**АВТОНОМНАЯ НЕКОММЕРЧЕСКАЯ ОРГАНИЗАЦИЯ ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**

**«БАЛТИЙСКИЙ ИНФОРМАЦИОННЫЙ ТЕХНИКУМ»**

**(АНО ПО «БИТ»)**

**УТВЕРЖДАЮ**

**Директор АНО ПО «БИТ»**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **В.В.Сергеев**

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2021 г.

**ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ**

**по профилю специальности**

**Что такое The UML**

**Виды диаграмм UML**

Обучающегося:

**Пономаренко Максима Борисовича**

(фамилия, имя, отчество)

Специальность: 09.02.07 «Информационные системы и программирование»

Квалификация: Программист

Курс: 2 группа \_\_\_\_\_ИС-20\З\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Сроки прохождения практики**: с 17 ноября 2021 г по 30 ноября 2021 г.**

Руководитель практики: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **И.О. Янькова**

(подпись)

Заведующая отделением \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **С.Н.Милютина**

(подпись)

**ОБЪЕМ УЧЕБНОЙ ПРАКТИКИ**

|  |  |  |  |
| --- | --- | --- | --- |
| Вид практики | Количество часов | Форма проведения | Вид аттестации |
| Учебной практика по Что такое The UML  Виды диаграмм UML | 72 | концентрированно | Дифференцированный зачет |

СОДЕРЖАНИЕ УЧЕБНОЙ ПРАКТИКИ

Таблица 3

|  |  |  |
| --- | --- | --- |
| Профессиональ-ные модули | Перечень работ |  |
| Учебной практика по по  Что такое The UML  Виды диаграмм UML | Что такое The UML  Виды диаграмм UML  Диаграмма классов: крупным планом    Диаграмма активностей: крупным планом  Диаграммы взаимодействия: крупным планом  Диаграммы прецедентов: крупным планом    Обзор CASE-средств для построения диаграмм UML  Заключение |
| Всего 72 часа | |

## 

## **АТТЕСТАЦИОННЫЙ ЛИСТ**

Ф.И.О: **Пономаренко Максим Борисович**

Студент 2 курса

Учебная группа 20-ИС\З

Специальность 09.02.07 «Информационные системы и программирование»

Что такое The UML

Виды диаграмм UML

|  |  |  |
| --- | --- | --- |
| № | Общие и профессиональные компетенции | Отметка о выполнении |
| 1 | Что такое The UML | да |
| 2 | Виды диаграмм UML | да |
| 3 | Диаграмма классов: крупным планом | да |
| 4 | Диаграмма активностей: крупным планом | да |
| 5 | Диаграммы взаимодействия: крупным | да |
| 6 | Диаграммы прецедентов: крупным планом | да |
| 7 | Обзор CASE-средств для построения диаграмм UML | да |
| 8 | Заключение | да |

Руководитель практики И.О. Янькова

(фамилия, имя, отчество, должность)

Подпись\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## ХАРАКТЕРИСТИКА

На: **Пономаренко Максима Борисовича***,*

обучающегося учебной группы: ИС-20\З, курса: 2

За период учебной практики

с \_17\_ноября\_2021г. по \_30\_ноября 2021г

Что такое The UML

Виды диаграмм UML

выполнял следующие виды работ:

Что такое The UML

Виды диаграмм UML

Диаграмма классов: крупным планом

Диаграмма активностей: крупным планом

Диаграммы взаимодействия: крупным планом

Диаграммы прецедентов: крупным планом

Обзор CASE-средств для построения диаграмм UML

Руководитель практики \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ И.О. Янькова

(подпись) (Ф.И.О.)

## **ОЦЕНКА УРОВНЯ ОСВОЕНИЯ КОМПЕТЕНЦИЙ**

|  |  |  |
| --- | --- | --- |
| Наименование компетенций | Уровень освоения\* | Примечание |
| Что такое The UML |  |  |
| Виды диаграмм UML |  |  |
| Диаграмма классов: крупным планом |  |  |
| Диаграмма активностей: крупным планом |  |  |
| Диаграммы взаимодействия: крупным |  |  |
| Диаграммы прецедентов: крупным планом |  |  |
| Обзор CASE-средств для построения диаграмм UML |  |  |
| Заключение |  |  |

*\*Оценивание осуществляется по пятибалльной системе*

Рекомендуемая оценка по практике \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021г.

Руководитель практики \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_И.О.Янькова

(МП) (подпись) (Ф.И.О.)

«Утверждаю»

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Руководитель практики И.О. Янькова | | | | | | | |
| (должность, ф.и.о. руководителя практики ) | | | | | | | |
|  | | | |  |  | | |
| (подпись) | | | |  | (расшифровка) | | |
| « |  | » |  | | | 2021г. |  |
| М.П. | | | |  |  | | |

## **ОТЧЕТ**

по практике с \_17.11.2021г по \_30.11.2021года

|  |  |  |  |
| --- | --- | --- | --- |
| по специальности | 09.02.07 «Информационные системы и программирование» | | |
|  | (код) |  | (наименование специальности/профессии) |

|  |  |  |
| --- | --- | --- |
| Выполнил  студент ФИО | | **Пономаренко Максим Борисович** |
|  | | (ФИО студента) |
| Учебной группы | | **ИС-20\З** |
|  | | (наименование группы) |
|  |  | **Что такое The UML**  **Виды диаграмм UML** |
| (Индекс модуля) | | (название профессионального модуля) |
|  | |  |
|  | |  |
| Дата сдачи | |  |
| Оценка | |  |
|  | |  |

### Отчет о учебной практики

**Что такое The UML**

**Виды диаграмм UML**

### Студент: Пономаренко Максим Борисович

**2** курс

Учебная группа **20-ИС\З**

Специальность 09.02.07 «Информационные системы и программирование»

**Содержание:**

**1.Что такое The UML 9-11**

**2.Виды диаграмм UML 11-16**

**3.Диаграмма классов: крупным планом 16-19**

**4.Диаграмма активностей: крупным планом 20-23**

**5.Диаграммы взаимодействия: крупным планом 23-26**

**6.Диаграммы прецедентов: крупным планом 27-29**

**7.Обзор CASE-средств для построения диаграмм UML 30-32**

**8.Заключение 32**

**1.Что такое The UML**

**Назначение языка**

*UML* - унифицированный *язык моделирования*. Из этих трех слов главным является *слово* " **язык** ". Что же такое язык? Не будем изобретать велосипед, а лучше заглянем в *глоссарий*, благо в Интернете их величайшее множество. Сделав это, мы скорее всего обнаружим *определение*, подобное приведенному ниже.

**Язык** - система знаков, служащая:

* средством человеческого общения и мыслительной деятельности;
* способом выражения самосознания личности;
* средством хранения и передачи информации.

Язык включает в себя набор знаков (словарь) и правила их употребления и интерпретации (грамматику).

К этому достаточно исчерпывающему определению нужно добавить, что языки бывают естественные и искусственные, формальные и неформальные. *UML* - язык формальный и искусственный, хотя, как мы увидим далее, этот ярлык к нему не совсем подходит. Искусственный он потому, что у него имеются авторы, о которых мы еще не раз упомянем в дальнейшем (в то же время, развитие *UML* непрерывно продолжается, что ставит его в один ряд с естественными языками). Формальным его можно назвать, поскольку имеются правила его употребления (правда, описание *UML* содержит и явно неформальные элементы, как мы, опять-таки, позже увидим). Еще один нюанс: *UML* - язык графический, что также немного путает ситуацию!

При описании формального искусственного языка, что мы уже видели на примерах описания языков программирования, как правило, описываются такие его элементы, как:

1. *синтаксис*, то есть определение правил построения конструкций языка;
2. *семантика*, то есть определение правил, в соответствии с которыми конструкции языка приобретают смысловое значение;
3. *прагматика*, то есть определение правил использования конструкций языка для достижения нужных нам целей.

Естественно, *UML* включает все эти элементы, хотя, как мы опять-таки увидим далее, в их описании тоже наблюдаются отличия от правил, принятых в языках программирования.

Второе *слово* в фразе, которой расшифровывается аббревиатура *UML* - *слово* " **моделирование** ". Да, *UML* - это *язык моделирования*. Причем *объектно-ориентированного моделирования*. Более подробно о смысле понятия "*моделирование*" мы поговорим чуть позже, а пока отметим, что *слово* это весьма многозначно. В английском языке есть целых два слова - **modeling** и **simulation**, которые оба переводятся как "*моделирование*", хотя означают разные понятия. *Modeling* подразумевает создание модели, лишь описывающей *объект*, а *simulation* предполагает получение с помощью созданной модели некоторой дополнительной информации об объекте. *UML* в первую *очередь* - *язык моделирования* именно в первом смысле, то есть средство построения *описательных моделей*. Как средство симулирования его тоже можно использовать, хотя для этой роли он подходит не так хорошо.

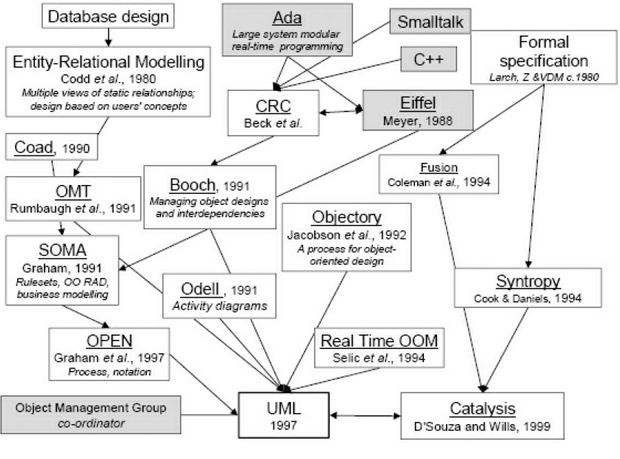
Третье *слово* в названии *UML* - *слово* " **унифицированный** ". Его можно понимать тоже неоднозначно. В литературе можно встретить описание эры "до *UML*" как "войны методов" моделирования, ни один из которых "не дотягивал" до уровня индустриального стандарта. *UML* как раз и стал таким единым универсальным стандартом для *объектно-ориентированного моделирования*, которое во времена его создания как раз "вошло в моду". "Единым" языком моделирования *UML* можно назвать еще и потому, что в его создании, как мы увидим далее, объединились усилия авторов трех наиболее популярных методов моделирования (и не только их).

