КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Факультет вычислительной математики и кибернетики

УТВЕРЖДАЮ:

Научный руководитель, Декан факультета ВМК, профессор

_____ Р.Х.. Латыпов

«_15_» ___декабря_____ 2010 г.

М.П.

МЕТОДИЧЕСКОЕ ПОСОБИЕ

по договору № 0002/23/370 от 1 января 2010 года.

по теме: Методическая разработка специального курса

«Объектно-ориентированный анализ и проектирование»

Казань-2010

Методическая разработка специального курса

«Объектно-ориентированный анализ и проектирование»

Н.Р.Бухараев, к.ф.-м.н, доцент факультета ВМК

Современный этап развития разработки программного обеспечения характеризуется стремлением осознать последствия произошедшего за короткий период времени (1970-1980 гг.) «скачка» - перехода от «программирования в малом», создания еще небольших программных систем в основном научно-исследовательского характера, к «программированию в большом», реализации крупных проектов по разработке программных систем масштаба предприятия.

Этот переход в новое качество породил целый ряд новых IT-дисциплин, еще недостаточно полно представленных в университетском курсе обучения (что неизбежно сказывается на профессиональной квалификации выпускников). К ним относится и рассматриваемая в данном пособии дисциплина объектно-ориентированного анализа и проектирования (ОО АП).

Несмотря на обилие литературы на данную тему - в том числе, авторизованных курсов именитых вендоров, большая ее часть ориентируется уже на зрелых IT-специалистов, а не студентов-старшекурсников или молодых преподавателей классического университета. Пособие дает введение в проблематику и основные принципы ОО АП - ссылаясь, но разумеется не включая во всей полноте то практически ценное, что уже накоплено здесь в области IT-образования. Его основная цель – обеспечить преемственность подготовки, связав уже достигнутое студентом с его будущей профессией. Иными словами, автор стремился пробудить интерес и сформировать ответственное отношение к крайне сложным задачам и решениям в сфере современных информационных технологий - с опорой на уже имеющиеся у него практический опыт «программирования в малом» и познания в области фундаментальных математических дисциплин (зачастую, крайне мозаичные).

СОДЕРЖАНИЕ.

Введение в унифицированный язык моделирования UML.

Азы ОО АП - объектно-ориентированного анализа и проектирования больших программных систем - *по-человечески*.

Глава 1. Зачем мне математика? Главная проблема современного программирования.

Опять скажу: никто не обнимет необъятного!

Козьма Прутков, афоризм 160.

Появление языка моделирования Unified Modeling Language (UML) в конце 1990-х гг. – одно из наиболее важных событий в эволюции методов разработки программного обеспечения (ПО). Именно в это время создание программных систем переходит в новое качество, перерастает из алгоритмики, «программирования в малом» в инженерию разработки ПО - «программирование в большом».

Глубина этих перемен столь велика, что даже изначально фундаментальное, предопределившее само возникновение компьютерного программирования понятие абстрактной и формальной (т.е. *математической*) модели приобретает качественно новый смысл. Программирование – по-прежнему, род математического моделирования. Но теперь моделирование понимается уже не только как «схематическое описание», но и «средство минимизации (снижения) будущих рисков». В главном, это и отличает современную разработку ПО от прежнего программирования.

Для осознания преемственности и различий - вспомним. Формирование в течение всего 19 века трудом многих поколений математиков понятия математической модели было реакцией на проблемы, связанные с появлением т.н. *нестандартных теорий*. Первым и важнейшим примером такой теории была нестандартная геометрия Н.И. Лобачевского. В то время они воспринимались как *внутренние* проблемы построения надежных оснований самой математики. Этот важнейший для дальнейшего возникновения компьютерного программирования этап завершился в начале 20 века

- формальным определением понятия *алгоритм* в качестве абсолютно надежного математического метода (тезис Черча-Тьюринга) и
- признанием невозможности решать все проблемы математики лишь такими методами (результаты о неразрешимости, теорема Геделя о неполноте).

Спустя век, мы снова говорим о проблемах надежности моделирования, но теперь уже – с точки зрения их внешних последствий. Для понимания того, о каких именно рисках теперь идет речь, вспомним ставший уже классикой пример о разнице в подходах к построению собачьей конуры и небоскреба от авторов UML [Booch,Jacobson,Rumbaugh]. Первое возможно лишь при понимании на уровне некой общей идеи, интуиции – второе требует фиксации технологии, самого серьезного отношения к организации работ. Мыслить устройство конуры мы можем сколь угодно изощренным – суть не в этом. Цена неудачи при строительстве жилого дома выше. Намек прозрачен. Строя абстрактные схемы – нужно заранее думать о тех людях, на головы которых могут обрушиться построенные нами дома и созданные нами программные системы.

Какие бы мощные (а потому более сложные в освоении) средства и инструменты мы при этом не использовали сами для выполнения этой тяжелой работы, главной целью остается долгосрочное облегчение жизни *всех* людей. Однако, как учит история - вопреки начальным ожиданиям, цель эта

постоянно оказывается крайне труднодостижимой. Потому – *спеши медленно.* Подход UML подкрепляет эту древнюю мудрость крайне полезными на практике рекомендациями.

Следуя этому принципу сам, автор решил отказаться от описания конкретной программной среды поддержки UML. *Роза пахнет розой – хоть розой назови ее, хоть нет* [1]. Во-первых - потому, что для рассматриваемых здесь игрушечных задач подойдет любая среда программирования, с которой читатель вполне может освоиться самостоятельно. Во-вторых - потому, что перед использованием программной среды нужно тщательно разобраться в том, что нового можно от нее ожидать – и что ждать бесполезно.

В знак уважения к пионерам (впереди идущим), достаточно сказать, что первой UML-средой стала Rational Rose компании Rational (впоследствии – IBM Rational), за которой последовал целый ряд все более мощных и специализированных инструментов под общим названием Rational знаменитой компании IBM. Ну и конечно – качественные программные продукты других, не менее знаменитых компаний и менее известных разработчиков, в том числе и свободно распространяемые.[2]

Как и в случае появления принципиально новых языковых средств программирования до него, создание UML обозначило некий новый комплексный подход к решению *проблемы масштабирования*, т.е. UML стал

- *реакцией на* качественно возросшую *сложность* (кризис разработки больших программных систем 1970+ гг.) в виде
- *формализации* - фиксации в языке уже хорошо оправдавших себя на практике способов распределения сложности труда разработчиков ПО,
- отражающих по мере возрастания сложности все более высокие уровни *абстрагирования*.

