную информацию и взаимодействующих между собой для реализации некоторого поведения. Информация, которую содержат объекты, моделируется как атрибуты, поведение — как операции. Используя механизм обобщения, несколько классов могут иметь одну общую структуру. Класс-потомок содержит в себе структуру и поведение, унаследованные от общего класса-предка, а также собственную, уникальную для него структуру и поведение. В процессе работы одни объекты имеют связи с другими. Такие отношения между объектами моделируются как ассоциации между соответствующими классами. Другие отношения между элементами, включая отношения между уровнями абстракции, связывание фактических значений с формальными параметрами шаблона, наделение правами и использование одного элемента другим, группируются как отношения зависимости. Классы могут иметь интерфейсы, которые описывают поведение класса, видимое извне. Имеются еще такие виды отношений, как зависимости включения и расширения элементов Use Case. Статическое представление системы изображается с помощью диаграмм классов или их вариаций. Его можно использовать для генерации большинства объявлений структуры данных в программе.

В языке UML существуют и другие элементы диаграмм: интерфейсы, типы данных, элементы Use Case и сигналы. Они носят общее название классификаторов и ведут себя в большинстве случаев как классы, но с некоторыми дополнениями и ограничениями для каждого типа классификатора.

### **Элементы проектирования**

Модели UML создаются как для логического анализа, так и для проектирования, обеспечивающего реализацию системы. Некоторые конструкции в модели, представляют собой проектные элементы. Структурированный классификатор раскрывает реализацию класса в виде набора частей, скрепленных между собой соединителями. Класс может инкапсулировать свою внутреннюю структуру за внешне видимыми портами. Кооперация моделирует набор объектов, играющих роли в меняющемся контексте. Компонент — это замещаемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает их реализацию. Компонент должен легко замещаться другими компонентами с тем же набором интерфейсов.

### **Элементы развертывания**

Узел — это вычислительный ресурс периода прогона, который определяет местонахождение исполняемых компонентов и объектов.

Артефакт — это физическая единица информации или описания поведения вычислительной системы. Артефакты развертываются в узлах. Артефакт может быть олицетворением (манифестацией) компонента (иначе говоря, его реализацией).

Представление развертывания системы описывает конфигурацию узлов работающей системы и расположение артефактов в этих узлах, включая отношения манифестации с соответствующими компонентами.

### **Динамическое поведение**

Существует три способа для моделирования поведения. Первый представляет историю жизни объекта и его взаимодействия с остальным миром. Второй описывает паттерны взаимодействия для набора соединенных объектов при реализации определенного поведения. Третий способ — это описание эволюции процесса исполнения программы в ходе осуществления ею разнообразной деятельности. Поведение отдельно взятого объекта описывается конечным автоматом. При этом рассматривается реакция объекта на события (на основе его текущего состояния), действия, из которых состоит эта реакция, и переход в новое состояние. Конечные автоматы изображаются на диаграммах автоматов.

Взаимодействие отражает структурированный классификатор или кооперацию с потоком сообщений между частями. Взаимодействия показываются на диаграммах последовательности и диаграммах коммуникации. Диаграммы последовательности выделяют упорядочение сообщений по времени, а диаграммы коммуникации — взаимосвязи объектов.

Деятельность описывает процесс выполнения вычислений. Она моделируется как набор узлов деятельности, связанных потоками управления и потоками данных. Деятельность может моделировать как последовательное, так и параллельное поведение. Она включает в себя традиционные конструкции потока управления — разветвления и циклы. На диаграммах деятельности можно изображать ход вычислений, а также рабочие потоки<sup>1</sup> в человеческих организациях.

В основе всех представлений поведения лежит набор элементов Use Case, каждый из которых описывает некий фрагмент функциональности системы с точки зрения актера — внешнего пользователя системы. Наряду со статической структурой элементов Use Case и участвующими в них актерами, представление Use Case содержит динамические последовательности сообщений между актерами, и даруемой, обычно выраженные в виде диаграмм последовательности или обычного текста.

### **Организация модели**

Компьютеры могут работать с большими «плоскими» моделями, а вот люди — нет. Поэтому при работе над большими системами вся информация о модели должна быть поделена на связанные понятные части, чтобы команды разработчиков могли параллельно работать над различными разделами системы. Даже при разработке небольших систем лучше разделить

содержимое модели на несколько пакетов приемлемого размера. Пакетами в языке UML называются иерархически организованные блоки моделей. Их можно использовать для хранения, контроля доступа, управления конфигурацией и конструирования библиотек, содержащих фрагменты моделей многократного пользования. Зависимости между пакетами определяются совокупностью зависимостей между содержимым пакетов и могут также задаваться системной архитектурой. Таким образом, содержимое пакетов должно соответствовать зависимостям между пакетами и заданной архитектуре системы.

### **Профили**

Как бы велики ни были возможности языка, людям рано или поздно захочется их расширить. UML обладает способностью к расширению, которой должно хватать на обычные, повседневные нужды, не требующие изменений в основных положениях языка.

Стереотип (stereotype) — это новый вид элемента модели, который обладает той же структурой, что и существующий элемент, но имеет дополнительные ограничения, другую интерпретацию и пиктограмму, а также по-другому обрабатывается генераторами кода и другими прикладными программами. Стереотип определяет набор теговых величин.

**Теговая** величина (tagged value) — это определяемый пользователем атрибут, который присваивается самим элементам модели, а не объектам в работающей системе. Они могут нести в себе, например, информацию об управлении проектом, указания по генерированию кода и сведения, касающиеся конкретной предметной области.

**Ограничение** (constraint) — это хорошо формализованное условие, выражаемое текстовой строкой на некотором языке ограничений (это может быть язык программирования, специализированный язык ограничений или естественный язык). В UML входит язык ограничений под названием ОСь.

**Профиль** (profile) — это набор стереотипов и ограничений с конкретным назначением, которые могут применяться к пользовательским пакетам. Профили можно разрабатывать в определенных целях, хранить в библиотеках и использовать для пользовательских моделей.

Как и к расширению возможностей любого другого языка, к расширению языка UML нужно подходить очень осторожно, так как можно легко создать новый диалект языка, непонятный для остальных. В случае же правильного использования механизмы расширения устраняют необходимость более радикальных изменений в языке.

### 3.3.4 Представления UML

В языке UML нет четких границ между различными концепциями и конструкциями, но для удобства их можно разделить на несколько представлений.- Представление модели — это просто подмножество конструкций, которое представляет один из аспектов моделируемой системы. Разделение моделей на представления делалось в некотором роде произвольно, но, как мы надеемся, обоснованно и наглядно. Концепции каждого из представлений моделей проиллюстрированы одной или двумя диаграммами. Представления, используемые в этой книге, не входят в спецификацию UML, но они служат вспомогательным средством для структурирования и пояснения концепций UML.

На самых высоких уровнях абстракции различают следующие группы представлений: структурная классификация, динамическое поведение, физическое размещение и управление моделью.

В таблице 4 перечислены представления UML и диаграммы, являющиеся графическим воплощением этих представлений, а также основные концепции, от носящиеся к каждому из представлений. Эту таблицу следует воспринимать не как жесткий набор правил, а скорее как общие указания по использованию языка, поскольку четких границ между представлениями нет.

Таблица 3. Представления модели и диаграммы в языке UML

Основная область	Представления	Диаграммы	Основные концепции
Структурная	Статическое представление	Диаграмма классов	Класс, ассоциация, обобщение, зависимость, реализация, интерфейс
	Представление проектирования	Внутренняя структура	Соединитель, интерфейс, часть, порт, обеспеченный интерфейс, роль, требуемый интерфейс
		Диаграмма кооперации	Соединитель, кооперация, использование кооперации, роль
		Диаграмма компонентов	Компонент, зависимость, порт, обеспеченный интерфейс, требуемый интерфейс, подсистема
	Представление Use Case	Диаграмма Use Case	Актер, ассоциация, расширение, включение, элемент Use Case, обобщение элемента Use Case
Динамическая	Представление конечных автоматов	Диаграмма автомата	Завершение перехода, осуществление деятельности, эффект, событие, область,

			состояние, переход, триггер
	Представление деятельности	Диаграмма деятельности	Действие, деятельность, поток управления, узел управления, поток данных, исключение, область расширения, разделение, слияние, объектный узел, контакт
	Представление взаимодействия	Диаграмма последовательности	Спецификация вхождения, спецификация исполнения, взаимодействие, фрагмент взаимодействия, операнд взаимодействия, линия жизни, сообщение, сигнал
		Диаграмма коммуникации	Кооперация, сторожевое условие, сообщение, роль, порядковый номер
Физическая	Представление развертывания	Диаграмма развертывания	Артефакт, зависимость, манифестация, узел
Управление моделью	Представление управления моделью	Диаграмма пакетов	Импорт, модель, пакет
	Профиль	Диаграмма пакетов	Ограничение, профиль, стереотип, теговая величина

**Структурная классификация** описывает системные сущности и их отношения между собой. Концепцию сущности в системе моделирует классификатор. В число классификаторов, имеющих в моделях UML, входят классы, элементы Use Case, актеры, узлы, кооперация и компоненты. Классификаторы являются базой, на которой строится динамическое поведение системы. К структурным представлениям относятся статическое представление, представление проектирования и представление Use Case.

**Динамическое поведение** описывает поведение системы или другого классификатора во времени. Поведение можно определить как ряд изменений в мгновенных снимках системы, полученных со статической точки зрения. Представления моделей динамического поведения включают в себя представление конечных автоматов, представление деятельности и представление взаимодействия.

**Физическое размещение** описывает вычислительные ресурсы системы и развертывание на них артефактов. Физическое размещение описывается представлением развертывания.