Подводя итоги, кратко можно сказать, что *UML* - искусственный язык, который имеет некоторые черты естественного языка, и *формальный язык*, который имеет черты неформального. Это звучит не очень понятно, но это действительно так!

**Историческая справка**

Откуда взялся The *UML*? Если говорить коротко, то *UML* вобрал в себя черты нотаций Грейди Буча (Grady Booch), Джима Румбаха (Jim Rumbaugh), Айвара Якобсона (Ivar Jacobson) и многих других.

В не такие уж и далекие 80-е годы было множество различных методологий моделирования. Каждая из них имела свои достоинства и недостатки, а также свою нотацию. То смутное время получило название "войны методов". Проблема в том, что разные люди использовали разные нотации, и для того чтобы понять, что описывает та или иная *диаграмма*, зачастую требовался "переводчик". Один и тот же символ мог означать в разных нотациях абсолютно разные вещи! На рисунке ниже можно увидеть лишь малую часть многообразия методов, которые существовали в то время и в какой-то мере повлияли на *UML* ([рис. 1.1](https://intuit.ru/studies/courses/1007/229/lecture/5952?page=1#image.1.1)).

[](https://intuit.ru/EDI/23_04_17_1/1492899714-28128/tutorial/356/objects/1/files/01_01.gif)

[увеличить изображение](https://intuit.ru/EDI/23_04_17_1/1492899714-28128/tutorial/356/objects/1/files/01_01.gif)  
**Рис. 1.1.**

К тому же примерно в это же время (начало 80-х) стартовала "объектно-ориентированная эра". Все началось с появлением семейства языков программирования SmallTalk, которые применяли некоторые понятия языка Simula-67, использовавшегося в 60-х годах. Появление объектно-ориентированного подхода в первую *очередь* было обусловлено увеличением сложности задач. *Объектно-ориентированный подход* внес достаточно радикальные изменения в сами принципы создания и функционирования программ, но, в то же время, позволил существенно повысить *производительность* труда программистов, *по*-иному взглянуть на проблемы и методы их решения, сделать программы более компактными и легко расширяемыми. Как результат, языки, первоначально ориентированные на традиционный подход к программированию, получили ряд объектноориентированных расширений. Одной из первых, в середине 80-х, была *фирма* Apple со своим проектом *Object Pascal*. Кроме этого, *объектно-ориентированный подход* породил мощную волну и абсолютно новых программных технологий, вершинами которой стали такие общепризнанные сегодня платформы, как Microsoft .*NET* Framework и *Sun* *Java*.

Но самое главное, что появление *ООП* требовало удобного инструмента для моделирования, единой нотации для описания сложных программных систем. И вот "три амиго", три крупнейших специалиста, три автора наиболее популярных методов решили объединить свои разработки. В 1991-м каждый из "трех амиго" начал с написания книги, в которой изложил свой метод ООАП. Каждая методология была *по*-своему хороша, но каждая имела и недостатки. Так, метод Буча был хорош в проектировании, но слабоват в анализе. *OMT* Румбаха был, наоборот, отличным средством анализа, но плох в проектировании. И наконец, Objectory Якобсона был действительно хорош с точки зрения *user experience*, на который ни метод Буча, ни *OMT* не обращали особого внимания. Основной идеей Objectory было то, что *анализ* должен начинаться с прецедентов, а не с *диаграммы классов*, которые должны быть производными от них.

К 1994-му существовало 72 метода, или частные методики. Многие из них "перекрывались", т. е. использовали похожие идеи, нотации и т. д. Как уже говорилось выше, чувствовалась острая потребность, "социальный заказ" - закончить "войну методов" и объединить в одном унифицированном средстве все лучшее, что было создано в области моделирования.

А что сейчас? The *UML* живет и развивается. Сейчас мы имеем *UML* 2.0 и десятки CASE-средств, поддерживающих *UML*, о многих из которых будет рассказано в ["Обзор CASE-средств для построения диаграмм UML"](https://intuit.ru/studies/courses/1007/229/lecture/5963). Вопреки популярному мнению, в наши дни Rational не владеет *UML*, но продолжает работать над ним. *UML* же принадлежит *OMG*, а сама Rational ныне является одним из подразделений *IBM* и фигурирует во всех документах как *IBM* Rational. *UML* же получил множество пакетов расширений, называемых *профайлами* и позволяющих использовать его для моделирования систем из специфических предметных областей.

**2.Виды диаграмм UML**

**Почему нужно несколько видов диаграмм**

Для начала определимся с терминологией. В предисловии к этой лекции мы неоднократно использовали понятия системы, модели и диаграммы. *Автор* уверен, что каждый из нас интуитивно понимает смысл этих понятий, но, чтобы внести полную *ясность*, снова заглянем в *глоссарий* и прочтем следующее:

**Система** - совокупность взаимосвязанных управляемых подсистем, объединенных общей целью функционирования.

Да, не слишком информативно. А что же такое тогда подсистема? Чтобы прояснить ситуацию, обратимся к классикам:

**Системой** называют набор подсистем, организованных для достижения определенной цели и описываемых с помощью совокупности моделей, возможно, с различных точек зрения.

Что ж, ничего не попишешь, придется искать *определение* подсистемы. Там же сказано, что **подсистема** - это совокупность элементов, часть из которых задает спецификацию поведения других элементов. Ян Соммервилл объясняет это понятие таким образом:

**Подсистема** - это система, функционирование которой не зависит от сервисов других подсистем. Программная система структурируется в виде совокупности относительно независимых подсистем. Также определяются взаимодействия между подсистемами.

Тоже не слишком понятно, но уже лучше. Говоря "человеческим" языком, система представляется в виде набора более простых сущностей, которые относительно самодостаточны. Это можно сравнить с тем, как в процессе разработки программы мы строим графический *интерфейс* из стандартных "кубиков" - визуальных компонентов, или как сам текст программы тоже разбивается на модули, которые содержат подпрограммы, объединенные *по* функциональному признаку, и их можно использовать повторно, в следующих программах.

С понятием системы разобрались. В процессе проектирования система рассматривается **с разных точек зрения** с помощью моделей, различные представления которых предстают в форме диаграмм. Опять-таки у читателя могут возникнуть вопросы о смысле понятий *модели* и *диаграммы*. Думаем, красивое, но не слишком понятное *определение* **модели как семантически замкнутой абстракции системы** вряд ли прояснит ситуацию, поэтому попробуем объяснить "своими словами".

**Модель** - это некий (материальный или нет) *объект*, отображающий лишь наиболее значимые для данной задачи характеристики системы. Модели бывают разные - материальные и нематериальные, искусственные и естественные, декоративные и математические...

Приведем несколько примеров. Знакомые всем нам пластмассовые игрушечные автомобильчики, которыми мы с таким азартом играли в детстве, это не что иное, как *материальная искусственная декоративная* модель реального автомобиля. Конечно, в таком "авто" нет двигателя, мы не заполняем его бак бензином, в нем не работает (более того, вообще отсутствует) коробка передач, но как модель эта игрушка свои функции вполне выполняет: она дает ребенку *представление* об автомобиле, поскольку отображает его характерные черты - наличие четырех колес, кузова, дверей, окон, способность ехать и т. д.

В ходе медицинских исследований опыты на животных часто предшествуют клиническим испытаниям медицинских препаратов на людях. В таком случае животное выступает в роли *материальной естественной* модели человека.

mg-a\cfrac{dx}{dt}=m\cfrac{d^2x}{dt^2}

Уравнение, изображенное выше - тоже модель, но это модель математическая, и описывает она движение материальной точки под действием силы тяжести.

Осталось лишь сказать, что такое *диаграмма*. **Диаграмма** - это графическое *представление* *множества* элементов. Обычно изображается в виде графа с вершинами (сущностями) и ребрами (отношениями). Примеров диаграмм можно привести множество. Это и знакомая нам всем со школьных лет *блок-схема*, и схемы монтажа различного оборудования, которые мы можем видеть в руководствах пользователя, и *дерево* файлов и каталогов на диске, которое мы можем увидеть, выполнив в консоли *Windows* команду tree, и многое-многое другое. В повседневной жизни диаграммы окружают нас со всех сторон, ведь рисунок воспринимается нами легче, чем текст...

Но вернемся к проектированию *ПО* (и не только). В этой отрасли с **помощью диаграмм можно визуализировать систему с различных точек зрения**. Одна из диаграмм, например, может описывать взаимодействие пользователя с системой, другая - изменение состояний системы в процессе ее работы, третья - взаимодействие между собой элементов системы и т. д. Сложную систему можно и нужно представить в виде набора небольших и почти независимых моделей-диаграмм, причем ни одна из них не является достаточной для описания системы и получения полного представления о ней, поскольку каждая из них фокусируется на каком-то определенном аспекте функционирования системы и выражает разный *уровень абстракции*. Другими словами, каждая модель соответствует некоторой определенной, частной точке зрения на проектируемую систему.

Несмотря на то что в предыдущем абзаце мы весьма вольготно обошлись с понятием модели, следует понимать, что в контексте приведенных выше определений **ни одна отдельная диаграмма не является моделью**. Диаграммы - лишь средство визуализации модели, и эти два понятия следует различать. Лишь **набор диаграмм составляет модель системы** и наиболее полно ее описывает, но не одна *диаграмма*, вырванная из контекста.