Проще говоря, методы программирования с возрастанием сложности разработки ПО *математизируются.* И это не столько новая, своеобразная математика. Просто серьезные разработчики – довольно своеобразные математики. С этим своеобразием нам и предстоит познакомиться.

Но все же UML – язык моделирования, а не язык программирования. Его цели более опосредованы, общи и многофункциональны. Объединив результаты предыдущих и своих собственных усилий, авторы UML Гради Буч, Джеймс Рамбо и Ивар Якобсон стремились в достаточно компактной форме вобрать и развить лучшее из богатого опыта формального моделирования. Как пишут сами авторы в [Booch,Jacobson,Rumbaugh],

UML - это язык.

[В частности,]
Это язык для

- **визуализации,**
- **специфицирования,**
- **конструирования и**
- **документирования**

артефактов программных систем.

---

[1] Впрочем, стоит перечитать *Ромео и Джульетту*, чтобы понять, что истинный смысл цитаты относиться не к собственно розам. Даже - рациональным или виртуальным. Главная тема трагедии обозначается в самом ее начале: *Две равно уважаемых семьи в Вероне, где встречают нас события…*

[2] Слушателям данного курса доступны лицензионные программные продукты компаний IBM и Microsoft – для этого достаточно обратиться к преподавателю курса (автору данного пособия)

Здесь:

Артефакт (от латинского *artefactum)* — искусственно созданное, сделанное, любой результат труда людей. Термин дает нам недвусмысленный намек на то, что программный код – целевой, но далеко не единственный результат разработки.

А также – намек на специфику профессии. По всей видимости, термин пришел в IT из археологии. Археолог изучает исторические пласты - следы, оставленные предыдущими эпохами. Он видит в статике динамику, в последовательности следов (трасса, trace) – «застывшее» время. Программист – то же.

Визуализация – наглядное представление, ориентированное на максимально широкий круг людей, в большей или меньшей степени отвечающих за процесс разработки (аналитик, архитектор, разработчик, тестер,… заказчик, пользователь). В данном случае – диаграммы UML.

Специфицирование – уточнение начала разработки – согласование постановки задачи в более строго формализованной (по возможности и необходимости) форме. См. далее *анализ*.

Конструирование – подготовка продолжения разработки – схем решения «верхнего уровня» (логики решения). Включая возможность частичной автоматизации (например - генерации программного кода по UML диаграммам). См. далее *проектирование*.

И все же один менее очевидный для «технически мыслящих» специалистов, но крайне важный для понимания момент в использовании UML я хотел бы подчеркнуть особо - его социальную функцию. Любой язык предназначен для общения. UML – не регламент работ и не свод жестких правил, предписывающий людям те или иные роли, те или иные алгоритмы поведения. В первую очередь, это язык для рабочего общения *между людьми* в процессе разработки ПО – весьма полезный, хотя и не строго обязательный.

Принципиально новым на современном этапе развития IT становится постепенный перенос внимания с множества ситуаций типа «человек-компьютер» (разработчик-компьютер, пользователь-компьютер и т.п.) на поддержку организации деятельности больших коллективов. По сути своей, UML и есть результат первой серьезной попытки создать цельную картину – мозаику из множества таких кусочков. Зачем?

Создание больших программных систем требует больших коллективов (команд) и специализации - разделения труда, «распределения ролей» - обязанностей, связанных с решением тех или иных задач, выполнением тех или иных функций. В конечном счете, это – распределение ответственности. Что порождает проблемы взаимопонимания – как внутри самой команды, так и в ее общении с окружением. Ответы на эти старые проблемы нужно искать в более современных подходах к организации взаимодействия людей, в существенной степени опирающихся на самоорганизацию. Они-то и лежат «за схемами» UML.

Словом – предмет разговора настолько серьезен, что трудно обойтись без шутки. Никто не идеален – в том числе сам UML. Но есть предельно простой способ проверки эффективности применения подхода UML - до знания деталей самого UML! Если данный подход не помогает поддержанию нормальной температуры (36.6°) в команде (не придает дополнительных стимулов участников к плодотворной совместной работе, а лишь вызывает горячие конфликты и охлаждение отношений) – то, скорее всего, он понимается и используется неверно. Во всяком случае, тогда UML «не работает».

Рекомендация. Тщательно выбирай аксиомы совместного существования. Не противопоставляй изначально «они и мы», особенно - «мы и *эти*». Говори сначала – «мы все [решим проблему лишь сообща]». Эту тему мы – по умолчанию - и выберем в качестве «сквозной нити» нашего рассказа о UML.

Читателю не стоит ожидать от данного пособия слишком многого. Оно мыслилось автором как первоначальное знакомство с предметом - введение в

проблематику и базовые концепции «программирования в большом». С ориентацией на людей, уже имеющих достаточно большой опыт «программирования в малом». И - с учетом бурного развития этой, далеко еще не во всем устоявшейся области.

Пособие не содержит полного описания UML и практического руководства по его использованию – но лишь общее виденье основных проблем и принципов разработки, дающих начальную мотивацию к более глубокому изучению предмета. Ссылки на полезные для этой цели ресурсы читатель найдет в тексте самого пособия и списке литературы.

Лучше сделать меньше, да лучше – так надежнее. Нельзя объять необъятное - нельзя на заведомо небольших примерах быстро научиться качественно разрабатывать большие долгосрочные проекты. Больше того - заранее предопределить для другого решения проблем ***до*** возникновения этих проблем в его собственной практике не просто невозможно. Такие «благие намерения» чаще оборачиваются «медвежьей услугой» - не упрощая, но запутывая суть дела. Особенно в разработке ПО, где смена концепций сегодня еще часто путается со сменой обозначений.

Автор – без претензий на большой практический опыт и доскональное знание области - видит свою задачу скромней. Он хотел бы показать читателю, что некоторые мелкие неудобства (дискомфорт), которые мы встречаем в своей индивидуальной работе уже *сегодня* - не придавая им особого значения - рискуют обернуться крупными неприятностями. Завтра. Но чтобы завтра было не хуже, чем сегодня - стоит заметить их уже сейчас. Словом, выбор прост и однозначен, хотя и далеко не легок в реализации. Строить сегодня конуру нужно вдумчиво, терпеливо и тщательно, *как если бы* мы строили небоскреб. Не наоборот. Даже (особенно!) когда речь идет о программных системах – небоскребах как бы не настоящих, а «виртуальных», т.е. находящихся в нашей голове.