**Представление управления моделью** — это описание разбиения модели на иерархические блоки. Пакет — это базовая организационная единица для моделей. Модель представляет собой иерархию пакетов, обеспечивающую семантически полную абстракцию системы с определенной точки зрения. Представление управления моделью организует все остальные представления моделей в структуру, позволяющую осуществлять процесс разработки конфигурации и управления ею.

Расширения языка UML структурируются по профилям. В UML-профилях объявляется ряд конструкций, которые обеспечивают пусть ограниченную, но полезную функцию расширения возможностей языка UML. В число таких конструкций входят ограничения (constraints), стереотипы (stereotypes) и определения тегов (tag definitions). Профили объявляются на диаграммах классов, а применяются на диаграммах пакетов. Стереотипы применяются обычно на диаграммах классов, хотя могут возникать и в других местах. Профили могут также включать библиотеки классов, специфичных для данной предметной области.

### 3.3.5 Представление Use Case

Представление Use Case описывает поведение подсистем, классов или всей системы с точки зрения пользователя. При этом вся функциональность системы делится на транзакции с участием идеализированных пользователей системы, называемых *актерами* (actors). Элементы взаимодействия актера с системой называются *элементами Use Case* (в русскоязычной литературе бытуют также термины *прецедент* и *вариант использования*) Элемент Use Case описывает взаимодействие системы с одним или несколькими актерами, имеющее вид последовательности сообщений. В понятие *актер* входят люди, компьютерные системы и процессы. На рисунке 1 приведена несколько упрощенная диаграмма Use Case для программы продаж по телефонному каталогу.

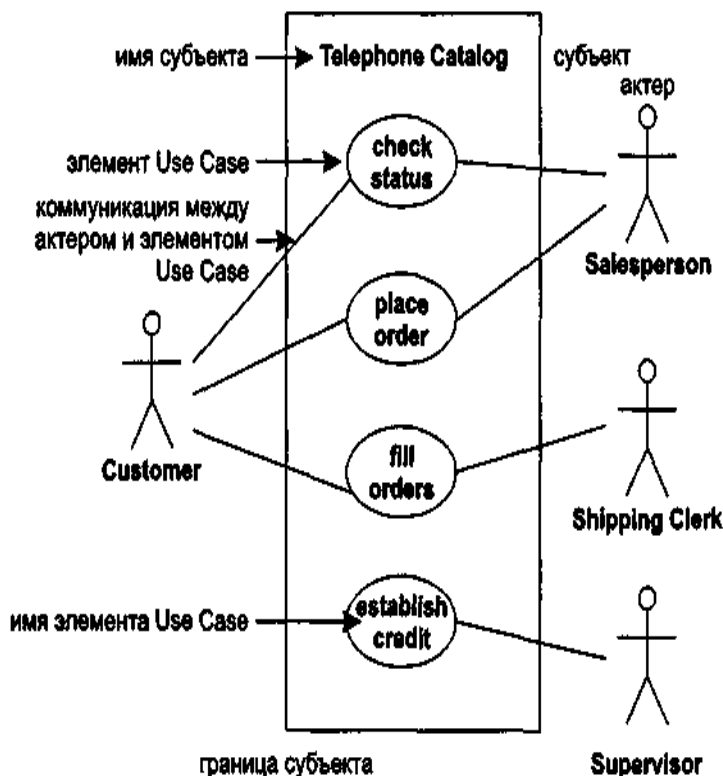


Рис. 1. Упрощенная диаграмма Use Case для программы продаж по телефонному каталогу

## **Актеры**

Актер (actor) — это идеализированное представление роли, которую играет внешняя сущность (например, человек или процесс), вступающая во взаимодействие с системой, подсистемой или классом. Актер определяет возможные виды взаимодействий между системой и некоторым классом ее пользователей. В реальном мире один физический пользователь может выполнять функции нескольких актеров системы. И наоборот, несколько пользователей могут соответствовать одному и тому же актеру; в таком случае пользователи являются экземплярами актера. Например, в разное время один и тот же человек может быть в магазине кассиром или покупателем.

Каждый актер может являться участником одного или нескольких элементов Use Case. Он взаимодействует с элементом Use Case (а следовательно, и с системой или классом, к которому относится данный элемент) посредством обмена сообщениями. При этом внутренняя реализация актера не имеет значения — достаточно описать атрибуты, которые определяют его состояние.

Актеров можно организовывать в иерархии обобщения, в которых описание абстрактного актера постепенно дополняется более конкретными чертами. Актер может быть человеком, компьютерной системой или некоторым исполняемым процессом.

На диаграммах актер изображается в виде «проволочного» человечка, под которым указано его имя.

## **Элементы Use Case**

Элементом Use Case называется целостный блок видимой извне функциональности, предоставляемой классификатором (который называется субъектом) и выраженной в виде последовательностей сообщений между субъектом и одним или несколькими актерами данного блока. Элемент Use Case описывает некоторый целостный фрагмент поведения системы, не вдаваясь при этом в особенности внутренней структуры субъекта. Определение элемента Use Case содержит все свойственные ему виды поведения: основные последовательности, различные варианты стандартного поведения и возможные при этом исключительные ситуации с указанием ответной реакции на них. С точки зрения пользователя, некоторые из видов поведения системы выглядят как ошибочные. Однако для системы ошибочная ситуация является одним из вариантов поведения, который должен быть описан и обработан.

В модели выполнение каждого элемента Use Case независимо от остальных, но для реализации элемента могут понадобиться объекты, которые задействуются другими элементами Use Case. Так возникают неявные зависимости между элементами Use Case. В действительности

каждый из них представляет собой самостоятельную часть функциональности системы, выполнение которой может производиться параллельно с другими такими частями.

Динамическое описание элемента Use Case осуществляется в языке UML с помощью диаграмм состояний, диаграмм последовательности, диаграмм коммуникации или неформальных текстовых описаний. Реализуются элементы Use Case в виде коопераций между классами системы. Один и тот же класс может участвовать в нескольких кооперациях, то есть, в нескольких элементах Use Case.

На системном уровне элементы Use Case представляют собой некоторые видимые для внешних пользователей фрагменты поведения системы. В этом смысле элемент Use Case подобен операции, которая может быть вызвана пользователем системы. Однако в отличие от системной операции, элемент Use Case в ходе своего выполнения может получать дополнительную информацию от своих актеров.


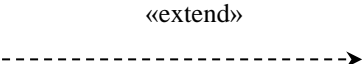

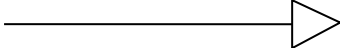
Рассмотрение в терминах элементов Use Case можно применить ко всей системе в целом и распространить на более мелкие ее части — например, подсистемы и отдельные классы. Внутренний элемент Use Case представляет собой поведение какой-либо части системы по отношению ко всей системе. Например, элемент Use Case класса — это целостный фрагмент функциональности, который данный класс предоставляет другим классам, играющим определенные роли в системе. Класс может иметь несколько элементов Use Case.

Элемент Use Case — это логическое описание определенной части функциональности системы. Он не является четкой конструкцией, которую можно напрямую реализовать в программном коде. Напротив, каждому элементу Use Case необходимо поставить в соответствие набор классов, на базе которого можно будет реализовать систему. Поведению, которое описывается элементом Use Case, ставятся в соответствие переходы и операции классов. Поскольку один и тот же класс может играть несколько ролей в реализации системы, он, соответственно, может участвовать в реализации нескольких элементов Use Case.

Одна из задач проектирования — найти для реализации элементов Use Case такие классы, которые удачно сочетали бы в себе нужные роли и не создавали при этом излишних сложностей. Реализацию элемента Use Case можно смоделировать в виде одной или нескольких коопераций. Кооперация — это реализация элемента Use Case. Помимо ассоциаций с актерами, элемент Use Case может иметь еще несколько видов отношений (таблица 4).



Таблица 4. Виды отношений вариантов использования

Отношение	Функция	Нотация
Ассоциация (Association)	Линия, по которой происходит обмен информацией между актером и элементом Use Case	
Расширить (Extend)	Включение добавочного поведения в исходный элемент Use Case «без ведома» последнего	
Включить (Include)	Включение добавочного поведения в исходный элемент Use Case, который явно описывает включение	
Обобщение элемента Use Case (Use case generalization)	Отношения между общим элементом Use Case и его более специфической разновидностью (второй наследует черты общего и добавляет к ним свои)	

На диаграммах элемент Use Case изображается в виде эллипса. Внутри эллипса или под ним указывается имя элемента. Сплошные линии соединяют элемент Use Case с его актерами.

Описание большого элемента Use Case можно разбить на более простые элементы. Это похоже на то, как описание класса можно вывести инкрементным путем из описания его предка. Элемент Use Case может включать в себя черты поведения других элементов Use Case. Такое отношение носит название *отношения включения* (include). Полученный этим путем элемент Use Case не является специализацией исходного и не может его заменять.

Элемент Use Case можно также определить как инкрементное расширение исходного элемента Use Case. Это называется *отношением расширения* (extend). У исходного элемента Use Case может быть несколько расширяющих вариантов, которые вносят дополнения в его

семантику. Все эти варианты могут применяться вместе. В таком случае создаваемый экземпляр элемента Use Case является экземпляром именно исходного элемента, а не его расширений.

Отношения включения и расширения изображаются на диаграммах в виде пунктирных стрелок. Над стрелками указывается ключевое для данного отношения слово («include» или «extend»). Стрелка «include» указывает на включаемый элемент Use Case, а стрелка «extend» — на расширяемый.

Элемент Use Case может иметь несколько потомков, любой из которых можно подставлять вместо родительского элемента Use Case. Этот механизм называется обобщением элементов Use Case.