**Виды диаграмм**

*UML* 1.5 определял **двенадцать типов диаграмм**, разделенных на три группы:

* четыре типа диаграмм представляют статическую структуру приложения;
* пять представляют поведенческие аспекты системы;
* три представляют физические аспекты функционирования системы (диаграммы реализации).

Текущая версия *UML* 2.1 внесла не слишком много изменений. Диаграммы слегка изменились внешне (появились фреймы и другие визуальные улучшения), немного усовершенствовалась *нотация*, некоторые диаграммы получили новые наименования.

Впрочем, *точное число* *канонических диаграмм* для нас абсолютно неважно, так как мы рассмотрим не все из них, а лишь некоторые - *по* той причине, что количество типов диаграмм для конкретной модели конкретного приложения не является строго фиксированным. Для простых приложений нет необходимости строить все без исключения диаграммы. Например, для локального приложения не обязательно строить диаграмму развертывания. Важно понимать, что перечень диаграмм зависит от специфики разрабатываемого проекта и определяется самим разработчиком. Если же любопытный читатель все-таки пожелает узнать обо всех диаграммах *UML*, мы отошлем его к стандарту *UML* (<http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML>). Напомним, что цель этого курса - не описать абсолютно все возможности *UML*, а лишь познакомить с этим языком, дать первоначальное *представление* об этой технологии.

Итак, мы кратко рассмотрим такие виды диаграмм, как:

* *диаграмма прецедентов*;
* диаграмма классов;
* *диаграмма объектов*;
* диаграмма последовательностей;
* диаграмма взаимодействия;
* диаграмма состояний;
* *диаграмма активности*;
* *диаграмма развертывания*.

О некоторых из этих диаграмм мы будем говорить подробнее в следующих лекциях. Пока же мы не станем заострять внимание на подробностях, а зададимся целью научить читателя хотя бы визуально различать виды диаграмм, дать начальное *представление* о назначении основных видов диаграмм. Итак, начнем.

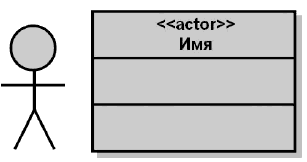
**Диаграмма прецедентов (use case diagram)**

Любые (в том числе и программные) системы проектируются с учетом того, что в процессе своей работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются **экторами**, причем каждый эктор ожидает, что система будет вести себя строго определенным, предсказуемым образом. Попробуем дать более строгое определение эктора. Для этого воспользуемся замечательным визуальным словарем по UML *Zicom Mentor*:

**Эктор (actor)** - это множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Эктором может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности.

Графически эктор изображается либо " *человечком* ", подобным тем, которые мы рисовали в детстве, изображая членов своей семьи, либо *символом класса с соответствующим стереотипом*, как показано на рисунке. Обе формы представления имеют один и тот же смысл и могут использоваться в диаграммах. "Стереотипированная" форма чаще применяется для представления системных экторов или в случаях, когда эктор имеет свойства и их нужно отобразить ([рис. 2.1](https://intuit.ru/studies/courses/1007/229/lecture/5954?page=1#image.2.1)).

Внимательный читатель сразу же может задать вопрос: *а почему эктор, а не актер*? Согласны, слово "эктор" немного режет слух русского человека. Причина же, почему мы говорим именно так, проста - эктор образовано от слова **action**, что в переводе означает *действие*. Дословный же перевод слова "эктор" - *действующее лицо*- слишком длинный и неудобный для употребления. Поэтому мы будем и далее говорить именно так.



**Рис. 2.1.**

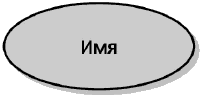
Тот же внимательный читатель мог заметить промелькнувшее в определении эктора слово "прецедент". Что же это такое? Этот вопрос заинтересует нас еще больше, если вспомнить, что сейчас мы говорим о *диаграмме прецедентов*. Итак,

**Прецедент (use-case)** - описание отдельного аспекта поведения системы с точки зрения пользователя (Буч).

Определение вполне понятное и исчерпывающее, но его можно еще немного уточнить, воспользовавшись тем же *Zicom Mentor* 'ом:

**Прецедент (use case)** - описание множества последовательных событий (включая варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецедент представляет поведение сущности, описывая взаимодействие между экторами и системой. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется.

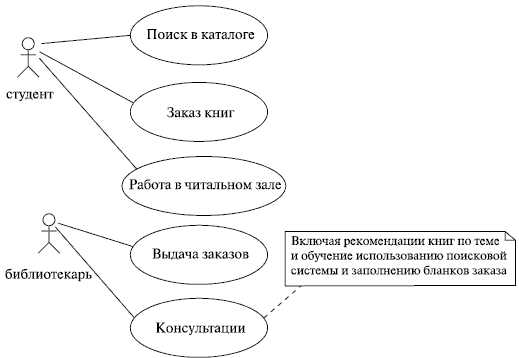
Прецеденты обозначаются очень простым образом - в виде эллипса, внутри которого указано его название. *Прецеденты и экторы соединяются с помощью линий*. Часто на одном из концов линии изображают *стрелку*, причем *направлена она к тому, у кого запрашивают сервис*, другими словами, чьими услугами пользуются. Это простое объяснение иллюстрирует *понимание прецедентов как сервисов*, пропагандируемое компанией IBM.



**Рис. 2.2.**

Прецеденты могут включать другие прецеденты, расширяться ими, наследоваться и т. д. *Все эти возможности мы здесь рассматривать не будем*. Как уже говорилось выше, цель этого обзора - просто научить читателя выделять диаграмму прецедентов, понимать ее назначение и смысл обозначений, которые на ней встречаются.

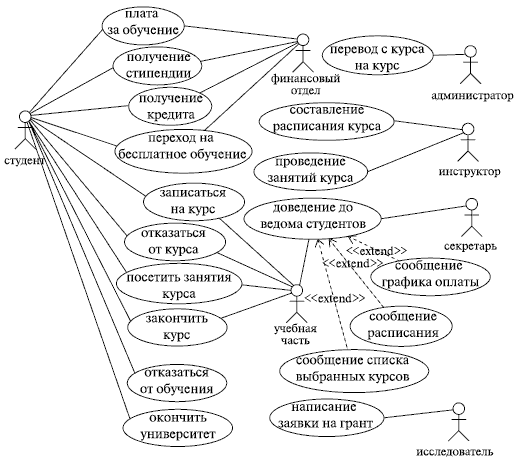
Кстати, к этому моменту мы уже потратили достаточно много времени на объяснение понятий и их условных обозначений. Наверное, пора уже, наконец, привести пример диаграммы прецедентов. Как вы думаете, что означает эта диаграмма ([рис. 2.3](https://intuit.ru/studies/courses/1007/229/lecture/5954?page=1#image.2.3))?



**Рис. 2.3.**

Полагаем, здесь все было бы понятно, если бы даже мы никогда не слышали о диаграммах прецедентов! Ведь так? Все мы в студенческие годы пользовались библиотеками (которые теперь для нас заменил Интернет), и потому все это для нас знакомо. Обратите также внимание на примечание, сопоставленное с одним из прецедентов. Следует заметить, что иногда на диаграммах прецедентов *границы системы обозначают прямоугольником*, в верхней части которого может быть указано *название системы*. Таким образом, прецеденты - действия, выполняемые системой в ответ на действия эктора, - помещаются внутри прямоугольника.

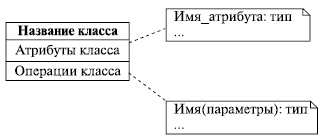
А вот еще один пример ([рис. 2.4](https://intuit.ru/studies/courses/1007/229/lecture/5954?page=1#image.2.4)). Думаем, вы сами, без нашей помощи, легко догадаетесь, о чем там идет речь.



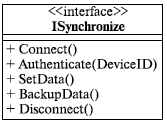
**Рис. 2.4.**

Из всего сказанного выше становится понятно, что *диаграммы прецедентов*относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное *средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями* продукта. Как мы уже могли убедиться, такие диаграммы очень просты для понимания и могут восприниматься и, что немаловажно, обсуждаться людьми, не являющимися специалистами в области разработки ПО.

Подводя итоги, можно выделить такие **цели создания диаграмм прецедентов**:

* определение границы и контекста моделируемой предметной области на ранних этапах проектирования;
* формирование общих требований к поведению проектируемой системы;
* разработка концептуальной модели системы для ее последующей детализации;
* подготовка документации для взаимодействия с заказчиками и пользователями системы
* **3.Диаграмма классов: крупным планом**
* **Как класс изображается на диаграмме UML?**
* *Архитектор* программного обеспечения в первую *очередь* обращает внимание на объекты *предметной области*. Программист же концентрируется на поведении этих объектов, пользуясь **классами**, к которым они принадлежат. Вот поэтому-то *диаграмма* классов и является одной из важнейших диаграмм *UML*. Она используется для документирования программных систем, и основным ее компонентом является **класс**. Что такое *класс*, мы уже говорили ранее, когда знакомились с видами диаграмм *UML*. В предыдущей лекции мы рассматривали назначение *диаграммы классов*, знакомились с примерами готовых диаграмм, но не вникали в тонкости обозначений, используемых на диаграмме. В тех примерах все казалось нам очень понятным и логичным. Тем не менее, некоторые нюансы все же следует рассмотреть, и как раз этим мы сейчас и займемся.
* *Класс* на диаграмме изображается в виде прямоугольника, разделенного горизонтальными линиями на три части. В первой части указывается название класса. Как правило, *имя класса* состоит из одного, *максимум* двух слов. Вторая часть содержит перечень атрибутов класса, которые характеризуют тот или иной *объект* этого класса в модели *предметной области*. Третья часть содержит перечень операций, отражающих его поведение в модели *предметной области* ([рис. 3.1](https://intuit.ru/studies/courses/1007/229/lecture/5956?page=1#image.3.1)). Все очень просто, не так ли?
* 
* **Рис. 3.1.**
* **А что внутри?**
* Мы узнали, как *класс* изображается и выглядит "снаружи". А что же внутри объектов класса? Пользователю об этом знать необязательно, более того, абсолютно не нужно. Для человека, использующего его, *объект* выступает в роли черного ящика. Скрывая от пользователя внутреннее устройство объекта, мы обеспечиваем его надежную работу. Сейчас мы рассмотрим, как убрать из поля зрения пользователя то, что ему знать не нужно.
* Читателя может слегка смутить *слово* "*пользователь*", которым мы злоупотребляли в предыдущем абзаце. Зачем вообще пользователю какие-то объекты и классы? Внесем *ясность*. Программист, использующий в своей программе созданные кем-то компоненты, как раз и выступает в роли такого пользователя. Зачем ему знать что внутри - он знает, какие атрибуты надо модифицировать и какие *операции* использовать, чтобы заставить *объект* работать именно так, как ему нужно! Более того, а многие ли из нас знают, как именно устроен и по каким принципам работает, например, *телевизор* - *объект* класса "Бытовой прибор"?
* Сокрытие от пользователя внутреннего устройства объектов называется *инкапсуляцией*. Если говорить более "научным" языком, то *инкапсуляция* - это защита отдельных элементов объекта, не затрагивающих существенных характеристик его как целого. *Инкапсуляция* нужна не только для того, чтобы создать иллюзию простоты объекта для пользователя (по словам Г. Буча). Но вернемся к примеру с телевизором. Нам этот прибор кажется очень простым только потому, что при работе с ним мы используем простой и понятный *интерфейс* - пульт дистанционного управления. Мы знаем: для того чтобы увеличить громкость звука, надо нажать вот эту кнопку, а чтобы переключить канал - вот эту. Как *телевизор* устроен внутри, мы не знаем. Более того - в отсутствие пульта ДУ такое *знание* было бы неудобным для нас и весьма опасным для самого телевизора, вздумай мы увеличить громкость с помощью паяльника. Поэтому-то пульт ДУ и защищает от нас "внутренности" телевизора! Вот так *инкапсуляция* реализуется в реальном мире.
* В программировании *инкапсуляция* обеспечивается немного по-другому - с помощью т. н. *модификаторов видимости*. С их помощью можно ограничить *доступ* к атрибутам и операциям объекта со стороны других объектов. Звучит это немного пугающе, но на самом деле все просто. Если *атрибут* или операция описаны с модификатором private, то *доступ* к ним можно получить только из *операции*, определенной в том же классе. Если же *атрибут* или операция описаны с модификатором видимости public, то к ним можно получить *доступ* из любой части программы. Модификатор protected разрешает *доступ* только из операций этого же класса и классов, создаваемых на его основе. В языках программирования могут встречаться *модификаторы видимости*, ограничивающие *доступ* на более высоком уровне, например, к классам или их группам, однако смысл инкапсуляции от этого не изменяется. В *UML* атрибуты и *операции* с модификаторами доступа обозначаются специальными символами слева от их имен:

|  |  |
| --- | --- |
| **Символ** | **Значение** |
| + | public - открытый доступ |
| - | private - только из операций того же класса |
| # | protected - только из операций этого же класса и классов, создаваемых на его основе |

* Рассмотренный ранее пример с телевизором средствами *UML* (конечно же, это очень высокоуровневая *абстракция*) можно изобразить так ([рис. 3.2](https://intuit.ru/studies/courses/1007/229/lecture/5956?page=1#image.3.2)):
* 
* **Рис. 3.2.**
* Не правда ли, все понятно и предельно просто? Зачем, например, пользователю знать числовые значения частот каналов? Он знает, что достаточно запустить процедуру автоматического поиска каналов и *телевизор* все сделает за него. Вот вам и *инкапсуляция* - оказывается, она повсюду вокруг нас. Оглянитесь и подумайте, сколько вещей вокруг имеют скрытые свойства и выполняют скрытые *операции*. Испугались? Вот то-то же!
* **Как использовать объекты класса?**
* Итак, мы рассмотрели инкапсуляцию - одно из средств защиты объектов. Все вроде бы понятно, но как же именно работать с объектом?
* Если уж говорить о защите объекта, то чтобы она действительно была эффективной, надо позаботиться о некоем стандартном и безопасном, не зависящим от языка программирования способе доступа к объекту. К тому же такой стандартный способ доступа должен быть простым и с точки зрения использования, и с точки зрения реализации. Вспомните пример с телевизором. Нажимая кнопки на пульте, мы ожидаем, что *телевизор* откликнется на это действие каким-то определенным образом - именно так, как мы ожидаем, а не иначе. То есть, с одной стороны, пульт ДУ является средством доступа к скрытым операциям, выполняемым телевизором, а с другой стороны - пульт обеспечивает нужное для нас поведение телевизора. В данном примере именно пульт является таким стандартным средством доступа к телевизору. Можно даже сказать, средством доступа, не зависящим от конкретной модели телевизора - вспомните об универсальных пультах и о том, как отключаете звук надоедливой рекламы на экране в вагоне поезда, используя КПК!
* В том же примере с телевизором у нас впервые промелькнуло *слово* *интерфейс*. И не случайно промелькнуло: именно так называют тот самый стандартный способ доступа к объекту. Более строго, *интерфейс* - это логическая *группа* открытых ( public ) операций объекта. Один и тот же *объект* может иметь несколько интерфейсов. У телевизора, например, их два - пульт ДУ и кнопки на корпусе. А может и больше - вспомните о возможности управлять бытовой техникой с помощью КПК или универсального пульта ДУ.
* Кстати, посмотрите внимательнее на пульт ДУ или на экран программы удаленного контроля. Что вы видите - кнопки? Или кнопки, сгруппированные по функциональному признаку? Да, именно так: кнопки, переключающие каналы, расположены отдельно, рядом - *группа* кнопок, отвечающих за регулировку громкости звука, рядом - *группа* программируемых кнопок, и т. д. В принципе, можно сказать, что пульт реализует не один, а несколько интерфейсов - по числу функциональных групп кнопок. Впрочем, это уже *формализм*: мы просто хотели проиллюстрировать слова "логическая *группа*" в определении интерфейса.
* Однако *интерфейс* - это не только и не столько *группа* операций объекта. *Интерфейс* отражает внешние проявления объекта, показывает, каким образом осуществляется взаимодействие с ним, скрывая остальные детали, не имеющие отношения к процессу взаимодействия.
* *Интерфейс* всегда реализуется некоторым классом, который в таком случае называют классом, *поддерживающим* *интерфейс*. Как мы уже говорили ранее, один и тот же *объект* может иметь несколько интерфейсов. Это означает, что *класс* этого объекта реализует все *операции* этих интерфейсов. К данному моменту в голове читателя может созреть вопрос: "Мы же, вроде бы, говорили о классах и объектах, а теперь вдруг перешли на интерфейсы. Да и вообще, используются ли они в практике программирования или являются просто изящной теоретической конструкцией?". Ответ на этот вопрос прост: многие из существующих технологий программирования (например, *COM*, *CORBA*, *Java Beans*) не только активно используют *механизм интерфейсов*, но и, по сути, полностью основаны на нем.
* Что ж, наверное, пришло время поговорить о том, как *интерфейс* изображается на диаграммах. Изображаться он может несколькими способами. Первый и самый простой из них - это *класс* со стереотипом <<interface>> ([рис. 3.3](https://intuit.ru/studies/courses/1007/229/lecture/5956?page=1#image.3.3)):
* 
* **Рис. 3.3.**
* Этот способ хорош, если нужно показать, какие именно *операции* предоставляет *интерфейс*. Если же такие подробности в данный момент не важны, предоставляемый *интерфейс* изображают в виде кружочка или, как говорят, "леденца" ( lollipop ) ([рис. 3.4](https://intuit.ru/studies/courses/1007/229/lecture/5956?page=1#image.3.4)):
* 
* **Рис. 3.4.**
* Обратите внимание на маленький значок на закладке папки ConduitSet. Это обозначение подсистемы, мы могли бы не рисовать его, а просто использовать стереотип <<subsystem>>. Впрочем, об этом мы еще поговорим.
* И наконец, еще один способ изображения интерфейса. Он не является альтернативой описанным ранее способам, а используется для изображения интерфейсов, **требующихся** объекту для выполнения его работы. Обозначается он очень простым и логичным символом. Впрочем, судите сами ([рис. 3.5](https://intuit.ru/studies/courses/1007/229/lecture/5956?page=1#image.3.5)):
* 
* **Рис. 3.5.**
* Наблюдательный читатель уже, наверное, заметил, как логически совмещаются символы предоставляемого и требуемого интерфейсов.
* Действительно, на диаграммах довольно часто можно увидеть такую картинку ([рис. 3.6](https://intuit.ru/studies/courses/1007/229/lecture/5956?page=1#image.3.6)):
* 
* **Рис. 3.6.**
* Да, кстати, вы заметили, что названия интерфейсов начинаются с буквы I? Эта традиция пошла из языка *Java*, и, как показывает практика, она весьма облегчает жизнь, если нужно, например, быстро разобраться в сложной диаграмме, составленной другим человеком.

**4.Диаграмма активностей: крупным планом**

### А ведь это вовсе не блок-схема!

Как мы уже говорили, *диаграммы активностей* (*Activity Diagrams*) являются представлением алгоритмов неких действий (активностей), выполняющихся в системе. Мы уже знаем, что *нотация* *UML* предлагает пять представлений системы:

* Вид системы с точки зрения *прецедентов*.
* Вид с точки зрения *проектирования*.
* Вид с точки зрения *процессов*.
* Вид с точки зрения *развертывания*.
* Вид с точки зрения *реализации*.