Главная проблема современного программирования – **проблема надежности** ПО. Общие и профессиональные подходы к ее решению реализуются теми или иными специальными методами программирования – локальными, частными, в том числе формальными и *отчасти* автоматизируемыми (инструментарием). Впрочем, и самые специальные методы имеют в своей основе вполне общечеловеческое происхождение и цену.

Надежное (reliable) ПО – это ПО, создаваемое надежными *людьми*. Надежен (reliable) тот, на кого можно положиться, кто оправдывает возложенные на него надежды. И надежный не может все знать и уметь. Но он точно знает, что «*серебряной пули*» – универсального способа быстрого и окончательного решения всех проблем – не существует. Узкий специалист знает частные профессиональные решения, но берется за все – в надежде на то, что инструменты сделают за него главное. Надежный профессионал знает и общие проблемы. Он благодарен другим людям за инструменты – но берется делать лишь то, за что впоследствии сможет ответить сам. Именно поэтому надежные тебя не подведут. На них можно положиться, на них можно надеется.

Пользуясь возможностью, автор благодарит

- координатора академических программ компании IBM Алексея Полунина;

  он сыграл роль дантовского Вергилия, введшего автора в сложный мир проблем объектно-ориентированного анализа и проектирования. На таких проводников можно положиться, несмотря на устрашающие надписи на входе («Оставь надежду всяк сюда входящий»);

- выпускницу факультета ВМК КГУ Анастасию Сабирзянову;

  ее энергия и энтузиазм в стремлении объединить теоретические познания с решением самых актуальных задач организации производства не устают восхищать автора. На такую

молодежь можно без боязни возлагать надежды. В данном пособии ей принадлежит методическая разработка примера разработки ПО (глава 4);

- компанию IBM - за программные средства и методические материалы, предоставленные в рамках программы « IBM Academic Initiative»

По факту текущей реализации курса, данное пособие предваряет освоение богатого содержания материалов курсов «IBM Rational University». Они содержат многочисленные ценные рекомендации, примеры и лабораторные работы - но при этом ориентированы в большей степени на существующих, а не будущих IT-профессионалов. Помимо указанных материалов и указанных в списке литературы книг, в пособии использованы также материалы Википедии и иных свободно распространяемых интернет-ресурсов.

Но ответственность за конечный результат, разумеется, лежит на авторе пособия. [3] Особенно - в части особенностей трактовки проблематики, вызванных желанием сохранить преемственность как в отношении структуры уже имеющихся курсов IT-обучения, так и традиций университетского образования в целом.

---

[3] В рамках, определяемых популярной шуткой. Преподаватель – преподавателю: «Ну и глупые же студенты попались. Объяснял, объяснял – уже сам все понял, а они все никак не поймут!»

Глава 2. Перед UML. Программирование как моделирование: базовые понятия. Жизненный цикл разработки (lifecycle). Виденье (vision), виды (views) и деятели (actors).

Уже в ходе накопления опыта «программирования в малом», индивидуального создания небольших программ, мы начинаем – каждый сам по себе, но с опорой на опыт предыдущих поколений программистов – выделять специальные виды и этапы деятельности. По мере возрастания сложности задач, такое разделение времени работ и самой работы становиться разделением труда – порождая новые специализации, профессии, роли.

Такое разделение труда разработчиков фиксируется в фундаментальном, крайне важном для понимания современного этапа развития области понятии *жизненный цикл разработки ПО*.

Здесь принципиально важно понимать, что изначально мы совмещаем в себе все возможные, в том числе профессиональные роли – и только потому способны понимать других людей хотя бы *в принципе*. Хотя, разумеется, такое совмещение часто происходит в голове, не в реальности. «Папа» физически не может стать «мамой» - просто у каждого человека была своя мама. С другой стороны, «рыбак рыбака видит издалека» - мы лучше понимаем тот вид труда, в котором сами имеем больший опыт работы.

Важно отметить и то, что исторически - разделение ролей между участниками разработки не предопределено извне, изначально. Роли выделяются по мере возрастания сложности работы. Оттого в разных определениях жизненного цикла можно встретить разное количество ролей.

Изначально, основным занятием разработчика ПО мы обычно считаем программирование в узком смысле программной **реализации/implementation** - записи, кодирования, формализации алгоритма (правила изменения данных) на конкретном языке программирования.

Все хорошо – пока и если «все идет по плану». Но любые планы идеальны – предполагая по умолчанию некий позитивный, предпочтительный для нас сценарий развития событий. Человек полагает – жизнь располагает. Оттого и сами планы впоследствии изменяются в зависимости от того, насколько хорошо они реализуются на практике.

Разделение труда, конечно же – не «серебряная пуля». Иначе говоря, это не универсальный и даже тупиковый путь борьбы со сложностью – если не сопровождается повышением квалификации, качества работы на каждом ее этапе. Параллельное специализации возрастание внутренней сложности этапа программирования отражается в эволюции языков программирования. Общеметодологический принцип структурирования (разделение целого на части, затем прямой и обратный переходы от целого к частям) своеобразно отражается в различных подходах, парадигмах программирования.

Любая модель содержит описание статики и динамики – состояний и изменений некоторого объекта внимания. Не обязательно некоторой вещи, материального объекта, но также ситуаций, связей, процессов… Проще говоря - всего того, что нам необходимо описать.

Что здесь считать основным, изначальным? Так, если процедурный подход фокусируется (выделяет в качестве изначального объекта внимания) на описании сложной динамики, изменений (структуры данных появляются здесь позже), то программирование баз данных отталкивается от сложности статики, данных (здесь позже появляется понятие процедуры).

Объектно-ориентированный подход – принятый также и в UML – изначально ориентируется на разбиение описания статики и динамики моделируемого объекта на компактный набор взаимосвязанных подмоделей, содержащих совместное описание данных (свойства, properties) и

процедур (методы, methods). Одновременно, в двух проекциях (ракурсах, планах, аспектах): более абстрактной, ранней и общей (классы) и более конкретной, специфичной и поздней (объекты).

Далее мы предполагаем в качестве предусловия

1) *практическое* знакомство читателя с основными принципами ООП и его
2) начальные *теоретические* познания современной математики – *неформальное* знакомство с основными идеями теории автоматов, теории графов, теории множеств и математической логики.

Естественно – лишь в рамках предыдущего университетского курса.