Обобщение элементов Use Case изображается так же, как и любое другое обобщение, — стрелкой с наконечником в виде большого полого треугольника, идущей от потомка к родителю. На рисунке 2 показаны отношения элементов Use Case для программы, которая обрабатывает продажи по телефонному каталогу.

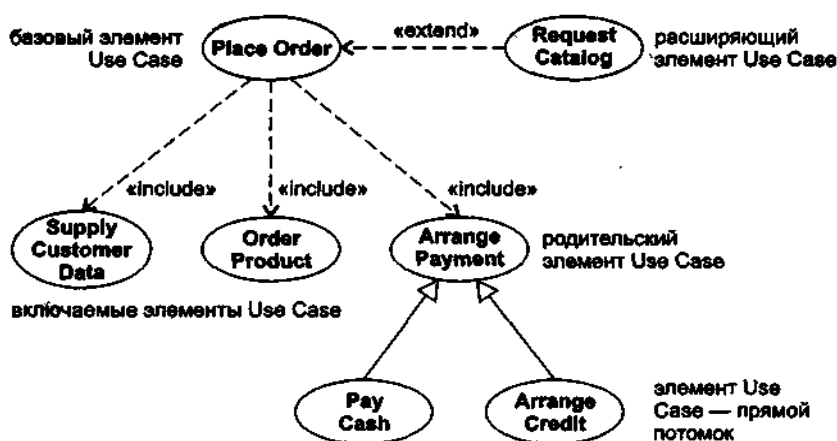


Рис. 2. Отношения элементов Use Case для программы, которая обрабатывает продажи по телефонному каталогу

### 3.4 Лабораторная работа №4

Тема: Построение объектной модели.

Задание: Для предметной области, изученной в лабораторных работах № 1-3, построить объектную модель.

Теоретический материал

#### 3.4.1 Объектная модель

Объектно-ориентированная технология основывается на так называемой *объектной модели*.

Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования.

На объектную модель влияет ранняя модель жизненного цикла программного обеспечения. Традиционная техника анализа основана либо на потоках данных в системе (анализ отношений), либо на процессах протекающих в системе (структурный анализ). Объектно-ориентированный анализ (или *ООА*) направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

***Объектно-ориентированный анализ — это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.***

#### 3.4.2 Классы и объекты

Объект можно неформально определить как осязаемую реальность, проявляющую четко выделяемое поведение. С точки зрения восприятия человеком объектом может быть:

- \* осязаемый и (или) видимый предмет;
- \* нечто, воспринимаемое мышлением;
- \* нечто, на что направлена мысль или действие.

Таким образом, мы расширили неформальное определение объекта новой идеей: **объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве.** Термин *объект* в программном обеспечении впервые был введен в языке Simula и применялся для моделирования реальности.

Объектами реального мира не исчерпываются типы объектов, интересные при

проектировании программных систем. Другие важные типы объектов вводятся на этапе проектирования, и их взаимодействие друг с другом служит механизмом отображения поведения более высокого уровня. Это приводит нас к более четкому определению: «Объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области». В еще более общем плане объект может быть определен как нечто, имеющее четко очерченные границы.

Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операции и проявляет хорошо определенное поведение.

Объекты могут быть осязаемыми, но иметь размытые физические границы: реки, туман или толпы людей. Подобно тому, как взявший в руки молоток начинает видеть во всем окружающем только гвозди, проектировщик с объектно-ориентированным мышлением начинает воспринимать весь мир в виде объектов. Разумеется, такой взгляд несколько упрощен, так как существуют понятия, явно не являющиеся объектами. К их числу относятся атрибуты, такие, как время, красота, цвет, эмоции. Однако, потенциально все перечисленное - это свойства, присущие объектам.

Полезно понимать, что объект — это нечто, имеющее четко определенные границы, но этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции. Можно дать следующее определение:

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

***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. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Мы говорим «как

правило», потому что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового автомата описывается в терминах других объектов, например, имеющихся в наличии напитков. Конкретные напитки — это самостоятельные объекты, отличные от торгового автомата (их можно пить, а автомат нет, и совершать с ними иные действия).

Таким образом, мы установили различие между объектами и простыми величинами: простые количественные характеристики (например, число 3) являются «постоянными, неизменными и непреходящими», тогда как объекты существуют во времени, изменяются, имеют внутреннее состояние, преходящи и могут создаваться, уничтожаться и разделяться.

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

**Что такое 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.

### **Что такое класс?**

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. Таким образом, можно говорить о классе «Млекопитающие», который включает характеристики, общие для всех млекопитающих. Для указания на конкретного представителя млекопитающих необходимо сказать «это — млекопитающее» или «то — млекопитающее».

В общепонятных терминах можно дать следующее определение класса: группа,

множество или вид с общими свойствами или общим свойством, разновидностями, отличиями по качеству, возможностями или условиями. В контексте объектно-ориентированного анализа дадим следующее определение класса:

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

Любой конкретный объект является просто экземпляром класса. Что же не является классом? Объект не является классом, хотя в дальнейшем мы увидим, что класс может быть объектом. Объекты, не связанные общностью структуры и поведения, нельзя объединить в класс, так как по определению они не связаны между собой ничем, кроме того, что все они объекты.

Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но не достаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, на достаточно высоком уровне абстракции графический интерфейс пользователя, база данных или система учета как целое, это явные объекты, но не классы. Лучше считать их некими совокупностями (кластерами) сотрудничающих классов.

### **3.4.3 Роль классов и объектов в анализе и проектировании**

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 абстрациях* задачи (совокупность классов и объектов), во втором — о *механизмах* реализации (совокупность структур).

На ранних стадиях внимание проектировщика сосредотачивается на внешних проявлениях ключевых абстракций и механизмов. Такой подход создает логический каркас системы: структуры классов и объектов. На последующих фазах проект, включая реализацию, внимание переключается на внутреннее поведение ключевых абстракций и механизмов, а также их физическое представление. Принимаемые в процессе проектирования решения задают архитектуру системы: и архитектуру процессов, и архитектуру модулей.

### **3.4.4 Объектно-ориентированный анализ**

Границы между стадиями анализа и проектирования размыты, но решаемые ими задачи

определяются достаточно четко. В процессе анализа мы моделируем проблему, *обнаруживая* классы и объекты, которые составляют словарь проблемной области. При объектно-ориентированном проектировании мы *изобретаем* абстракции и механизмы, обеспечивающие поведение, требуемое моделью.

Рассмотрим основные пункты этого этапа:

- **Выделить важнейшие компоненты системы и создать их классы**
- **Определить характерные признаки каждого класса (включая их названия)**
- **Связать классы между собой:**
  - **Найти отношения между классами**
  - **Использовать свойства и методы классов для моделирования бизнес - процесса**
  - **Использовать свойства и методы классов для моделирования правил предприятия**
  - **Использовать свойства и методы классов для моделирования проблемных ситуаций**
- **Составить "эскиз" системы предприятия для подтверждения ее структуры и дальнейших уточнений**
- **Составить окончательный "чертеж" для формирования спецификаций**
- **Разработать спецификации**

Анализ системы служит для обеспечения эффективной связи с конечным пользователем. Детали реализации, понятные только программистам и способные смутить непосвященного пользователя, остаются при этом в стороне. Это помогает и самому аналитику понять, что же действительно важно для пользователя. Аналитик, при анализе системы сосредоточившись на нуждах клиента (а не системы), в будущем, при разработке, станет реализовывать только нужные возможности, не увлекаясь чрезмерным усложнением программы.

Результаты этапа разработки, хотя и являются конкретной реализацией задачи, не предназначены для конечного пользователя. Они используются для составления технической документации и при дальнейшем написании текста программы.

Способ представления модели играет важную роль. Взаимосвязь аналитика и пользователя основана именно на модели и позволяет им уточнять и поправлять модель, добиваясь ее точного соответствия с системой предприятия. Поэтому модель нужно представлять в формате, облегчающем общение.

Для этих целей был приспособлен объектно-ориентированный метод. Символы "языка эскизов" позволяют кратко и точно моделировать систему. При анализе систем очень полезно применять диаграммы, которые помогают вам и пользователю лучше понять друг друга. Диаграммы можно представлять в произвольном формате.

Эскизы модели удобно строить с помощью какого-нибудь графического пакета. Но применение программного обеспечения не обязательно, все нужные заметки можно делать и в записной книжке. Важно помнить только одно - на данной стадии рассматривается концепция проекта, а не его воплощение.

### **3.4.5 Выполнение объектно-ориентированного анализа**

Зная терминологию, можно поэтапно выполнить системный анализ для нашего проекта с фирмой торгующей пищевыми продуктами. В этом разделе показано выполнение различных шагов системного анализа, которые приведут к полной картине системы, допускающей воплощение на конкретном компьютере.

Рассмотрим работу аналитика над проектом. Перед началом работы он должен составить для себя список основных задач:

- Составить список всех абстрактных существительных, применяемых для описания системы
- Повторно рассмотреть составленный список, выделив в нем возможные классы
- Там, где это возможно, выделить иерархию классов
- Перечислить свойства и методы каждого класса
- Объединяя классы, составить эскиз системы
- Встретиться с руководством фирмы, уточнить и пополнить информацию
- Дополнить полученной информацией свойства и методы классов
- Разработать окончательную модель системы

Выполняя приведенные рекомендации, можно провести объектно-ориентированный анализ и построить модель.



### **3.5 Лабораторная работа №5**

Тема: Разработка спецификаций.

Цели и задачи работы: изучение видов спецификаций, построение модели верхнего уровня будущей программы.

Задание: Для объектной модели, построенной в лабораторной работе № 4, разработать спецификации будущей программы.