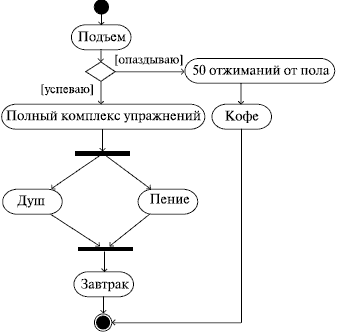
И при этом каждый из перечисленных способов представления системы может содержать последовательности действий, которые могут быть описаны с помощью алгоритмов. Вот здесь-то и выходят на сцену диаграммы деятельностей. Вообще говоря, любой элемент модели, имеющий динамическое поведение, может быть дополнен диаграммой деятельности - именно для уточнения этой самой динамики. Как хорошо подходящий *по* контексту пример следует упомянуть возможность применения диаграмм активности для описания бизнес-процессов, существующих в компании (нотации Grapes-BM, *BPML*/BPMN и др.). Вот уж где самая что ни на есть динамика!

Можно построить несколько диаграмм деятельности для одной и той же системы, причем каждая из них будет фокусироваться на разных аспектах системы, показывать различные действия, выполняющиеся внутри ее. Читатель, конечно же, понял, что, когда мы говорим о *динамике*, мы подразумеваем *поведение*системы в целом или ее частей. Говоря более формально, диаграммы активности, в общем-то, не имеют монополии на описание поведенческих особенностей динамических частей системы. Для этой же цели могут использоваться еще *диаграммы прецедентов*, последовательности, кооперации и состояний. Почему же мы говорим именно о диаграмме активности? Нет, не только потому, что так называется эта лекция.

Именно на диаграмме деятельности представлены переходы потока управления от одной деятельности к другой. Это, *по* сути, разновидность диаграммы состояний, где все или большая часть состояний являются некоторыми деятельностями, а все или большая часть переходов срабатывают при завершении определенной деятельности и позволяют перейти к выполнению следующей. Как мы уже говорили (повторение - мать учения), *диаграмма* деятельности может быть присоединена к любому элементу модели, имеющему динамическое поведение. Кстати, исходя из вышесказанного, логичнее говорить не "*диаграмма* деятельности", а "*диаграмма* деятельностей" - во множественном числе. А еще мы предполагаем, что читатель понимает смысл понятий "*деятельность*", "переход" и "*объект*". Об объектах как об экземплярах классов мы уже говорили ранее. Понятия же деятельности (*activity*) как протяженного во времени составного (неатомарного) вычисления (действия, action) и перехода как передачи контроля, надеемся, понятны интуитивно, без дополнительных объяснений.

Диаграммы деятельности позволяют моделировать сложный **жизненный цикл объекта**, с переходами из одного состояния (деятельности) в другое. Но этот вид диаграмм может быть использован и для описания динамики совокупности объектов. Они применимы и для детализации некоторой конкретной *операции*, причем, как мы увидим далее, предоставляют для этого больше возможностей, чем "классическая" *блок-схема*. Диаграммы деятельности описывают переход *от одной деятельности к другой*, в отличие от диаграмм взаимодействия, где акцент делается на переходах потока управления *от объекта к объекту*.

Как говорится, лучше один раз увидеть, чем сто раз услышать. Мы достаточно разрекламировали диаграммы деятельностей. Пора взглянуть на пример ([рис. 4.1](https://intuit.ru/studies/courses/1007/229/lecture/5958?page=1#image.4.1)).



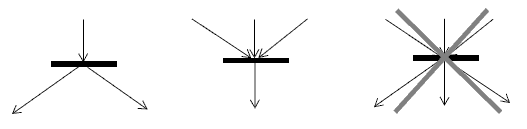
**Рис. 4.1.**

Эта *диаграмма* довольно точно описывает ежеутреннюю последовательность действий автора этих строк (до момента ухода на работу). Как видим, все очень просто и понятно. Действия показаны скругленными прямоугольниками, как в блок-схеме, - мы узнаем даже ромбик символа принятия решения с обозначениями условий возле переходов. Да, отличия от блок-схемы не так уж сильны. Более того, эти отличия выглядят как логичное расширение нотации блок-схем. Обратим внимание на то, что начало и конец уже не изображаются одинаковым безликим кружком. Начало теперь закрашено, а конец изображен в виде символа, напоминающего кошачий глаз ([рис. 4.2](https://intuit.ru/studies/courses/1007/229/lecture/5958?page=1#image.4.2)) (кстати, это образное название - "кошачий глаз" - уже намертво въелось в жаргон архитекторов и аналитиков).



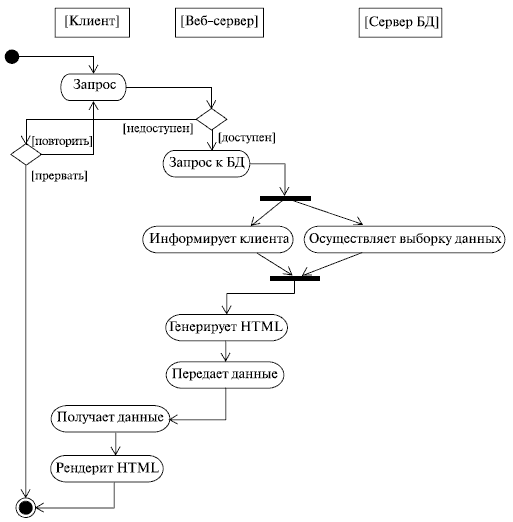
**Рис. 4.2.**

Без пояснений понятен также смысл символа, предшествующего принятию душа и пению и следующего за ними - он означает *распараллеливание*, а затем опять слияние воедино ( *синхронизацию* ) потоков управления, т. е. *операции* "пение" и "душ" выполняются *одновременно*. *Нотация* проста: несколько потоков управления сливаются в один или один *поток* разделяется на несколько. Третьего не дано ([рис. 4.3](https://intuit.ru/studies/courses/1007/229/lecture/5958?page=1#image.4.3)).



**Рис. 4.3.**

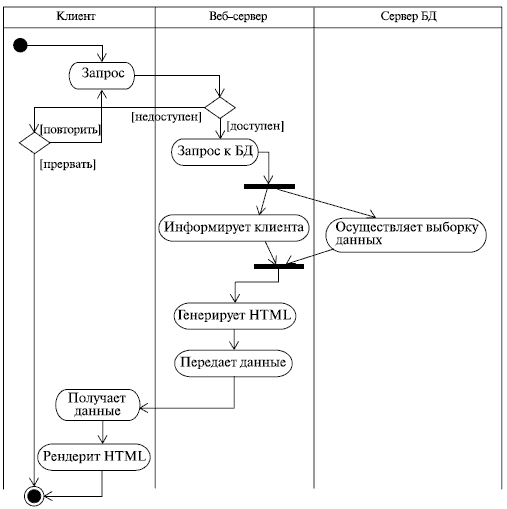
Конечно, это не единственные отличия *диаграммы активностей* от блок-схемы. На диаграмме деятельностей можно не только показать параллельно выполняемые действия, но и указать состояния объектов (так же, как и на представлениях конечных автоматов, о которых нам так много говорили в университетах), также есть возможность показывать распределение ролей и т. д. Вот еще пример, подтверждающий, что *диаграмма* активностей - это нечто большее, чем *блок-схема* ([рис. 4.4](https://intuit.ru/studies/courses/1007/229/lecture/5958?page=1#image.4.4)).



**Рис. 4.4.**

Смысл диаграммы вполне понятен и без дополнительных объяснений. Как вы уже, конечно, догадались, на ней показана работа с веб-приложением, которое решает некую задачу в удаленной базе данных. Привлекает внимание странное расположение активностей на этой диаграмме: они как бы разбросаны *по* трем беговым дорожкам, каждая из которых соответствует поведению одного из трех объектов - клиента, веб-сервера и сервера баз данных. Благодаря этому легко определить, каким из объектов выполняется каждая из активностей, и неожиданно приходит понимание того, что "странность" этой диаграммы, оказывается, очень упрощает ее восприятие.

*Аналогия* с дорожками действительно очень удачна. Именно таково официальное название элемента нотации *UML*, позволяющего указать распределение ролей на диаграмме активностей. Только дорожки это не беговые, а плавательные - они так и называются: *swimlanes*. Более формально, дорожка - часть области диаграммы деятельности, на которой отображаются только те деятельности, за которые отвечает конкретный *объект*.



**Рис. 4.5.**

Предназначены они для разбиения диаграммы в соответствии с распределением ответственности за действия. Имя дорожки может означать роль или *объект*, которому она соответствует. При использовании дорожек *нотация* слегка изменяется. Вот как, к примеру, выглядит *диаграмма* из предыдущего примера, перерисованная с использованием дорожек ([рис. 4.5](https://intuit.ru/studies/courses/1007/229/lecture/5958?page=1#image.4.5)).

**5.Диаграммы взаимодействия: крупным планом**

**Диаграмма взаимодействия**- это диаграмма, на которой представлено взаимодействие, состоящее из множества объектов и отношений между ними, включая и сообщения, которыми они обмениваются. Этот термин применяется к видам диаграмм с акцентом на взаимодействии объектов (диаграммах кооперации, последовательности и деятельности).

Несмотря на то величайшее уважение, которое мы питаем к Г. Бучу, это *определение* не кажется нам уж очень удачным. Хотя суть понятия оно передает. Наиболее важное *слово* в этом определении - это *слово* "сообщения". Действительно, как люди программирующие, мы понимаем, что взаимодействие-то как раз и состоит в обмене сообщениями между объектами! И к вопросу о сообщениях мы в этой лекции еще не раз вернемся. А пока же посмотрим, что Буч говорит дальше.