Но разработчики – люди; все люди не идеальны и допускают ошибки. **Тестирование/testing** – проверка программ путем поиска и исправления ошибок программной реализации – первый спутник программирования. Как и любое другое дело, такая проверка по мере возрастания сложности требует особых знаний и использования специальных, лишь отчасти автоматизируемых методов.

Как уменьшить объем тестирования? Помимо внутренних ошибок алгоритмизации и программной реализации в целом, существенные причины ошибок лежат извне. В том числе, в самом начале – в неточной постановке задачи моделирования. По факту, эта предварительная перед собственно программированием стадия чревата целым рядом ошибок и неточностей, вызванных

- Недостатками концептуальной модели (**business model**) - содержательного описания конкретной предметной области (domain) в целом, в ее статике и динамике (бизнес-процессов)
- Недостатками запроса, формулирующего требования к функциональности и иным качествам ПО (**requirements**) - описания того, что же именно мы собираемся автоматизировать.

Традиционно, мы разделяем в описаниях ошибки содержательные и формальные – ошибки экспертизы, принципиального незнания необходимого содержания предмета описания и ошибки выражения, неумения точно выразить такое знание.

Отсюда – взгляд новичка на формальное и абстрактное моделирование (математику) как занятие бессодержательное и оторванное от жизни. Она оторвана от конкретики лишь постольку, поскольку мы не можем заведомо знать и уметь решать все конкретные проблемы практики. Но мы можем к ним подготовиться теоретически. И должны – поскольку есть ошибки… и ошибки. Ошибки, допущенные на ранних этапах – самые опасные.

Отметим также и то, что, как правило, разработка ПО не имеет (не может иметь) дело с «поэзией» - плохо определенными предметными областями (доменами), плохо известными и/или плохо описанными закономерностями. Это не означает, что «поэзия» никому не нужна - это означает лишь, что многие области хуже нами поняты и описаны. Но по факту, в разработке ПО мы чаще сталкиваемся с иной ситуацией – мы «слишком» много знаем. Т.е. многое понимаем интуитивно, но затрудняемся выделить и зафиксировать главное – наиболее актуальное, одновременно нужное и *выполнимое* на данном этапе.

Этот достаточно новый момент в понимании *абстрагирования* - выделение главного, отвлечения от деталей - как способа борьбы со сложностью стоит отметить особо. Все так делают, строя свои планы – не все люди закладываются на ошибки начальных планов и предусматривают возможность возврата, «отката» (rollback). Оттого (в частности) надежность *не является* объективной характеристикой самого ПО. Важность этого момента обусловлена и тем, что

традиционные области применения ПО (точные науки, юриспруденция, бухгалтерский учет и т.п.) расширяются сегодня до крайне ответственного уровня [крупного] *бизнеса* – моделирования деятельности больших предприятий со сложной внутренней организацией. Проще говоря – результаты нашей работы начинают все более затрагивать все большее количество людей. Чем далее, тем более.

По своему происхождению и назначению UML тесно связан именно с разработкой систем «масштаба крупного предприятия». Это заметно по используемому словарю (глоссарию) экономических терминов. Особенно, по основному понятию *сервиса,* предоставления услуг – т.е. помогающих решению некоторой задачи клиента дополнительных возможностей. На основе договора, фиксирующего договоренность по взаимным обязательствам сторон. Как мы знаем, это понятие стало в современной разработке ПО буквально всепроникающим – как правило, в виде метафоры «клиент-сервер».

Посмотрим на диаграмму ниже. Интересующий нас здесь **этап анализа и проектирования** следует за этапами описания концептуальной модели предметной области – и предшествует реализации абстрактной модели этой области в конкретном языке программирования. Иными словами, этот вид деятельности имеет дело с высоким уровнем абстракции – способами выделения основных черт, общими для *всех* концептуальных моделей (*анализ*), и общими принципами создания на их основе абстрактных спецификаций к программной реализации ПО (*проектирование*).

С позиции принятого в UML подхода, спецификацию – уточнение задачи в рамках специального этапа предварительной подготовки последующего этапа (программирования), нужно понимать расширительно. Здесь предполагается не только передача некого артефакта - документа, точнее определяющего требования к ПО (условие, постановку задачи), но и планирование процесса ее дальнейшего решения. Имеется в виду частичное решение задачи в виде *архитектуры* создаваемой системы, и даже – предварительное планирование *организации процесса* работ по ее решению.

Детали подхода UML, переводящие многие привычные акценты программирования в иную – менее жесткую плоскость согласования видов работ с более размытыми границами, см. ниже. Но цель уже прозрачна – по мере возможности, нужно постараться облегчить всем участникам команды надежное прохождение всех последующих этапов. (Как увидим далее, в UML – на основе взаимности). Всем – по мере возможности, но в первую очередь – идущим следом программистам. Здесь мы четко видим новое назначение формальных схем высокого уровня абстракции (математики).

Особо отметим здесь необходимость *контроля* степени продвижения с *учетом* неизбежной последующей корректировки начальных планов (manage changes and assets) *всеми* участниками процесса разработки.

Manage, англ. – не только (и не столько) «управлять», но и «*справляться»*. В том числе - с проблемой, задачей, внезапно возникшей в ходе дела трудностью и т.п. В данном случае речь идет о продвижении (asset – ценное, приобретенное, достигнутое) и изменении (changes) относительно первоначальных планов. В общей концепции современного менеджмента как *управления проектами* – в том числе, проектами по разработке ПО - важно заметить то, что оно ставит своей целью контроль не над людьми, но возникающими в процессе их рабочего взаимодействия проблемами и направленными на решение этих проблем соглашениями (agreements, от agree – согласится, быть согласным). Центральный вопрос здесь – не кто кем управляет, но кто за что отвечает. За результат работы отвечают все – за конкретный вид работ отвечает специалист. Желательно – высококвалифицированный.