Теоретические сведения

#### **3.4.1 Спецификации и проектирование**

На этом этапе проектирования программы нам необходимо оформить информацию полученную на предыдущем этапе в таком виде, чтобы ее мог понять любой другой человек и чтобы он мог, используя эту информацию, реализовать следующий этап проектирования. Так как над большим проектом работает обычно группа людей, информация должна быть представлена таким образом, чтобы она не допускала двоякого толкования и не было необходимости привлекать человека, который ее создавал для “расшифровки” созданного описания.

#### **3.4.2 Процедурная абстракция**

Процедура выполняет преобразование входных аргументов в выходные. Более того, это есть отображение набора значений входных аргументов в выходной набор результатов с возможной модификацией входных значений. Набор входных или выходных значений или оба этих набора могут быть пусты.

##### *3.4.2.1 Преимущества абстракции*

Абстракция представляет собой некоторый способ отображения. При этом мы “абстрагируемся” от несущественных подробностей, описывая лишь те, которые имеют непосредственное отношение к решаемой задаче. Реализации абстракций должны быть согласованы по всем таким “существенным” подробностям, а в остальном могут отличаться.

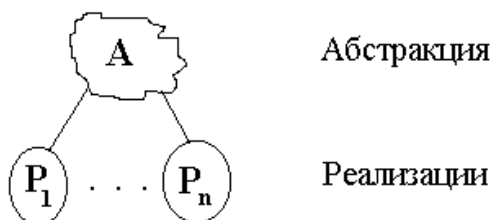
В абстракции через **параметризацию** мы абстрагируемся от конкретных используемых данных. Эта абстракция определяется в терминах формальных параметров. Фактические данные связываются с этими параметрами в момент использования такой абстракции. Значения конкретных используемых данных являются несущественными; важно лишь их количество и типы. Параметризация позволяет обобщить модули, делая их полезными в большом числе ситуаций. Преимущество таких обобщений заключается в том, что они уменьшают объем программы и, следовательно, объем модификаций.

В абстракции через спецификацию мы фокусируем внимание на особенностях, от которых зависит пользователь, и абстрагируемся от подробностей реализации этих особенностей.

Существенным является “поведение”, то, что делается, а несущественным то, “как” это делается. *Например*, в процедуре сортировки массива существенным является факт сортировки массива, а не сам алгоритм сортировки.

Главное преимущество абстракции через спецификацию заключается в несущественности способа реализации, что позволяет нам переходить к другой реализации без внесения изменений в программу, использующую данную реализацию.

Реализации могут быть даже выполнены на различных языках программирования при том условии, что типы данных аргументов в этих языках трактуются одинаково.



Абстракция через спецификацию наделяет структуру программы двумя отличительными особенностями.

Первая из этих особенностей заключается в локальности, которая означает, что реализация одной абстракции может быть создана или рассмотрена без необходимости анализа реализации какой-либо другой абстракции. Для написания программы, использующей абстракции, программисту достаточно понимать только ее поведение, а не подробности ее реализации.

Как само составление программы, так и анализ уже созданной абстракции облегчает так называемый принцип локальности. Этот принцип также позволяет составлять программу из абстракций, создаваемых людьми, работающими независимо друг от друга. Один человек может создать абстракцию, которая использует абстракцию, созданную кем-то другим. После достижения договоренности об общих задачах, выполняемых программой, группа людей может работать над отдельными абстракциями независимо друг от друга, получая при этом корректно выполняющиеся программы.

Кроме того, для понимания работы программы, достаточно понимания функций только одной абстракции.

Для понимания программы, реализующей одну из абстракций, необходимо знать, что реализуют собой сами используемые абстракции, а не конкретные операторы их тела.

В большой программе объем “несущественной” информации может быть весьма велик, и мы можем игнорировать не только внутренний текст используемых в ней абстракций, но и коды абстракций, которые они сами используют, и так далее.

**Второй** особенностью является модифицируемость. Абстракция через спецификацию позволяет упростить модификацию программы. Если реализация абстракции изменяется, но ее спецификация при этом остается прежней, то эти изменения не повлияют на оставшуюся часть программы. Разумеется, если число подлежащих изменению абстракций достаточно велико, то на это по-прежнему тратится много времени.

Объем работ может быть значительно сокращен путем выделения потенциальных модификаций, уже на начальном этапе разработки программы и последующей попыткой ограничения их небольшим числом абстракций.

**Например**, эффект машинной зависимости может быть ограничен несколькими модулями, что обеспечивает простоту в переносимости программ на другую вычислительную машину.

Модифицируемость существенно повышает эффективность программы. Программисты обычно плохо прогнозируют фактическое время, необходимое на разработку сложной системы, вероятно, по той причине, что предугадать все возможные трудности заранее невозможно.

По причине неразумности изобретения структур, избегающих несуществующих трудностей, рекомендуется начать с простого набора абстракций, апробировать систему, выясняя все узкие места, а затем модифицировать соответствующие абстракции.

#### *3.4.2.2 Спецификации*

Очень важно дать абстракциям четкие определения. В противном случае не могут быть достигнуты преимущества, рассмотренные выше.

Например, мы можем заменить одну реализацию абстракции на другую только в том случае, если все то, что поддерживала старая реализация, поддерживает и новая.

Элементом, создающим данную зависимость, является абстракция. Следовательно, мы должны знать, что она собой представляет.

Мы будем определять абстракции посредством спецификаций, которые создаются на языке спецификаций, причем последний может быть как формальным, так и неформальным. Преимущество формальных спецификаций заключается в том, что они имеют точно определенное значение. Неформальные спецификации легче читать и понимать, однако точное их содержание установить затруднительно, поскольку язык неформальных спецификаций точным не является. Несмотря на это, неформальные спецификации могут быть весьма информативны и составлены таким образом, что читатели без труда поймут их смысловое содержание.

Спецификация отлична от любой определяемой ей реализации абстракции. Все реализации сходны между собой, поскольку они реализуют одну и ту же абстракцию, Отличие их заключается в том, что это делается разными способами. Спецификация определяет их схожесть.

#### *3.4.2.3 Спецификация процедурных абстракций*

Спецификация процедуры состоит из заголовка и описания функции, выполняемой процедурой.

**Заголовок** содержит имя процедуры, номер, порядок и типы входных и выходных параметров. Кроме этого, выходные параметры могут, а входные должны быть поименованы.

**Пример:**

```
remove_dups=proc (a:array[int])
```

```
sqrt=proc (x:real) returns (rt:real)
```

В общем виде

```
<имя>=proc ([<вх.пар>:<тип>[,<вх.пар>:<тип>...]])
```

```
returns ([<вых.пар>:<тип>[,<вых.пар>:<тип>...]])
```

Информация в заголовке описывает “форму” процедуры. Это аналогично описанию “формы” математической функции:

**$f : \text{integer} \rightarrow \text{integer}$**

Ни в одном из случаев смысл действий, выполняемых процедурой или функцией, не описывается. Эта информация приводится в семантической части спецификации, в которой смысл выполняемых процедурой действий описывается на естественном (разговорном) языке, возможно расширенном привычными математическими обозначениями. В этом описании используются имена входных и выходных параметров.

Семантическая часть спецификации состоит из трех предложений:

- **requires** (требует),
- **modifies** (модифицирует),
- **effects** (эффекты).

Эти предложения должны появляться в указанном ниже порядке, однако предложения **requires** и **modifies** обязательными не являются.

Шаблон спецификации для процедурных абстракций.

Pname = proc (...) returns (...)

requires //этот оператор задает необходимые требования

modifies //этот оператор идентифицирует все модифицируемые входные данные

effects //этот оператор описывает выполняемые функции.

Предложение **requires** задает ограничения, накладываемые на абстракцию. Предложение **requires** необходимо в том случае, если процедура является **частичной**, т.е. ее поведение для некоторых входных значений недетерминировано.

Если же процедура **глобальна**, т.е. ее поведение определено для всех входных значений, то предложение **requires** может быть опущено. В этом случае единственными требованиями предъявляемые при обращении к процедуре, являются требования, указанные в заголовке, т.е. число и типы аргументов.

Оператор **modifies** задает список имен входных параметров, модифицируемых процедурой. Если входные параметры не указаны или не модифицируются, то это предложение может быть опущено.

Предложение **effects** описывает работу процедуры со значениями, не охваченные предложением **requires**. Оно определяет выходные значения и модификации, производимые над входными параметрами, перечисленными в списке **modifies**. Предложение **effects** составляется исходя из предложения, что требования предложения **requires** удовлетворены. Однако в том случае, когда требования в предложении **requires** не удовлетворены, о поведении процедуры ничего не сообщается.

**Например:**

concat=proc (a,b:string) returns (ab: string)

effects по возврату ab есть новая строка, содержащая символы из a (в том порядке, в котором они расположены в a), за которыми следуют символы из b (в том порядке, в котором они расположены в b).

Remove\_dupls=proc (a:array [int])

modifies a

effects удаляет из массива *a* все повторяющиеся элементы. Нижняя граница *a* остается без изменений, однако порядок следования оставшихся элементов может изменяться, если, например, перед вызовом *a*=[1:3, 13,3,6], то по возвращению массив *a* имеет нижнюю границу, равную 1, и содержит три элемента 3,13 и 6, расположенных в некоторой неопределенной последовательности.

Search=proc (a:array[int],x:int) returns (i:int)

requires массив *a* упорядочен по возрастанию.

effects Если элемент *x* принадлежит массиву *a*, то возвращается значение *i* такое, что *a*[*i*]=*x*; в противном случае значение *i* на единицу больше, чем значение верхней границы массива *a*.

#### 3.4.2.4 Реализация процедур

Реализация процедуры должна выполнять действия, определенные в спецификации. В частности, она должна модифицировать только те входные параметры, которые указаны в предложении *modifies*, а условия в предложении *requires* должны выполняться для всех входных значений. Выходные значения должны соответствовать требованиям, указанным в предложении *effects*.