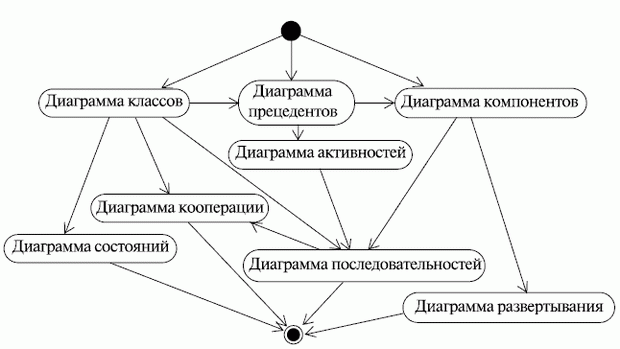
А дальше он объясняет, что такое диаграммы кооперации и последовательностей.

**Диаграмма последовательностей**- диаграмма взаимодействия, в которой основной акцент сделан на упорядочении сообщений во времени.

**Диаграмма кооперации**- диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения.

То есть *диаграмма последовательности* описывает (и именно поэтому так и называется) *последовательность*, в которой объекты отправляют и получают сообщения, а *диаграмма* кооперации - это аналог диаграммы последовательностей, который тоже показывает *обмен сообщениями* между объектами, но акцентирует внимание на *ролях*, которые объекты играют во взаимодействии. Эти два типа диаграмм вообще-то взаимозаменяемы, и решение, какую именно из них использовать в каждом конкретном случае, каждый проектировщик принимает исходя из личных предпочтений. Например, *автор* этих строк считает диаграммы последовательностей более понятным и более выразительным способом моделирования взаимодействий. Ваше мнение может быть противоположным.

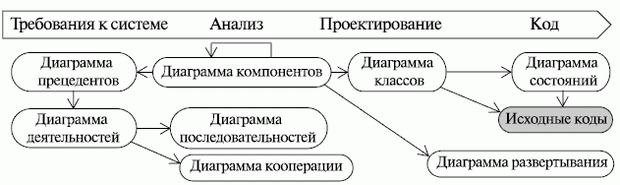
А какое же *место* *диаграммы взаимодействия* занимают среди других диаграмм *UML*? На этот вопрос можно ответить двояко. Можно просто говорить о построении диаграмм взаимодействия как об определенном этапе в процессе моделирования. А можно вспомнить о фазах жизненного *цикла* разработки *ПО* и посмотреть, где же *диаграммы взаимодействия* окажутся в таком случае. Да, кстати, кто помнит, какая *диаграмма* *UML* наилучшим образом подходит для описания процессов? Хм, что-то не видно леса рук... Ах да, видим одну руку - девушка, сидящая в дальнем углу зала, за колонной... Правильно! *Диаграмма активностей*. Что ж, попробуем нарисовать *диаграмму активностей*, описывающую процесс построения модели системы. Вот вариант такой диаграммы, предложенный одним из наших студентов ([рис. 5.1](https://intuit.ru/studies/courses/1007/229/lecture/5960?page=1#image.5.1)):



**Рис. 5.1.**

М-да, не совсем *диаграмма* и не совсем активностей. Но все же она показывает то, что мы хотели показать, а именно, что *диаграммы взаимодействия* строятся после того, как описана *структура системы* (*диаграмма* классов, *диаграмма* компонентов), способы ее взаимодействия с внешним миром (*диаграмма прецедентов*) и алгоритмы действий, выполняющихся в системе (диаграмме активностей). Это как бы последний штрих, уточнение того, как именно ведет себя система путем изображения взаимодействия объектов внутри ее.

Для того же, чтобы показать *место* диаграмм взаимодействия в жизненном цикле разработки *ПО*, нарисуем еще одну "псевдодиаграмму". Правильнее было бы сказать, что та *диаграмма*, которую вы сейчас увидите ([рис. 5.2](https://intuit.ru/studies/courses/1007/229/lecture/5960?page=1#image.5.2)), показывает, какие артефакты разработки документируются какими диаграммами.



**Рис. 5.2.**

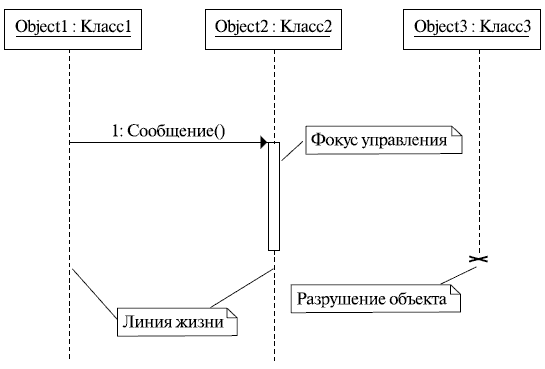
И опять все вроде бы логично - мы строим *диаграммы взаимодействия* во время анализа поведения системы. Кстати, из рисунка (сказать "*диаграмма*" язык не поворачивается) очень хорошо видно, что *диаграмма* последовательностей и *диаграмма* кооперации взаимозаменяемы и являются альтернативными друг другу шагами процесса.

**Диаграммы последовательностей и их нотация**

Вступительная часть этой лекции наконец-то закончилась, и мы с полным правом можем перейти к рассмотрению нотации диаграмм взаимодействия. Начнем с диаграмм последовательностей. Итак, мы уже говорили, что *диаграмма* последовательностей показывает последовательность, в которой объекты в процессе взаимодействия обмениваются сообщениями. Но как же сами объекты изображаются на такой диаграмме? А изображаются они точно таким же способом, каким мы пользовались ранее. Т. е. *объект* - это просто *прямоугольник*, внутри которого указаны подчеркнутые *имя объекта* и название класса (не обязательно), разделенные двоеточием. Объекты располагаются в верхней части диаграммы друг за другом. А вниз от каждого объекта тянется пунктирная линия, которую называют *линией жизни объекта*. Линия жизни объекта - это линия, которая изображает существование объекта на протяжении некоторого промежутка времени, и чем длиннее линия, тем дольше существует *объект*. Сообщения, которыми обмениваются объекты, изображаются в виде стрелок, направленных от линии жизни одного объекта к линии жизни другого. Линии жизни объектов, тянущиеся вниз, играют роль шкалы времени, так что сообщения, отправленные ранее, расположены выше, чем отправленные позже. Таким образом, последовательность сообщений легко читается "сверху вниз". Чуть позже мы еще вернемся к обсуждению сообщений и поговорим о том, каких видов они бывают и как их различить на диаграмме последовательностей. А пока же убедимся, что мы одинаково понимаем сам термин "сообщение": мы рассматриваем сообщение как спецификацию передачи информации от одного объекта к другому. *Объект* отправляет сообщение в расчете на то, что оно вызовет некую реакцию и за этим последует некоторая *деятельность*.

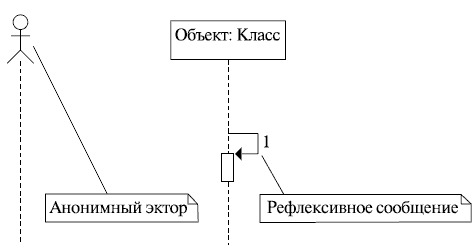
Еще одна вещь, которую можно увидеть на диаграммах последовательностей - это длинные прерывистые полосы на линиях жизни. Таким образом обозначаются периоды времени, когда *объект* имеет *фокус управления*, т. е. выполняет некоторое действие (причем неважно как - непосредственно или путем вызова некоей подчиненной *операции*). *Фокус управления* на диаграммах последовательностей часто не изображают: ведь и так понятно, где он должен располагаться, достаточно взглянуть на положение стрелок, изображающих сообщения. Рисовать фокус или нет - дело привычки каждого проектировщика. Впрочем, многие средства *UML*- моделирования рисуют фокус автоматически, так что человеку не нужно заботиться о его изображении. Если *объект* в процессе взаимодействия разрушается, этот факт помечают на его линии жизни крестиком, который, собственно, эту линию и заканчивает. Да, все мы смертны. Иногда так и тянется рука написать "R.I.P." рядом с таким крестиком...

Не полагаясь на выразительную силу и образность наших описаний, все же покажем примеры всех этих обозначений ([рис. 5.3](https://intuit.ru/studies/courses/1007/229/lecture/5960?page=1#image.5.3)).



**Рис. 5.3.**

А вот еще парочка обозначений. Первое из них - это *анонимный эктор*, которого изображают, если нужно показать использование объектов системы некоей *внешней сущностью* или абстрактным пользователем. Второе - это *рефлексивное сообщение*. Помните, что такое рефлексия? Правильно, самосозерцание! Тут, в принципе, происходит нечто подобное: *объект* посылает сообщение самому себе. Так рисуют, если нужно показать действие, выполняемое самим объектом (или внутри него), либо то, что *объект* сам себя вводит в некоторое состояние ([рис. 5.4](https://intuit.ru/studies/courses/1007/229/lecture/5960?page=1#image.5.4)).



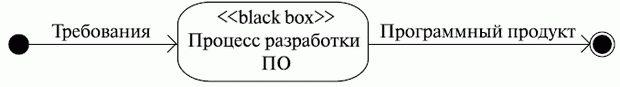
**Рис. 5.4.**

**6.Диаграммы прецедентов: крупным планом**

**Несколько слов о требованиях**

Итак, поговорим о требованиях. Что это такое, мы, в общем, понимаем - когда заказчик описывает нам, чего же именно он хочет, мы всегда слышим фразы типа "хотелось бы, чтобы проверка обновлений проводилась автоматически, как в антивирусах", "хочу большую зеленую кнопку в центре окна, которая начинает процесс", "*программа* должна позволять просматривать и печатать отчеты", "и чтоб красивенько все было, с полупрозрачностями, как в Висте", "при выходе должно выводиться подтверждение" и т. д. и т. п. Конечно, как настоящие разработчики, мы понимаем и то, что заказчик никогда не знает, что именно ему нужно, а если понимает, то объяснить не может. Но ведь фразы-то всегда, *по* сути, одинаковы! Они описывают, как заказчик представляет себе систему, чего заказчик хочет от системы, функциональность, которой он от нее ожидает, требования, которые к ней предъявляет.