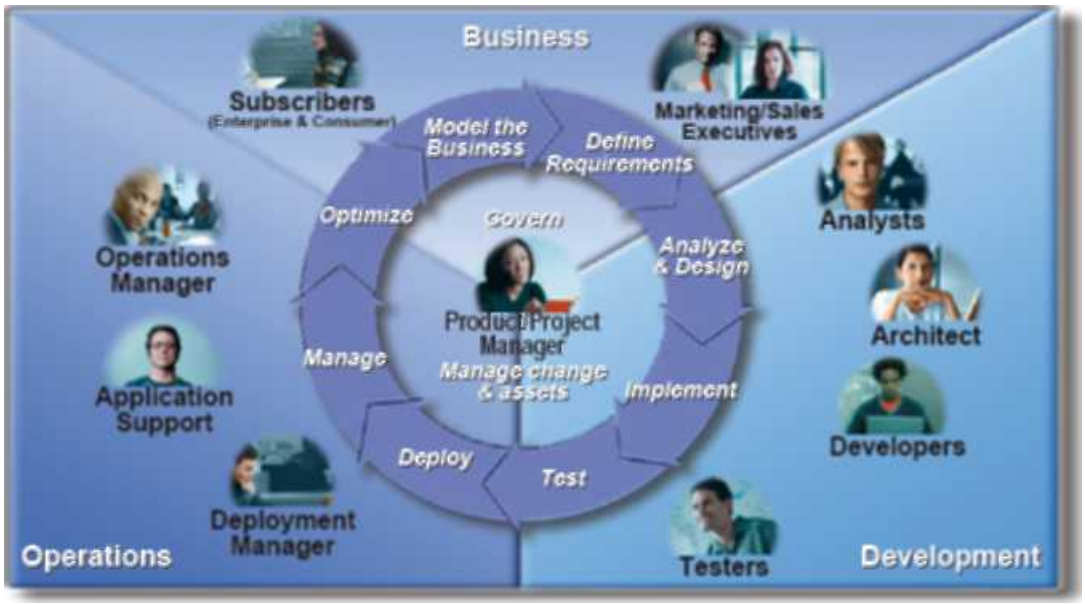
В силу имеющихся у него возможностей и общих ресурсов. Общая дисциплина управления проектами выделяет материальные (например, компьютер), финансовые (денежные) и временные

(сроки работ) и человеческие ресурсы (например, количество работающих). В нашем случае очевидно уместнее всего (до учета всех иных) начать с *интеллектуального* ресурса. Что не стоит путать с хитроумием или IQ – способностью быстро находить решения. Такой ресурс строго *ограничен* – у *любого нормального* человека (хотя не все это признают - но «суперменам» вряд ли стоит принимать участие в разработке больших систем). Как справиться с этой проблемой? Повышение квалификации как умения использовать абстрагирование для распределения сложности во времени (за счет времени) – пожалуй, наилучшая мотивация для изучения новых подходов.

Важно отметить при этом, что роль (специализация) отделена здесь от конкретного человека. На практике человек может одновременно исполнять разные роли и даже, при общем согласии - изменять свою роль в ходе выполнения работ. Иначе говоря – при необходимости - для достижения лучших результатов коллектив может динамически изменять распределение ролей. Имея в виду не только конечный результат, но и сам ход работ.

Менеджер проекта отвечает главным образом за взаимоотношения людей - а потому эта роль может выполняться человеком, не глубоко понимающим многочисленные скрытые от него конкретные детали каждого этапа - постольку и поскольку это не сказывается на общей атмосфере в коллективе. Но конечно его вклад в создание и поддержание нормальной температуры (36.6°) этой атмосферы – самый важный. Со своей стороны специалист – не отвечая формально «по должности» за задачи общего менеджмента, должен постараться по всей возможности не создавать другим разнообразных проблем не профессионального характера. Как известно, горячка в работе приводит к охлаждению отношений.

Цель же всей команды, коллектива разработчиков нацелена на создание программной системы – некоторого специального продукта, облегчающего жизнь клиенту (пользователю системы). Но при этом *не* создающего для всех других людей проблем не специального свойства – что имеет отношение уже не к этапам разработки, но более всего – к *содержанию* (content) разрабатываемых нами систем. Есть масштабы, крупнее масштаба предприятия. Эту ограниченность современного применения «клиент-серверных» метафор тоже стоит отметить особо.



Оставшиеся, более «приземленные» этапы *жизненного цикла* (lifecycle) разработки ПО не рассматриваются глубоко в данном пособии. Конечно же – не потому, что они не важны. Напротив. Работа на каждом этапе должны по мере возможности облегчать прохождение последующих. Но при этом мы можем исходить лишь из принципиального понимания их сути – заведомо не зная во всей полноте возникающих внутри них сложностей.

Здесь отметим своеобразное преломление центральной для кибернетики идеи «черного ящика» и созвучных ей идей инкапсуляции (сокрытия, hiding) в программировании. Находясь «вне», мы не знаем всех сложностей «внутри» – точно зная лишь то, что на деле «внутри» все сложно.

Короче говоря – всем работающим трудно, всем сложно (не только тебе и мне). Именно это и заставляет нас быть проще в отношениях с людьми.

Этап развертывания (deploy) имеет дело с проблемами внедрения ПО в конкретной физической среде – с учетом особенностей устройства (архитектуры) конкретных компьютеров, специфики построения компьютерной сети и т.п.

Как мы хорошо знаем, на этапе программирования такой учет специфики «железа» (т.е. знание логической схемы построения физических устройств) может сделать систему более эффективной, в чисто вычислительном значении термина. И, одновременно, менее надежной. В целом – менее масштабируемой и гибкой, стабильной, устойчивой при непредвиденных ранее изменениях. В еще большей степени это относиться к этапу анализа и проектирования.

Этап сопровождения (manage) решает организационные проблемы внедрения ПО в конкретной социальной среде, организации.

Здесь облегчает дело то, что типов организации не так уж много. Усложняет - то, что «типов» людей – ровно столько, сколько самих людей. Проще говоря, все люди абсолютно индивидуальны. Собственно, лучше решать проблему согласованной работы самых разных людей и помогает нам UML. Хотя, разумеется, не нужно быть высококвалифицированным менеджером, чтобы понимать, что никакая технология не решит за нас наши проблемы - автоматически.

Ключевым в понятии жизненного цикла ПО (lifecycle) понятие *версии* – как результата работы, подлежащего последующей **оптимизации (optimization).** Здесь имеется в виду проблема дальнейшего улучшения качества ПО – уже не только в более привычном специальном значении (вычислительная оптимизация), но и в самом широком понимании термина «качественное, хорошее».

Из-за чего результат работы может быть не идеальным, не совершенным и не полным – а потому не может считаться законченным? Мы помним, как в программировании термин *ошибка* (error) постепенно заменился более общим термином *исключение* (exception). Зачастую, но далеко не всегда недостатки результата работы – это ошибки разработчиков. После окончания работ выясняется, что те или иные, большие или мелкие недостатки были допущены на *всех* предыдущих этапах – начиная с этапа неформальной постановки задачи ее заказчиком.