Каждый язык программирования имеет свой механизм реализации процедурных абстракций.

При реализации абстракции желательно придерживаться следующих соглашений. Во-первых, использовать для формальных параметров те же имена, что и в спецификации. Это соглашение помогает связать реализацию абстракции со спецификацией. За заголовком помещаем комментарий, поясняющий работу алгоритма. Придерживаться общепринятых правил форматирования.

#### 3.4.2.5 Создание процедурных абстракций

Процедуры, как и другие виды абстракций, в процессе своего создания необходимо минимизировать. В них должны быть реализованы только необходимые функции. Это предоставляет разработчику большую свободу, позволяя ему создавать более эффективную версию.

К одному из пунктов, который обычно остается неопределенным относится сам метод, используемый в конкретной реализации. Обычно пользователям предоставляется свобода в его выборе. (Впрочем, имеются исключения: например, процедура, работающая с числами, может быть ограничена хорошо известным числовым методом с тем, чтобы при округлении ее работа приводила к известным, четко определяемым погрешностям). Также

могут быть оставлены неопределенными некоторые выполняемые процедурой функции. В такой ситуации процедура становится **недоопределенной**. Это означает, что для определенных значений входных параметров на выходе вместо единственного правильного результата имеется набор допустимых результатов. Реализация может ограничивать этот набор только одним значением, однако он может быть любым из числа допустимых.

Процедура search является недоопределенной, поскольку мы не указываем точно, какой индекс должен быть возвращен в том случае, если значение x встречается в массиве несколько раз. Рассмотрим реализации абстракции search.

```
#include <iostream.h>
#include <stdlib.h>

int search1( int a[20], int x, int n)
{
    for ( int i=0; i<n; i++ )
        if ( a[i]==x ) return i;
    return n;
}

int search2 ( int a[20], int x, int n )
{
    int fl=1;
    int aa=0, bb=n, cc=int((aa+bb)/2);

    while (fl)
    {
        if (a[cc]==x) return cc;
        if (a[cc]<x) aa=cc;
        else bb=cc;
        if ( abs(aa-bb)==1) fl=0;
        else cc=int((aa+bb)/2);
    }
    return n;
}

void main()
{
    int a[20]={1,2,7,3,3,4,5,6};
    int n=8;
    int r1,r2;
    int x=7;

    r1=search1(a,x,n);
    r2=search2(a,x,n);

    count << r1 << r2 ;
}
```

Неопределенная абстракция может иметь детерминированную реализацию, т.е. такую, которая, будучи вызванной два раза с идентичными входными данными, выполняется одинаково.

Помимо требований минимизации другим важным свойством процедур является **обобщаемость**, которая достигается путем использования параметров вместо переменных.

Например, процедура, работающая с массивами произвольного размера, является более обобщенной, чем та, которая работает с массивами фиксированного размера. Аналогично процедура, работающая с массивами элементов произвольного типа, является более обобщенной, чем процедура, требующая, чтобы все элементы массива были целыми числами.

Однако обобщение процедуры полезно в том случае, если эффективность ее применения повышается. Это почти всегда так, когда речь идет о независимости размеров, поскольку при этом небольшие изменения контекста применения процедуры (например, удваивание размера массива) требуют небольших изменений в программе.

Другой важной характеристикой процедур является **простота**. Процедура должна обладать хорошо определенным и легко объяснимым назначением, независимым от контекста ее использования. Хорошим правилом может служить присваивание процедуре имени, которое описывает ее назначение. Отсутствие такого дополнительного пояснения может породить сложности.

Наконец, процедура должна действительно выполнять некоторые функции. Процедуры создаются в процессе написания программы и служат цели упрощения и облегчения работы с ней, а также создания более ясной структуры программы. При этом программа становится более легко понимаемой. Однако существует опасность введения слишком большого числа процедур.

Частичные процедуры не так безопасны как общие, поскольку они требуют от пользователя выполнения требований, заданных в предложении `requires`. Если эти требования не удовлетворены, то поведение процедуры становится неопределенным, что может привести к неверной работе программы. Например, при задании для процедуры `search` неупорядоченного массива она может вернуть неверный индекс. Эта ошибка может оставаться необнаруженной долгое время после возврата из процедуры `search`. Вследствие этого причина ошибки будет неясна, а объекты данных могут быть разрушены.

С другой стороны, частичные процедуры могут оказаться более эффективными в реализации, чем общие.



При выборе между частичной и общей процедурами мы должны придерживаться определенных соглашений. С одной стороны критерием должна являться эффективность. С другой - корректное выполнение с меньшим числом потенциальных ошибок.

Одним из важных факторов является ожидаемая область применения. Если процедура создается для общего пользования (например, доступна как часть библиотеки программы), соображения безопасности играют существенную роль. В такой ситуации невозможно осуществить статистический анализ всех обращений к процедуре, позволяющий удостовериться в том, что необходимые требования удовлетворены. Следовательно, в этом случае полагаться на такой анализ неразумно.

Другой случай предполагает использование некоторых процедур в ограниченном контексте. В ограниченном контексте легко обеспечить выполнение необходимых требований. Следовательно, мы имеем безопасное поведение, не жертвуя при этом эффективностью.

Наконец, необходимо наличие четкой и понятной спецификации.

### **3.4.3 Абстракция данных**

Процедуры дают нам возможность добавлять в базовый язык новые операции. Кроме операций, однако, базовый уровень предусматривает различные типы данных, например, целые, вещественные, логические, строки и массивы. Нам необходимо иметь возможность добавлять в базовый уровень не только новые процедуры, но и типы данных. Эта необходимость удовлетворяется абстракцией данных.

Какие новые типы данных необходимы, зависит от области применения программы.

В каждом конкретном случае абстракция данных состоит из набора объектов и набора операций.

Например: матричные операции включают в себя сложение, умножение и т.д., а операции над счетами снятия и вклады.

Новые типы данных должны включать в себя абстракции как через параметризацию, так и через спецификацию.

Абстракции через параметризацию могут быть осуществлены точно также, как и для процедур, - использованием параметров там, где это имеет смысл.

Абстракции через спецификацию достигаются за счет того, что мы представляем операции как часть типа.

Почему операции необходимы, посмотрим, что получится, если будем рассматривать тип просто как набор объектов. В этом случае все, что необходимо сделать для реализации

типа - это выбрать представление памяти для объектов. Тогда все программы могут быть реализованы в терминах этого представления.

Однако, если представление изменится (или даже изменится интерпретация этого представления), все программы, которые используют этот тип, должны быть изменяемы.

С другой стороны, предположим, что мы включаем в тип следующим образом:

**абстракция данных = <объекты, операции>.**

При этом мы требуем, чтобы пользователи употребляли эти операции непосредственно, не обращаясь к представлению. Затем для реализации типа мы реализуем операции в терминах выбранного представления. При изменении представления мы должны заново реализовать операции. Однако переделывать программы уже нет необходимости, так как они не зависят от представления, а зависят только от операций. Следовательно, мы получаем абстракцию через спецификацию.

Если обеспечено большое количество операций, отсутствие доступа к представлению не будет создавать для пользователей никаких трудностей - все, что они хотят сделать с объектами, может быть сделано (и при том эффективно) при помощи использования операций.

Обычно имеются операции для создания и модификации объектов, а также для получения информации об их значениях. Конечно, пользователи могут увеличивать набор операций определением процедур, но такие процедуры не должны использовать представление.

Абстракция данных - наиболее важный метод в проектировании программ. Выбор правильных структур данных играет решающую роль для создания эффективной программы. При отсутствии абстракций данных структуры данных должны быть определены до начала реализации модулей. Но в этот момент, однако, структуры данных еще не вполне ясны. Следовательно, в выбранных структурах может отсутствовать какая-либо необходимая информация или же эти структуры могут быть организованы не эффективно.

Абстракции данных позволяет отложить окончательный выбор структур данных до момента, когда эти структуры станут нам вполне ясны. Вместо того, чтобы определять структуру непосредственно, мы вводим абстрактный тип со своими объектами и операциями. Затем можно осуществлять реализацию модулей в терминах этого абстрактного типа. Решение же относительно реализации типа может быть принято позже, когда использование данных станет для нас вполне понятным.

Абстракции данных также полезны при модификации и эксплуатации программ. Чаще предпочтительнее изменить структуры данных, чем улучшать производительность или

приспосабливаться к изменяющимся требованиям. Абстракции данных сводят все изменения к изменениям в реализации типа - модули же нет необходимости изменять.

#### 3.4.3.1 Спецификации для абстракций данных

Точно также, как и для процедур, значение типа не должно задаваться никакой его реализацией. Вместо этого должна иметься определяющая спецификация. Так как объекты типа используются только вызовом операций, основная часть спецификации посвящена описанию того, что эти операции делают. Спецификация состоит из заголовка, определяющего имя типа и имени его операций, и двух главных секций: секции описания и секции операций.

```
Dname = data type is // список операций
Descriptions
// Здесь приводится описание абстракций данных
Operations
//Здесь задаются спецификации для всех операций.
End dname
```

В секции описания тип описывается как целое. Иногда там дается модель для объектов, т.е. объекты описываются в терминах других объектов - таких, которые по предположению понятны тем, для кого эта спецификация предназначена. В секции описания должно также говориться, изменяемый или неизменяемый этот тип.

В секции операций содержатся спецификации для всех операций. Если операция - процедура, то ее спецификация будет процедурной спецификацией. Операция также может быть абстракцией через итерацию. В этих спецификациях могут использоваться концепции, введенные в секции описания.

#### Пример:

Спецификация абстракции данных intset.