Если обратиться к классикам, например, к той же "банде трех" (Якобсон, Буч, Рамбо), мы узнаем, что *требование - это желаемая функциональность, свойство или поведение системы*. Именно со сбора требований начинается процесс разработки *ПО*. Если изобразить процесс разработки *ПО* в виде " *черного ящика* " (уверены, читатель знает, что это такое, если нет - "Википедия" к вашим услугам), на выходе которого мы получаем *программный продукт*, то на *вход* этого "черного ящика" будет подаваться именно набор требований к программному продукту ([рис. 6.1](https://intuit.ru/studies/courses/1007/229/lecture/5962?page=1#image.6.1))!



**Рис. 6.1.**

Кстати, какую диаграмму напоминает этот рисунок? Правильно, *диаграмму активностей*. И выбор именно этой диаграммы тут абсолютно оправдан - помните, мы говорили, что *диаграммы активностей* часто используют для описания бизнес-процессов? Единственный нюанс: обычно процесс разработки не заканчивается с выпуском программного продукта - грядет новая *итерация*, новые, уточненные требования, новая версия и т. д.

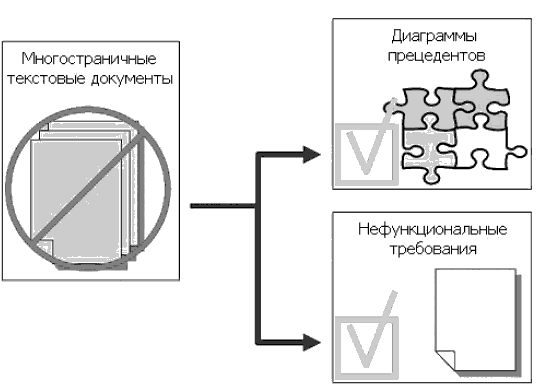
Кстати, вернемся к требованиям. Да, мы сказали, что на вход нашего "черного ящика" подается набор требований. Но в какой форме? Как их документируют, эти требования? Думаю, большинство читателей помнит, что такое *техническое задание* - основной документ, без составления которого не начинался в советские времена ни один проект. Документ это был большой, многостраничный, с четкой структурой, определяемой ГОСТами (государственными отраслевыми стандартами). И описывал он, *по* сути, не что иное, как требования к создаваемой системе!

*Техническое задание* - вещь *по*-своему хорошая. Но время шло, менялись стандарты, нотации, способы описания требований. И вот постепенно *техническое задание* уступило *место* набору артефактов, состоящему из документов двух видов:

* *диаграммы прецедентов*;
* нефункциональные требования.

*Диаграммы прецедентов* составляют *модель прецедентов* (вариантов использования, use-cases). *Прецедент* - это функциональность системы, позволяющая пользователю получить некий значимый для него, ощутимый и измеримый результат. Каждый *прецедент* соответствует отдельному сервису, предоставляемому моделируемой системой в ответ на *запрос* пользователя, т. е. определяет способ использования этой системы. Именно *по* этой причине use cases, или прецеденты, часто в русской терминологии фигурируют как *варианты использования*. Варианты использования чаще всего применяются для спецификации внешних требований к проектируемой системе или для спецификации функционального поведения уже существующей системы. Кроме этого, варианты использования неявно описывают типичные способы взаимодействия пользователя с системой, позволяющие корректно работать с предоставляемыми системой сервисами.

*Нефункциональные требования* - это описание таких свойств системы, как особенности среды и реализации, *производительность*, *расширяемость*, *надежность* и т. д. Часто нефункциональные требования не привязаны к конкретному варианту использования и потому выносятся в отдельный *список* дополнительных требований к системе ([рис. 6.2](https://intuit.ru/studies/courses/1007/229/lecture/5962?page=1#image.6.2)).



**Рис. 6.2.**

Но вернемся же к прецедентам (вариантам использования). Идентифицировать прецеденты и действующие лица - обязанность системного аналитика. И делает он это для того, чтобы:

* четко разграничить систему и ее окружение;
* определить, какие действующие лица и как именно взаимодействуют с системой, какой функционал (варианты использования) ожидается от системы;
* определить и описать в словаре предметной области (глоссарии) общие понятия, которые необходимы для детального описания функционала системы (прецедентов).

Подобный вид деятельности обычно выполняется в такой последовательности:

1. Определение действующих лиц.
2. Определение прецедентов.
3. Составление описания каждого прецедента.
4. Описание модели прецедентов в целом (этот этап включает в себя создание словаря предметной области).

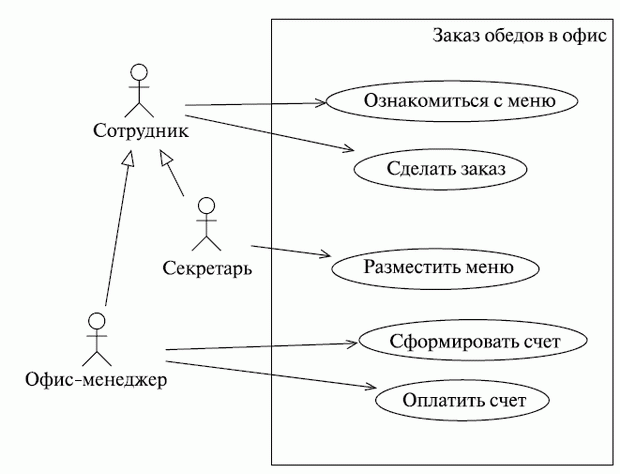
Вначале требования оформляются в виде обычного текстового документа, который создается или самим пользователем, или пользователем и разработчиком вместе. Далее требования оформляют в виде таблицы. В левую колонку помещают прецеденты, а в правую - действующих лиц, участвующих в прецеденте.

Рассмотрим пример. Секретарь размещает на сервере *меню* обеденных блюд на неделю. Сотрудники должны иметь возможность ознакомиться с *меню* и сделать заказ, выбрав блюда на каждый день следующей недели. *Офис*-менеджер должен иметь возможность сформировать счет и оплатить его. Система должна быть написана на *ASP*.*NET*. Такое вот нехитрое интернет-*приложение* для автоматизации заказов обедов в *офис*.

Думаем, здесь все понятно. *Таблица* с описанием требований может быть, например, такой:

|  |  |
| --- | --- |
| **Прецедент** | **Действующее лицо** |
| разместить меню | секретарь |
| ознакомиться с меню | сотрудник, секретарь, офис-менеджер |
| сделать заказ | сотрудник, секретарь, офис-менеджер |
| сформировать счет | офис-менеджер |
| оплатить счет | офис-менеджер |

Здесь нигде не сказано о том, что система должна быть написана на *ASP*.*NET*. Почему - понятно: это ведь нефункциональное требование! И еще, очевидно, что секретарь и *офис*-менеджер тоже являются сотрудниками. Читатель, внимательно прочитавший предыдущие лекции, заподозрит, что в данном случае, создавая модель прецедентов, говоря о действующих лицах, можно бы применить генерализацию. Действительно, *диаграмма прецедентов*, построенная на основе этой таблицы, может быть, например, такой ([рис. 6.3](https://intuit.ru/studies/courses/1007/229/lecture/5962?page=1#image.6.3)):



**Рис. 6.3**

**7.Обзор CASE-средств для построения диаграмм UML**

*UML* - отличное средство моделирования, но, как уже говорилось выше, строить диаграммы на бумаге - не всегда удобно, хотя бы *по* причине сложностей с редактированием, распространением и т. д. Чтобы облегчить труд проектировщика, были созданы CASE-средства - программы специального вида. CASE-средства помогут вам построить профессионально выглядящие диаграммы, даже если вы не в состоянии провести прямую линию на бумаге!

**CASE-средства**(от Computer Aided Software/*System Engineering*) - позволяют проектировать любые системы на компьютере. Необходимый элемент системного и структурно-функционального анализа, CASE-средства позволяют моделировать бизнес-процессы, базы данных, компоненты программного обеспечения, деятельность и структуру организаций. Применимы практически во всех сферах деятельности. Результат использования CASE-средств - оптимизация систем, снижение расходов, повышение эффективности, снижение вероятности ошибок.

Существует немало подобных программ. Выбор CASE-средства "*по* себе" - личное дело каждого читателя, и мы ни в коей мере не собираемся влиять на него. Мы лишь попытаемся предоставить ему этот выбор, рассмотрев некоторые наиболее достойные внимания, с точки зрения авторов, CASE-средства для построения *UML*-диаграмм. Причем постараемся рассказать и о признанных лидерах рынка, и о его "аутсайдерах", и о коммерческих "монстрах", и о "легких" программах с открытым исходным кодом. И начнем, пожалуй, с пакета, являющегося фактическим стандартом в области *UML*-проектирования.

**IBM Rational Rose**

Rational Rose - современное и мощное средство анализа, моделирования и разработки программных систем. Rational Rose пригодится при решении практически любых задач проектирования информационных систем: от анализа бизнес-процессов до *кодогенерации* на определенном языке программирования. Такой арсенал позволит не только спроектировать новую систему, но и доработать старую, произведя процесс *обратного проектирования*.

Для того чтобы наиболее полно покрыть весь сегмент рынка средств проектирования и разработки, выпускается несколько версий продукта:

* **Rational Rose Modeler**

Эта версия позволит аналитикам и проектировщикам проводить анализ бизнес-процессов и проектировать систему. Данная редакция, увы, не поддерживает *кодогенерацию*.