Иначе говоря, недостатки продукта часто порождаются принципиальной невозможностью сразу сделать все идеально - неспособностью *всех* вовлеченных в проект людей предусмотреть все возможные проблемы и запланировать подходящие для них решения заранее. Признание этого принципиального факта существенно отличает подходы к разработке «в малом» и «в большом» *любого* ПО. И большого ПО, и малого – дело тут не в объеме, но потенциальных опасностях (рисках).

Что нам видеть в понятии жизненного цикла?

Каждый этап разработки связан с профессиональным видением - представлением (view) каждого участника разработки о специальных, внутренних для данного этапа проблемах и решениях. Однако, командная разработка невозможна без общего виденья (vision) общих целей своей деятельности и самых принципиальных решений возникающих в ней всеми участниками проекта.

Только принципиальных – но далеко не всех возможных.

Иными словами, view связано с нашим представлением о том, как наилучшим образом «сыграть свою роль». Vision же - с тем, зачем вообще людям нужно распределение ролей и какие именно роли нужны в данном деле - крайне серьезном и предельно далеком от всяческих игр.

Посмотрим, какое новое виденье понятия «жизненный цикл ПО» предлагает *стандартный* уже ныне подход к разработке больших программных систем, характерный для UML. По сути, он предлагает нам по-новому взглянуть на это понятие – и традиционную организацию работ.

Как известно, термин *стандарт* имеет много значений. *Общепринятый* – в данном случае, во многом устоявшийся в среде серьезных разработчиков ПО (что не исключает самых разнообразных вариаций на основную тему). Но здесь нам стоит воспринимать его чуть иначе: *хуже – уже нельзя* (лучше можно).

Глава 3. Как понимать UML? Каскадный и инкрементный подход к разработке ПО.

Математика - это язык. Исаак Ньютон.[4]

Слова, слова… UML – взятый «сам по себе», вне контекста его происхождения и назначения - будет для нас лишь еще одним набором обозначений, если не видеть сутью нового для нас языка некий качественно новый методологический подход, за которым стоит огромный опыт разработчиков.

Больше того. Все мы начинаем изучение программирования с «картинок» - блок-схем, автоматных диаграмм, графов и т.п. схем. Очевидно (в самом буквальном значении слова), что возврат создателями UML к диаграммам, визуальному представлению знаний - в определенном смысле, есть *шаг назад* по сравнению с куда более изощренными языками вроде современных языков и сред программирования.

Попытаемся понять, где же здесь скрыт *шаг вперед.* Это важно и потому, что сегодня и сам UML становиться все более изощренным языком.

Новый язык – это новая попытка выразить компактно, минимальными средствами некоторую методологию работы, рекомендуемую для лучшего ее выполнения совокупность «работающих» на практике принципов. В данном случае, главный акцент делается на согласованном понимании будущей программной системы всеми участниками разработки.

Иными словами, по факту своего происхождения UML связывается с «моделированием процесса моделирования (процесса разработки моделей)» - предварительным описанием моделей программных систем, направленном на снижение рисков неудачной разработки. В особенности тесна генетическая связь UML с *итеративным* подходом к разработке таких систем.

Далее, по мере детализации (конкретизации, приближения к реализации на практике) подход (идея, «философия», «идеология») перерастает в методологию. Далее - в технологию. Классический пример связанной с итеративным подходом технологии - Rational Unified Process (RUP).

Однако, UML – весьма абстрактный язык с крайне широкой областью применения. Строго говоря, формальный язык не предопределяет однозначно все возможные варианты ни исходной методологии, ни сферы своего применения. С другой стороны, он бесполезен в применении без понимания базовых принципов, которое и служит для нас предварительным условием его освоения. Именно поэтому здесь мы обсуждаем именно и только подход.

Инкрементный (т.е. пошаговый) или итеративный (циклический) подход, как и более ранний *каскадный*, исходит из уже затронутого нами понятия *жизненного цикла ПО* – разделения процесса создания ПО на этапы, ответственность за успешное выполнение которых несут разные специалисты.

Выше мы постарались очертить происхождение разных специализаций как разделения труда во времени и между людьми - по мере возрастания сложности разработки. Не стоит «стрелять из пушки по воробьям» (вполне очевидное исключение здесь составляют цели обучения). Не трогай того, что работает – пока оно работает достаточно хорошо …

Итеративный подход *не заменяет* каскадный в более простых случаях. Точно также, например, объектно-ориентированный подход не заменяет полностью процедурный,

---

[4] Говорят, что Ньютон произнес эту знаменитую фразу в связи с обсуждением распределения академических часов на занятия иностранными языками и математикой.

функциональный и т.д. Новый подход нов потому, что выражает иное, не стандартное применение традиционных понятий. Т.е. пока еще непривычное для нас - но уже опробованное на практике другими.

Иными словами, между старыми и новыми подходами к созданию ПО нет прямого противоречия, но присутствует косвенное определение сложных систем. Сложны такие системы, для которых прежний подход перестает быть осуществимым.

Каскадный подход предполагает строго последовательное выполнение этапов:

1. Во времени: следующий этап должен начаться не ранее предыдущего, и
2. Связь по результату: участник предыдущего этапа должен передать законченный итог своей работы участникам последующего.

Итеративный подход предполагает (как всегда, *в идеале*) частичное продвижение во времени к общей цели через своеобразный параллелизм выполнения этапов через близкое (постоянное, непрерывное, частое и тесное) взаимодействие *всех* участников, их соучастие в выполнении каждого этапа в той мере, в который на данный момент он может быть успешно выполнен.

Теперь мы не продвигаемся к конечной цели согласно жизненному циклу – мы стараемся (по мере возможности) продвинуть каждый этап жизненного цикла. Вместе (а не один за счет другого). Т.е. сам цикл работ – со всеми ее этапами.

Параллелизм здесь не означает полной независимости. Совсем наоборот. Для лучшего понимания нового для нас подхода, вспомним столь знаменитое в раннем программировании «разделяй и властвуй». Есть кардинальная разница между стремлением продвинуть работу и продвинуться самому. Разделяй – сложность работы, властвуй – над собой. В основе нашей работы лежат ранние, уже существующие концептуальные модели той или иной формы организации работ, но и они постоянно изменяются. Прощай, Цезарь…

Структурный подход как разделение сложности изначально воспринимался достаточно прямолинейно - как разбиение системы на части (компоненты, модули), каждая из которых может быть успешно реализована одним или разными программистами независимо, изолированно от остальных. Почти независимо – т.е. в предположении, что статические (структуры данных) и динамические (передача управления) связи между частями могут быть явно осознаны и описаны заранее.