Наборы целых чисел intset - это неограниченные множества целых чисел с операциями создания нового, пустого набора, проверки данного целого числа на принадлежность данному набору intset и добавления или удаления элементов. В секции описания, описываются наборы целых чисел в терминах математических наборов. Отмечаем, что наборы целых чисел - изменяемые, и перечисляем все изменяемые операции.

$S_{post}$  - значение S при возврате.

```
Intset= data type is create, insert, delete, member, size,
choose
```

## Descriptions

Наборы целых чисел `intset` - это неограниченных математических множества целых чисел. Наборы целых чисел изменяемые: операции `insert` и `delete` добавляют и удаляют целые числа из набора.

## Operations

`create = proc() returns (intset)`

**effects** Возвращает новый, пустой набор `intset`

`insert = proc (s:intset, x:intset)`

**modifies** `s`

**effects** Добавляет `x` к элементам `s`; после добавления - возврат,

$S_{post} = S \cup \{x\}$ , где  $S_{post}$  - это набор значений в `s` при возврате из `insert`.

`Delete = proc (s: intset, x: int)`

**modifies** `s`

**effects** Удаляет `x` из `s` (т.е.  $S_{post} = S - \{x\}$ ).

`Member = proc (s:intset, x: int) returns (bool)`

**effects** Возвращает значение `true`, если  $x \in S$

`size = proc (s:intset) returns (int)`

**effects** Возвращает число элементов в `s`

`choose = proc (s: intset) returns (int)`

**requires** набор `s` не пуст

**effects** возвращает произвольный элемент `s`.

**End intset**

В общем случае неформальное описание - слабое место неформальных спецификаций. Однако не все типы могут быть хорошо описаны в терминах математических концепций. Если концепции неадекватны, мы должны описать тип настолько хорошо, насколько возможно, даже с использованием рисунков, но тем не менее все равно остается опасность, что читатель не поймет описания или истолкует в другом, чем мы имели в виду смысле.

### Пример 2:

Полиномы `poly` - это неизменяемые полиномы с целыми коэффициентами. Предоставляются операции для создания одночленного полинома, для сложения, вычитания и умножения полиномов, а также для проверки двух полиномов на равенство.

Poly=**data type is** create, degree, coeff, add, mul, sub, minus, equal

### **Descriptions**

Полиномы poly - это неизменяемые полиномы с целыми коэффициентами.

Operations

create = **proc**(c,n: int) **returns** (poly)

**requires**  $n \geq 0$

**effects** возвращает полином cx. Например:

poly::create(6,3)= $6x^3$

poly::create(3,0)=3

poly::create(0,0)=0

degree = **proc**(p:poly) **returns** (int)

**effects** Возвращает степень P, т.е. наибольшую степень при ненулевом коэффициенте. Степень нулевого полинома равна 0.

Например:

poly::degree( $x^2+1$ )=2

poly::degree(17)=0

coeff = **proc**(p:poly,n:int) **returns** (int)

**requires**  $n \geq 0$

**effects** Возвращает коэффициент члена P со степенью n.

Возвращает 0, если n больше степени P.

Например:

poly::coeff( $x^3+2x+1$ ,4)=0

poly::coeff( $x^3+2x+1$ ,1)=2

add=**proc**(p,q:poly) **returns** (poly)

**effects** Возвращает полином, являющийся суммой полиномов p и q.

Mul=**proc**(p,q:poly) **returns** (poly)

**effects** Возвращает полином, являющийся произведением полиномов p и q.

Sub=**proc**(p,q:poly) **returns** (poly)

**effects** Возвращает полином, являющийся разностью полиномов p и q.

Minus=**proc**(p:poly) **returns** (poly)

effects Возвращает полином, являющийся разностью полиномов  $z$  и  $p$ , где  $z$  - нулевой полином.

```
Equal=proc (p,q:poly) returns (bool)
```

effects Возвращает значение true, если  $p$  и  $q$  имеют одинаковые коэффициенты при соответствующих членах, и значение false - в противном случае.

```
End poly.
```

#### *3.4.3.2 Реализация абстракций данных*

Для реализации типа данных мы выбираем представление для объектов и реализуем операции в терминах этого представления. Выбранное представление должно предоставлять возможности для довольно простой и эффективной реализации всех операций. Кроме того, если некоторые операции должны выполняться быстро, представление должно предоставлять и эту возможность. Часто представление, обеспечивающее быструю работу некоторых операций, приводит к тому, что другие операции выполняются медленно. В этом случае мы должны использовать несколько различных реализаций одного и того же типа.

Например, возможное представление для объекта `intset` - это массив целых чисел, где каждое целое число набора `intset` соответствует элементу массива. Мы должны решить - должен ли каждый элемент набора встречаться в массиве только один раз или же он может встречаться много раз. В последнем случае операция `insert` будет работать быстрее, однако операции `delete` и `member` будут выполняться медленнее. Если операция `member` используется часто, мы должны остановиться на первом случае.

Заметим, что здесь мы говорим о двух разных типах: новом абстрактном типе `intset`, который мы реализуем, и массиве целых, который используется как представление.

Каждая реализация будет иметь два таких типа: абстрактный тип и тип представления. Предполагается, что с типом представления мы имеем дело только при реализации. Все, что мы можем делать с объектами абстрактного типа, - это применять к ним соответствующие данному типу операции.

#### *3.4.3.3 Использование абстракций данных*

Если абстракция данных определена, то она отличается от встроенных типов и ее объекты и операции должны использоваться точно так же, как объекты и операции встроенных типов. Насколько это возможно, зависит от используемого языка программирования. Язык C++ предоставляет такие возможности.

Определенные пользователем типы могут также использоваться для представления других определяемых пользователем типов.

#### 3.4.3.4 Параметризованные абстракции данных

Типы - выгодные параметры для типов, точно также, как и для процедур. Например: рассмотрим абстракцию общего набора `set`, в котором элементы набора могут быть произвольного типа. Конечно, не все типы могут быть имеющими смысл параметрами общего набора, т.к. общие наборы `set` не хранят дублирующих друг друга элементов, должен быть некий способ определить дублируют элементы друг друга или нет. Следовательно, спецификация `set` требует, чтобы элементы типа имели операцию `equal`. Эти требования помещены сразу после заголовка. Требования на индивидуальные операции также разрешены.

```
Set = data type [t:type] is set, insert, ~set, member, size,  
choose, del
```

```
requires t имеет операцию equal : proctype (t,t) returns  
(bool), т.е. условие равенства t.
```

##### Description

Общие наборы `set` - неограниченные математические наборы. Эти наборы изменяемые: операции `insert` и `del` добавляют и уничтожают элементы набора.

##### Operations

```
set = proc() returns (set[t])
```

```
effects Возвращает новый пустой набор
```

```
insert = proc(s:set[t],x:t)
```

```
modifies s
```

```
effects Добавляет x к элементам s; после insert Возвращается
```

$$S_{post} = S \cup \{x\}$$

```
del = proc(s:set[t],x:t)
```

```
modifies s
```

```
effects Удаляет x из S (т.е.  $S_{post} = S - \{x\}$ ).
```

```
Member = proc(s:set[t],x:t) returns (bool)
```

```
effects Возвращает  $x \in s$ .
```

```
Size = proc(s:set[t]) returns (int)
```

```
effects Возвращает число элементов в S.
```

Choose = **proc**(s:set[t]) **returns** (t)

**requires** набор **S** не пуст

**effects** Возвращает произвольный элемент **S**.

#### 3.4.3.5 Изменяемость

Абстракции данных либо изменяемы (с объектами, значения которых могут изменяться), либо неизменяемы. К определению этого аспекта типа следует относиться внимательно. В общем случае тип должен быть неизменяемым, если его объекты естественным образом имеют неизменяющиеся значения.

Как правило, тип должен быть изменяемым, если моделируется что-то из реального мира, где значения объектов естественным образом изменяются с течением времени.

Однако мы можем все равно предпочесть использовать в этом случае неизменяемый тип, потому что он обеспечивает большую надежность.

Принимая решение об изменяемости иногда необходимо учесть соотношение эффективности и надежности.

Неизменяемые абстракции надежнее, чем изменяемые, т.к. не возникает никаких проблем при разделении объектов. Однако для неизменяемых абстракций объекты могут создаваться и уничтожаться чаще, что означает, что чаще будет необходима сборка мусора.

Заметим, что в любом случае неизменяемость или изменяемость - это свойство типа, а не его реализации. Реализация должна просто поддерживать это свойство абстракции.

#### 3.4.3.6 Классы операций

Операции абстракции данных распадаются на четыре класса.

1. Примитивные конструкторы. Эти операции создают объекты соответствующего им типа, не используя никаких объектов в качестве аргумента. Примером таких операций является операция create для набора intset.
2. Конструкторы. Эти операции используют в качестве аргументов объекты соответствующего им типа и создают другие объекты такого же типа.
3. Модификаторы. Эти операции модифицируют объекты соответствующего им типа. Например, операции insert и delete - модификаторы для наборов intset. Очевидно, что только изменяемые типы, могут иметь модификаторы.
4. Наблюдатели. Эти операции используют в качестве аргументов объекты соответствующего им типа и возвращают результаты другого типа. Они используются для получения информации об объектах. Сюда относятся, например, операции size, member, choose для наборов intset.



Обычно примитивные конструкции создают не все, а только некоторые объекты. Другие объекты создаются конструкторами или модификаторами.

Модификаторы играют ту же роль для изменяемых типов, что и конструкторы для неизменяемых. Изменяемый тип может иметь как конструкторы, так и модификаторы.

Иногда наблюдатели комбинируются с конструкторами или модификаторами.