* **Rational Rose Professional**

Как видно из названия, это профессиональная редакция продукта. В зависимости от выбранного языка программирования позволяет выполнять прямое и *обратное проектирование*. Rose Professional заказывается только в определенной конфигурации (например, Rose Professional С++ или Rose Professional С++ DataModeler). Rational Rose Professional, конечно, не создает 100 % исполняемого кода. На выходе разработчик получает каркасный код информационной системы на определенном (заказанном) языке программирования, который впоследствии нужно еще программировать и программировать. Продукт нацелен и на аналитиков, и на разработчиков.

* **Rational Rose RealTime**

Версия продукта, созданная специально для получения 100 % исполняемого кода в реальном масштабе времени. Конечно, RealTime позволяет проводить прямое и *обратное проектирование* на языках С или С++. По заверениям разработчиков, на выходе модель автоматически компилируется и собирается в исполняемый файл. Само собой, продукт предназначен именно для разработчиков.

* **Rational Rose Enterprise**

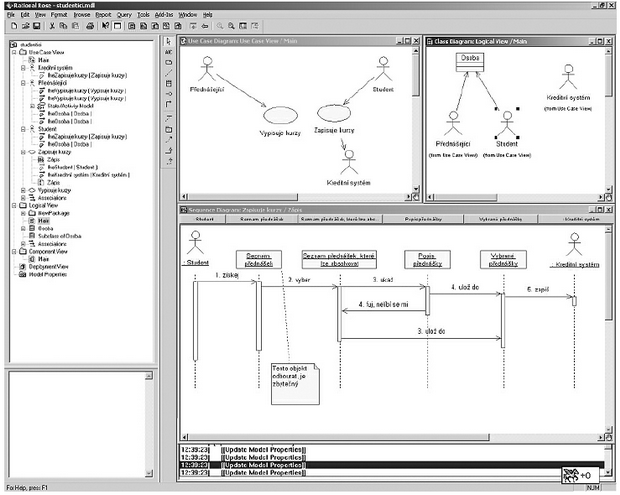
Абсолютно полная версия. Поддерживаются все функции других редакций, за исключением возможности 100 % *кодогенерации*. Таким образом, эта версия продукта покрывает весь спектр задач по проектированию, анализу и *кодогенерации*. Это программный пакет для всех участников проекта.

* **Rational Rose DataModeler**

Это не конкретный вариант продукта, а функциональность по проектированию баз данных. Функции DataModeler входят в состав Rose Enterprise или Professional.

К сожалению, нет бесплатной версии продукта, но **для образовательных учреждений все программное обеспечение IBM доступно бесплатно** (для использования в учебных целях) в рамках программы *IBM* Academic Initiative.

А как же выглядит это чудо? Не слишком изысканно, но вполне функционально - судите сами ([рис. 7.1](https://intuit.ru/studies/courses/1007/229/lecture/5963?page=1#image.7.1)):

[](https://intuit.ru/EDI/23_04_17_1/1492899714-28128/tutorial/356/objects/7/files/07_01.gif)

[увеличить изображение](https://intuit.ru/EDI/23_04_17_1/1492899714-28128/tutorial/356/objects/7/files/07_01.gif)  
**Рис. 7.1.**

В зависимости от поставки, в Rational Rose может быть расширен или сужен набор визуальных *компонент* (возможных диаграмм). Впрочем, Rational Rose и так достаточно функционален. Вот основные возможности продукта:

* прямое и *обратное проектирование* на языках: ADA, Java, С, C++, Basic;
* поддержка технологий COM, DDL, XML;
* возможность генерации схем БД Oracle и SQL.

Также Rational Rose имеет открытый *API*, позволяющий самому создавать модули для других языков программирования. На рынке уже имеется достаточное число модулей для популярных языков программирования и RAD-систем, таких как Delphi, ErWin, Jbuilder, VisualCafe, Jdeveloper, VisualAge SmallTalk. Одна из ведущих компаний в области создания дополнительных модулей - *Ensemble* Systems (http://www.*ensemble*-systems.*com*/).

Rational Rose много раз признавалось различными изданиями лучшим средством проектирования. Вот только некоторые из них ([рис. 7.2](https://intuit.ru/studies/courses/1007/229/lecture/5963?page=1#image.7.2)):

[](https://intuit.ru/EDI/23_04_17_1/1492899714-28128/tutorial/356/objects/7/files/07_02.gif)

[увеличить изображение](https://intuit.ru/EDI/23_04_17_1/1492899714-28128/tutorial/356/objects/7/files/07_02.gif)  
**Рис. 7.2.**

Если вы программировали в MS *Visual Studio* 6.0, то, возможно, вы уже познакомились с одним из продуктов семейства Rational Rose, поскольку в этот пакет встроен *Visual* Modeler - усеченный вариант Rational Rose 98. С помощью *Visual* Modeler можно рисовать *диаграммы классов* в трех различных нотациях - нотации Буча, ОМТ и на *UML*. *По* диаграммам классов можно провести генерацию каркасного кода (на C++, VB или *Java*). Такая генерация программного кода называется прямым проектированием (*forward engineering*). Взаимозависимости классов, изображенных на диаграмме классов, отображаются в программном коде.

Большой интерес представляет *обратное проектирование* (*reverse engineering*), когда *по* исходному коду восстанавливается *диаграмма* классов, позволяющая понять структуру программы. Это тоже можно делать с помощью *Visual* Modeler, причем на основе Microsoft *Foundation* Classes (*MFC*)! К ограничениям *Visual* Modeler относится тот факт, что он не поддерживает *диаграммы развертывания*, описывая лишь внутреннюю функциональность создаваемой системы.

Также Rational Rose интегрируется с *Visual* *Component* *Manager*, репозиторием Microsoft *Repository*, системой управления версиями Microsoft *Visual* SourceSafe и Rational ClearCase. Плюс многое-многое другое...

Конечно, можно еще долго петь дифирамбы этому продукту, являющемуся, *по* сути, стандартом де-факто в области *UML*-проектирования (с субъективной точки зрения авторов, этот продукт не слишком интуитивен и удобен, хотя, без сомнения, сверхфункционален). Честно говоря, как ни парадоксально это звучит, особого впечатления на авторов этот продукт не произвел, возможно, *по* причине недостаточного с ним знакомства. Думаем, что сами разработчики расскажут о своем продукте гораздо лучше. Предоставим же читателю возможность оценить этот продукт, основываясь на информации "из первых рук"! Вы можете это сделать, посетив такие сайты:

[http://www-306.ibm.com/software/rational/](http://www-306.ibm.com/software/rational/%20)Это официальный *сайт* Rational, где вы сможете найти информацию о Rational Rose и других продуктах Rational (на англ. языке). Также можете попробовать сходить *по* "старому адресу" - [www.rational.com](http://www.rational.com/).

[http://interface.ru/](http://interface.ru/%20)*Сайт* компании "*Интерфейс*". Как уверяют авторы ресурса, на сегодняшний день здесь собран самый большой (в Рунете) *архив* информации *по* продуктам Rational. Здесь можно найти множество статей, обзоров, руководств и описаний "*по* теме" и на русском языке.

[http://sunset.usc.edu/cse/](http://sunset.usc.edu/cse/%20)А здесь можно найти неплохой *архив* презентаций и статей *по* темам программной инженерии, в том числе и о Rational Rose (на англ. языке)

**8.Заключение**

Ну, вот вы и перевернули последнюю страницу последней главы. Надеемся, что это первое знакомство с *UML* не отпугнуло вас от дальнейшего углубленного изучения языка. Все, что мы хотели рассказать читателям, мы рассказали. Конечно, многое осталось за рамками этой книги, но о многих видах диаграмм и элементах нотации мы умолчали намеренно: ведь цель книги состояла не в том, чтобы дать еще одно полное описание *UML*, а в том, чтобы рассказать только о тех элементах *UML*, знать о которых новичку абсолютно необходимо.

Возможно, это слишком смелое предположение, но мы все же позволим себе надеяться, что после прочтения этой книги большинство читателей научились читать *UML*-диаграммы и почувствовали в себе способности к построению простых *UML*-моделей. Более того, мы будем искренне рады, если эта книга поможет читателям сдать тест UM0100 и получить статус OCUP *Fundamental*. В таком случае мы будем считать, что книга исполнила свое предназначение, а *автор* трудился не напрасно, и его попытка передать свой *опыт* читателю удалась. Кстати, не надейтесь, что на этом мы с вами прощаемся, - в приложении к книге вас ждет подробная *информация* о программе OCUP и экзамене UM0-100, примерные вопросы, описание процедуры тестирования, *список* шпаргалок по *UML* и глоссариев *UML*-терминов. Обязательно загляните туда - не пожалеете!

Человеку свойственно ошибаться, и, как и любой достаточно большой текст, эта книга содержит ошибки. Это могут быть ошибки самого разного рода: опечатки, заблуждения, субъективные суждения... Если вы обнаружили нечто подобное - смело пишите по адресу alexander.v.babich@acm.*org*. Мы будем благодарны за любые вопросы, уточнения и замечания. Они не только помогут автору в работе над последующими изданиями этой книги (если таковые будут), но и могут также помочь и нам, и вам лучше разобраться в предмете.

Что ж, на этом все. Спасибо, что перебороли в себе праведное желание швырнуть эту книгу в мусорную корзину и дочитали эту писанину до конца. Возможно, мы еще встретимся на страницах других книг. Удачи вам всем! Удачи в сдаче экзамена. Удачи в использовании *UML* и в профессиональной деятельности. Удачи... И пусть вас никогда не постигнет "*паралич* аналитика", подобный показанному в этом чудесном комиксе от Laurent Gregoire (<http://tnerual.eriogerg.free.fr/0xBABAF000L/index.html>)!