Этот специальный (как оказалось впоследствии) метод хорошо отражает общее понимание тех простых процессов, когда некий *исполнитель* принимает результаты (аргументы функции) обрабатывает что-то (вычисление функции) и по завершению работы передает результат другому (или же, как вариант - возвращает их некоторому главному исполнителю). Ничего иного, кроме выполнения своей функции по реализации отдельной компоненты, этого *исполнителя* не заботит. Это и есть первоначальная идея специализации. Современные подходы в программировании – включая итеративный – никак не отменяют специализацию (компетентность в данном виде работ), но ищут способ минимизировать вызываемые ей риски.

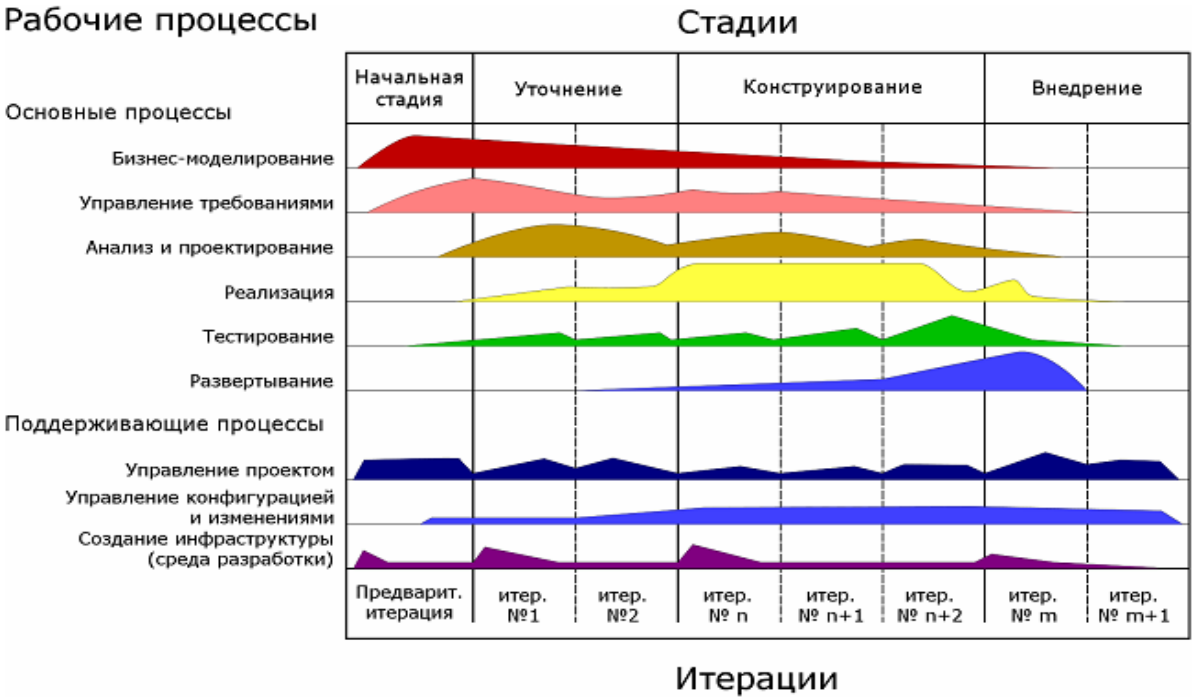
Пожалуй, наиболее наглядным примером подобных рисков служат ошибки спецификации. Программист может прекрасно – правильно, эффективно и т.п. - решить *не ту* задачу, которую подразумевал ее постановщик. Эта серьезная проблема в свое время вызывала большие надежды на теорию формальных спецификаций программ. Крайне полезный для (математически грамотного) программиста, такой подход не учитывал то, что постановщики задач – тоже люди, которые не могут заранее определиться в требованиях к ПО. Во всяком случае, это тяжело сделать до тех пор, пока нам не видны основные возможности их реализации.

Словом, здесь мы рискуем попасть в тупиковую ситуацию, известную в программировании как deadlock («заклинивание»). Следуя заданной программе, каждый ничего не может сделать, потому что ждет для начала своей работы окончания работы другого. И так – для каждого вида работ…

По факту же это означает, что большой жизненный цикл создания всего продукта разбивается на ряд небольших итераций – согласованных с реальными возможностями участников разработки, потому более мелких жизненных циклов частичного создания будущей системы. Причем не по отдельным ее компонентам - но так, чтобы итогом каждой итерации (по возможности) получался не полнофункциональный, но все же достаточно полноценный, доступный для оценки его качества продукт.

См. на диаграмме примерный ход разработки при итеративном подходе.

При каскадном подходе получилась бы лестница с четко ограниченным начало и концом времени прохождения каждого этапа – с передачей результатов на границе этапов. При итеративном подходе ступеньки «расплываются» - сохраняясь лишь в виде тенденции, желания распределить общую нагрузку на использование тех или иных, неизбежно ограниченных ресурсов. Особенно – внутренних ресурсов самих участников.



Практические рекомендации для определения длительности и количества итераций в зависимости от предполагаемой длительности проекта и выделенных на него ресурсов можно найти в [Larman].

Понятно, почему такой подход называют то инкрементным, то итеративным. По сути своей он скорее ***спиральный.*** Мелкими оборотами (итерациями) мы понемногу (инкрементно, т.е по шагам) продвигаемся вверх, ближе к желаемому результату. Но - *не гладко*.

Здесь мы явно ощущаем продолжение предыдущих идей версионности и прототипирования – раннего создания предварительной версии (прототипа) продукта. *Прототип* – не полноценная, частичная реализация, но уже вполне достаточная для обсуждения основных свойств будущей системы с ее пользователем и заказчиком. Понятно, что этот метод исходит из необходимости решения затронутой выше проблемы невозможности заранее специфицировать все свойства системы ее заказчиком.

Однако, если предыдущие подходы однозначно ориентировались на удовлетворение требований заказчика, то современные – на возможность выполнения своих ролей всеми, прямо или

косвенно вовлеченными в процесс разработки. Разумеется, включая заказчиков – инициаторов построения системы как главных заинтересованных лиц (stakeholder – буквально, держащий шест, т.е. опора), выделяющих необходимые для работы ресурсы.