#### *3.4.3.7 Полнота*

Тип данных является полным, если он обеспечивает достаточно операций для того, чтобы все требующиеся пользователю работы с объектами могли быть проделаны с приемлемой эффективностью. Строгое определение полноты дать невозможно, хотя существуют пределы относительно того, насколько мало операций может иметь тип, оставаясь при этом приемлемым.

#### **Например:**

Для наборов `intset` мы предоставили только операции `create`, `insert`, `delete` и программы не могут получить никакой информации относительно элементов набора (т.к. не имеется наблюдателей). С другой стороны, если мы добавим к этим трем операциям только одну операцию `size`, то сможем иметь информацию об элементах набора (например, можем проверять на принадлежность, удаляя целые числа и анализируя, изменился ли размер)., но тип в этом случае получится неэффективным и неприемлемым.

В общем случае абстракция данных должна иметь операции по крайней мере трех из четырех рассматриваемых нами классов. Она должна иметь примитивные конструкторы, наблюдатели и либо конструкторы (если она неизменяемая), либо модификаторы (если она изменяемая).

Полнота зависит от контекста использования, т.е. тип должен иметь достаточно богатый набор операций для его предполагаемого использования. Если тип предполагается использовать в ограниченном контексте (таком, как , например, одна программа), то должно быть обеспечено достаточно операций только для этого контекста. Если тип предназначен для общего использования, желательно иметь богатый набор операций.

Чтобы решить имеет ли абстракция достаточно данных операций, установите прежде всего все, что пользователи могут пожелать делать с ней. Затем продумайте как эти вещи могут быть осуществлены с использованием данного набора операций. Если что-то окажется слишком неэффективным или слишком громоздким (или то и другое вместе), исследуйте,

может ли этому помочь какая-нибудь дополнительная операция. Иногда существенное улучшение в производительности можно получить открыв доступ к представлению.

**Например:**

Мы можем исключить операцию `member` для наборов `intset` так как эта операция может быть реализована вне типа при помощи других операций. Однако проверка на принадлежность набору - операция общего пользования, и будет работать быстрее, если выполняется внутри реализации. Следовательно, тип данных `intset` должен иметь эту операцию.

Может быть таких слишком много операций в типе. В этом случае абстракция будет менее понятной, а ее реализация и работа с ней - более сложными. Введение дополнительных операций должно быть соотнесено с затратами на реализацию этих операций.

Если тип является полным, его набор операций может быть расширен процедурами, функционирующими вне реализации типа.

#### **3.4.4 Исключительные ситуации**

Процедурная абстракция есть отображение аргументов в результаты с возможной модификацией некоторых аргументов. Аргументы принадлежат области определения процедуры, а результаты - области значений.

Часто процедура имеет смысл только для аргументов, принадлежащие подмножеству ее области определения.

**Например:**

Операция `choose` для наборов целых чисел имеет смысл, только если ее аргументы - не пустой набор. Пока мы учитываем эту ситуацию, используя процедуру, заданную на некоторой части ее области определения.

```
Choose = proc (s:intset) returns (int)
```

```
requires s - не пустой набор.
```

```
Effects Возвращает произвольный элемент s
```

Тот, кто обращается к такой процедуре, должен гарантировать, что аргументы принадлежат допустимому подмножеству, области определения, и тот, кто реализует процедуру, может игнорировать аргументы, не принадлежащие этому подмножеству. Таким образом, реализуя операцию `choose`, мы игнорируем случай пустого набора.

Бывает, однако, что такие процедуры плохи. Компилятор не может гарантировать (в общем случае), что аргументы такой процедуры, как `choose`, действительно принадлежат допустимому подмножеству, и ввиду этого процедура может быть вызвана с аргументами, не

принадлежащими этому подмножеству. Когда это происходит возникает ошибка, которую очень трудно отследить. Например, программа может продолжать выполняться некоторое время после того, как произошла эта ошибка, и испортить важные данные.

Такие процедуры приводят к неустойчивым программам. Устойчивая программа - это такая программа, которая ведет себя корректно даже в случае ошибки, конечно, в случае ошибки программа может быть неспособна вести себя точно также, как если бы не было ошибки, но она должна вести себя определенным, предсказуемым образом. В идеальном случае она должна продолжать работать после ошибки, осуществляя некоторую аппроксимацию ее поведения при отсутствии ошибки. Говорят, что такая программа обеспечивает хорошую деградацию. В худшем случае она должна остановиться с сообщением об ошибке, не испортив при этом никаких постоянных данных.

Метод, который гарантирует устойчивость, заключается в использовании процедур, заданных на всей области определения. Если процедура не способна осуществить свою функцию для некоторых аргументов, она должна оповестить обратившуюся к ней о возникшем затруднении. Таким образом разрешение ситуации передается обратившемуся к процедуре, которой может быть в состоянии что-то предпринять или избежать катастрофических последствий ошибки.

Конечно, проверка входных аргументов на принадлежность допустимому подмножеству области определения занимает время, и есть искушение не делать эти проверки или использовать их только при отладке и отключить при счете. Однако в общем случае это не самое мудрое решение. Лучше выработать привычку защитного программирования, т.е. составлять каждую программу так, чтобы она защищала саму себя от ошибок. Ошибки могут возникнуть из-за других процедур, из-за аппаратной части или из-за неправильного ввода данных пользователем; эти последние ошибки будут иметь место даже при отлаженном математическом обеспечении.

Защитное программирование облегчает отладку. При эксплуатации оно еще более ценно, т.к. исключает возможность того, что незначительная ошибки приведет к серьезной проблеме, как, например, к порче данных. Проверка на нелегальные аргументы может быть отключена только если мы точно знаем, что ошибки быть не может или если проверка приводит к крайней неэффективности.

Как может быть оповещен обратившийся к процедуре или возникла проблема? Одна из возможностей - это определить специфический результат, возвращаемый в случае ошибки.

Однако это не очень хорошее решение, т.к. обращение к процедуре с нелегальными аргументами, вероятно, является ошибкой, более конструктивно рассматривать этот случай особо, чтобы использующему данную процедуру программисту было труднее игнорировать эту ошибку по невнимательности. Кроме того, если каждое значение возвращаемого типа есть возможный результат процедуры, то это решение не приводит к положительному результату, т.к. нет значения, с помощью которого можно было бы оповестить об ошибке. Предпочтительнее был бы метод, который не зависел бы от того, есть такое значение или нет.

Разделим область определения  $D$  на несколько подмножеств.

$$D = D_0 \cup D_1 \cup \dots \cup D_n$$

Процедура ведет себя по-разному на каждом из них.

**Например:** Для операции choose

$D_0 = \{\text{все непустые наборы}\}$

$D_1 = \{\text{пустой набор}\}$

Мы обеспечим оповещение, имея различные области значения для каждого подмножества области определения и придав различным случаям различные имена. Произвольно дадим случаю аргумента из  $D_0$  имя "обычный". Других случаев могут быть выбраны тем, КТО ОПРЕДЕЛЯЕТ ПРОЦЕДУРУ. Таким образом имеем:

$p: D_0 \rightarrow \text{обычный} (R_0)$

$D_1 \rightarrow \text{имя}_1 (R_1)$

.....

$D_n \rightarrow \text{имя}_n (R_n),$

где  $\text{имя}_1, \dots, \text{имя}_n$  - имена "исключительных" ситуаций, соответствующих аргументам, принадлежащим  $D_1, \dots, D_n$ . Соответственно, а  $R_0, \dots, R_n$  описывает число, порядок и типы результатов, возвращаемые в каждом случае.

$\text{Имя}_1, \dots, \text{имя}_n$  называются **исключительными ситуациями**.

**Например:**

choose:  $D_0 \rightarrow \text{обычный}(\text{int})$

$D_1 \rightarrow \text{empty}(),$

где  $R_1$  пусто, т.е. в этом случае пользователю не возвращается никакого результата.

#### 3.4.4.1 Спецификации

Чтобы специфицировать процедуры, которые вызывают исключительные ситуации, добавим в спецификацию предложение **signals**:

**signals** // здесь приводится список имен и результатов исключительных ситуаций.

Это предложение - часть заголовка. Оно следует за предложением `returns`. Если исключительных ситуаций не имеется, оно может быть опущено. Исключительные ситуации должны быть разделены запятыми, а их результаты (если такие имеются) должны быть заключены в скобки.

**Например:**

предложение

```
choose = proc(i:intset) returns(int) signals(empty)
```

говорит о том, что процедура `choose` может вызывать исключительную ситуацию `empty` и что в этом случае не возвращается никакого результата.

Предложение

```
search=proc(a:array[int], x: int) returns(ind:int)
```

```
signals(not_in, duplicate(ind1:int))
```

говорит о том, что процедура `search` может вызывать две исключительные ситуации: `not.in`(без результата) и `duplicate` (результат - целое число). Имя `ind1` вводится для ссылок на этот результат в остальной части спецификации.

Как и раньше, секция `effects` должна определять поведение процедуры для всех фактических аргументов, отвечающих предложению `requires`, т.к. это поведение включает в себя исключительные ситуации, секция `effects` должна определять, что приводит к вызову каждой исключительной ситуации и что делает процедура в каждом таком случае. Если процедура сигнализирует об исключительной ситуации для аргументов, принадлежащих  $D_i$  возможные значения аргументов, принадлежащих  $D_i$  должны отсутствовать в предложении `requires`. Завершение процедуры оповещением об исключительной ситуации - это один из нормальных режимов работы этой процедуры.

**Например:**

```
choose = proc(s:intset) returns(int) signals(empty)
```

**effects** Если `size(s)=0`, то сигнализировать об исключительной ситуации `empty`, иначе произвольный элемент `s`.

```
Search = proc(a:array[int], x:int) returns(ind: int)
```

```
signals(not_in, duplicate (ind:int))
```

**requires** Массив `a` упорядочен в возрастающем порядке.

**Effects** Если  $x \in a$  один раз, то возвратить  $ind$ , такой, что  $a[ind]=x$ ; Если  $x \in a$  более чем один раз, то сигнализировать об исключительной ситуации  $duplicate(ind)$ , где  $ind1$  – индекс для одного из  $x$ , иначе сигнализировать об исключительной ситуации  $not\_in$ .

Заметим, что спецификация процедуры  $search$  содержит предложение  $requires$  и что, как обычно в секции  $effects$  предполагается, что все требования предложения  $requires$  удовлетворены. Область определения процедуры  $search$  есть.

$$D_0 = \{ \langle a, x \rangle \mid x \in a \text{ ровно один раз} \}$$
$$D_1 = \{ \langle a, x \rangle \mid x \in a \}$$
$$D_2 = \{ \langle a, x \rangle \mid x \in a \text{ более чем один раз} \}.$$

#### 3.4.4.2 Использование исключительных ситуаций в программах

При реализации процедуры с исключительными ситуациями работа программиста заключается в том, чтобы обеспечить соответствующую спецификации работу процедуры. Если спецификация включает в себя исключительные ситуации, программа должна в нужное время сигнализировать о соответствующих исключительных ситуациях и передавать определенные в спецификации значения. Для выполнения этой задачи, программа, возможно, должна будет обрабатывать исключительные ситуации, возникающие в процедурах, к которым она обращается.

Исключительные ситуации могут обрабатываться двумя различными способами. Иногда исключительная ситуация распространяется до другого уровня, т.е. вызывающая процедура также завершает сигнализацией об исключительной ситуации с тем же самым или другим именем.

Перед распространением исключительной ситуации вызывающая процедура может осуществить некоторую локальную обработку. Такая обработка иногда необходима для достижения соответствия со спецификацией вызывающей процедуры.

Например: операция типа данных должна гарантировать, что объекты перед ее завершением будут удовлетворять инварианту представления.

Другой способ заключается в том, что вызывающая процедура маскирует исключительную ситуацию, т.е. обрабатывает исключительную ситуацию сама.

#### 3.4.4.3 Некоторые аспекты проектирования программ

Исключительные ситуации следует использовать для устранения большинства ограничений, перечисленных в предложениях  $requires$ . Эти предложения следует оставлять

только из соображений эффективности или если контекст использования настолько ограничен, что мы можем быть уверены, что ограничения удовлетворяются.

Исключительные ситуации также следует использовать для того, чтобы избежать кодирования информации в обычных результатах. Используя исключительную ситуацию, мы можем легко отличить этот результат от обычного, что позволяет избежать потенциальной ошибки.

Не все ошибки вызывают исключительные ситуации.

**Например:**

Пусть в большом вводном файле имеется ошибочная запись и будет возможно продолжать обработку файла, пропустив эту запись. В этом случае целесообразно оповестить пользователя (а не программу) о произошедшей ошибке.

Исключительные ситуации являются механизмом взаимодействия программ, а не программ с пользователями. Для взаимодействия с пользователями на какое-нибудь устройство вывода может быть выдано сообщение об ошибке. Реакция на ошибку определяется в спецификации абстракции.

Однако исключительные ситуации не всегда связаны с ошибками. Для некоторых абстракций может быть более чем один тип обычного поведения процедур, и в этом случае исключительные ситуации - удобный инструмент. Они обеспечивают средства для обеспечения нескольких типов поведения и дают возможность вызывающему процедуру различать между различными случаями.

Обычно исключительные ситуации требуют больших затрат, чем обычный возврат. Следовательно, случай, который предположительно будет возникать, чаще нужно рассматривать как обычный случай. Если решение о том, какой случай будет обычным, принимается из соображений производительности, то очевидно, что с исключительной ситуацией не должно быть связано никаких понятий типа “ошибка”.

Стоит поговорить о взаимосвязи модификации аргументов с завершением процедуры выходом на исключительную ситуацию. Секция `modifies` спецификации указывает, что аргумент может модифицироваться, но не говорит о том, когда это происходит. Если имеются исключительные ситуации, то обычно модификации происходят только для каких-нибудь из них. Что в точности происходит, должно быть описано в секции `effects`. Модификации должны быть описаны явно в каждом случае, в котором они совершаются; если не описано никаких модификаций, это означает, что они не совершаются вообще.

Если исключительная ситуация означает либо ошибку в математическом обеспечении, либо сбой в аппаратной части, либо другую ошибку, то полезно внести ошибку в протокол, чтобы ее можно было исправить впоследствии.

При реализации процедуры программист должен гарантировать, что она во всех ситуациях закончится в соответствии со спецификацией. Следует сигнализировать только об исключительных ситуациях, указанных в спецификации. Сигнализировать следует, кроме того, только по правильным причинам. При создании реализации разумно использовать исключительные ситуации вызываемых процедур и в зависимости от обстоятельств распространять или маскировать их.

#### **3.4.5 Абстракция итерации**

Абстракция итерации или итераторы являются некоторым обобщенным итерационным методом, имеющихся в большинстве языков программирования. Они позволяют пользователям выполнять итерации для произвольных типов данных удобным и эффективным способом.

Требуется некоторый общий метод итерации, который удобен и эффективен и который сохраняет абстракцию через спецификацию. Итератор обеспечивает эти требования. Он вызывается как процедура, но вместо окончания с выдачей всех результатов имеет много результатов, которые выдает каждый раз по одному. Полученные элементы могут использоваться в других модулях, которые задают действия, выполняемые для каждого такого элемента.

Каждый раз, когда итератор выдает некоторый элемент, над этим элементом выполняется тело цикла. Затем управление возвращается в итератор, так что он может выдавать следующий элемент.

Отметим разделение обязанностей в такой форме. Итератор ответственен за получение элемента, а модуль, содержащий цикл, определяет, что действие, которое будет над ним выполняться. Итератор может использоваться в различных модулях, которые выполняют разные действия над элементами, и он может быть реализован различными способами, не оказывая влияния на эти модули.

Если модуль выполняет некоторый цикл поиска, итератор может быть остановлен, как только будет найден интересующий его элемент.



### 3.4.5.1 Спецификация

Как и другие абстракции, итераторы должны быть определены, через спецификации. Форма спецификации итерации аналогична форме для процедуры. Заголовок имеет такой вид:

```
iname=iter(...) yields (...) signals (...)
```

Здесь мы используем ключевое слово `iter` для обозначения абстракции итератора. Итератор может совсем не выдавать объектов на каждой итерации или выдать несколько объектов. Число и тип этих объектов описывается в предложении `yields`. (Если для каждого `yields` не выдается ни одного объекта, то предложение `yields` может быть опущено.) Итератор может не выдавать никаких результатов, когда он заканчивается нормально, но он может заканчиваться по исключительной ситуации с именем и результатами, указанными в предложении `signals`.

**Например:**

```
elements = iter (s:intset) yields (int)
```

**requires** `s` не модифицируется в теле цикла.

**Effects** выдает элементы `s` в некотором произвольном порядке, причем каждый элемент только один раз.

Эта операция вполне правдоподобна для набора `intset`. Отметим, что операция `elements` не имеет исключительных ситуаций. Если ей будет задан в качестве аргумента пустой набор `intset`, она просто заканчивается, не выдавая ни одного элемента. Характерно, что использование итераторов исключает проблемы, связанные с заданием некоторых аргументов (таких, как пустой набор `intset`) для соответствующих процедур (таких, как процедура `choose`).

### 3.4.5.2 Вопросы проектирования

Итераторы будут включаться среди других операций в большинство типов данных, особенно в такие типы, чьи объекты являются совокупностями других объектов. Итераторы часто необходимы для полноты - они позволяют организовать доступ к элементам некоторой совокупности таким способом, который эффективен и удобен.

Тип может иметь несколько итераторов.

Для изменяемых совокупностей мы постоянно требовали, чтобы совокупность, по которой происходит итерация, не изменялась в теле цикла. Если мы опустим это требование, итератор должен работать по известным правилам даже в том случае, когда сделаны модификации.

Один подход состоит в том, что требуется, чтобы итератор выдавал элементы, содержащиеся в его аргументе совокупности в момент обращения, даже если модификации происходят позднее.

Поведение заданного таким образом итератора хорошо определено, но его реализация, вероятно будет неэффективна.

Модификации самим итератором совокупности должны избегаться.

## Литература

1. UML : спец. справ. / Джеймс Рамбо, Айвар Якобсон, Грэди Буч. - СПб. [и др.] : Питер, 2002. - 656 с. : ил. - (Справочник). - Алф. указ.: с. 632-652. - 5000 экз. - ISBN 5-318-00174-2
2. Ананьев П.И. Разработка приложений на базе СУБД. [Электронный ресурс]: Учебное пособие. Барнаул 2015. – 123с. - Режим доступа: <http://elib.altstu.ru>, свободный.
3. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. - М.: “Издательство Бином”, СПб.: “Невский диалект”, 1999г. - 560 с., ил.
4. Переиздание: Буч Г., Якобсон А., Рамбо Дж. UML. Классика CS. 2-е изд. / Пер. с англ.; Под общей редакцией проф. С. Орлова – СПб.: Питер, 2006. – 736 с.: ил.
5. Маглинец Ю.А. Анализ требований к автоматизированным информационным системам БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий - ИНТУИТ.ру, 2008
6. Маклафлин Б. Объектно-ориентированный анализ и проектирование /Б. Маклафлин, Г. Поллайс, Д. Уэст ; [пер. с англ. Е. Матвеев].-Санкт-Петербург: Питер, 2013.-601 с.: ил.