Но на сегодня считается, что наивысший приоритет при разработке имеют интерфейсные функции (услуги, сервис), предоставляемые будущей системой ее будущим конечным пользователям, клиентам. Не случайно понятие use case - прецедента использования – считается в итеративном подходе первоочередным. Возможно – даже *единственным*, поскольку использование остальных типов диаграмм не строго обязательно. Точнее – иные диаграммы нужны для уточнения прецедента, когда и если в том есть необходимость. Она же в случае достаточно крупных программных систем – конечно же, возникает.

Глава 3. Как применять UML? Обзор языка «с высоты птичьего полета».

Для того, чтобы иметь возможность сделать первые выводы о заложенных в UML возможностях, рассмотрим несколько примеров использования диаграмм - в качестве иллюстраций базовых понятий, относящихся большей своей частью к подготовке лучше нам знакомого этапа программной реализации.

При этом мы будем уделять больше внимания на моменты, хуже нам знакомые – особенно, на понятии прецедента. Впрочем, как известно, «новое – это хорошо забытое старое». Точнее, с точки зрения автора – это вспомненное и *по-иному*, лучше понятое старое. Дело не в том, что мы не знакомы с чем-то вообще – это исключение. Как правило – мы встречали все это ранее. В более простых ситуациях – и именно поэтому мы не придаем этому особого значения. До тех, пока… *гром не грянет, мужик не перекреститься*.

Проблема масштабирования обычно трактуется сегодня как *возрастание* некоторого количества. Например, если пользователь один и стоит рядом (а еще лучше – если это ты сам) – все просто. А если - миллион? Но более существенная проблема состоит в *убывании*. Обычно мы предполагаем некий автоматизм (само собой!) сохранения того простого (в целом - хорошего), что имеем. *Что имеем – не храним, потерявши плачем.*

В разработке ПО инварианты (неизменное, постоянное, константы) не сохраняются - их *сохраняют.* А потому разумный человек, естественно, старается все-таки *по мере своей возможности* предупредить нежелательное развитие событий. Немного, но заблаговременно (а потому «не креститься» все время - иначе *весь лоб расшибет*).

Но при этом нужно сразу отметить некоторые оригинальные черты применения UML – полностью соответствующие изложенными выше принципами и назначению языка, но зачастую далекие от наших первоначальных ожиданий. Ни одна из используемых в процессе разработки диаграмм *не* дает и *не* призвана давать полного и завершенного формального описания (спецификации) разрабатываемой системы. Поэтому:

- UML – не чисто визуальный язык. Диаграммы формулируют только основное текущее согласие участников в понимании системы, достигнутом на какой-то стадии разработки. Детали соглашения (в том числе, спорные) фиксируются неформально в комментариях, являющихся неотъемлемой частью каждой диаграммы.
- Разные типы диаграмм дают разный взгляд (проекции) на те или иные аспекты - стороны функционирования или строения (архитектуры) системы. Диаграммы могут содержать одну и ту же информацию (проекции не ортогональны) и
- Одна диаграмма может служить уточнением другой, т.е. служить *частичным* описанием некоторого аспекта поведения или строения системы. Именно описывающий один аспект группу взаимосвязанных по тематике и/или времени диаграмм называют в UML view, т.е. представление или взгляд на разработку с точки зрения того или иного участника разработки. Точнее, ролью участника на границе смежных этапов «жизненного цикла разработки ПО». В русском переводе, их обычно также называют моделью [относящейся к заданной теме]. Скажем, модель приложения связана с границей «проектирование и программная реализация», модель предметной области – с границей «анализа и бизнес-моделирование». Во избежание путаницы, далее в этом случае мы предпочитаем на равной основе говорить просто о *задаче описания*. Всегда подразумевается - описания *частичного,* незавершенного.

Единственным законченным описанием служит лишь результат разработки в виде завершенной, готовой к использованию *версии* программной системы.

Далее, в согласии с подходом UML, мы связываем типы диаграммы с их основным назначением, связанным с их происхождением и традиционным применением на некотором этапе разработки. Не всегда, но изредка – когда считаем нужным подсказать будущему разработчику, еще не вполне определившемуся со своей будущей ролью (конкретной специализацией) связанные с ней ограничения. Но на будущее не стоит воспринимать такую связь слишком буквально. Она не фиксирована и *любой* тип диаграмм можно использовать для решения задач *любого* этапа. Не произвольно – но при необходимости продвинуться в решении задачи этого этапа.

Например, едва затрагиваемые нами здесь диаграммы развертывания традиционно связываются не с программной реализацией, а с внедрением программных систем на реальных физических устройствах. В данном контексте они лучше понимаются – но с равным успехом могут быть использованы и на этапе программной реализации. Например для машин виртуальных.

- Когда именно какие типы диаграмм в каком порядке нужно использовать? Универсального «алгоритма разработки ПО» – нет. А потому это крайне сложный вопрос, сильно зависящий от конкретной задачи (см. пример разработки далее) – и квалификации конкретного разработчика. В каком порядке *пробовать* ту или иную форму того или иного типа диаграммы? Ответ (на словах) более простой – нужно знать эволюцию (историю) своего дела. Она шла от простого к сложному. Первый шажок в понимании этой эволюции (с опорой на уже имеющиеся у читателя знания и умения[5]) мы и попытаемся сейчас сделать.

Иначе говоря, дальнейшее стоит воспринимать как примеры использования тех или иных типов диаграмм UML для решения той или иной задачи описания. Общее назначение того или иного типа диаграмм, как правило, вполне прозрачно из его названия.

Задача описания прецедентов.

Модель прецедентов строится на этапе анализа основных требований к программной реализации, отражая согласованное видение (согласие между всеми заинтересованными лицами относительно) использования будущей системы.

Актор[6] (act – действовать, actor – исполнитель [роли], действующее лицо) - некая активная, т.е. взаимодействующая с системой часть ее окружения, идентифицируемая исключительно по характеру такого взаимодействия. Например - идентифицируемый своей ролью человек (например, конечный пользователь), организация, иная компьютерная система, внешнее устройство, датчик, источник и/или приемник сигналов и сообщений и т.п.

---

[5] Согласно учебной программе :(- но смотри послесловие ☺

[6] Русскоязычная терминология, относящая к рассматриваемым проблемам, еще не вполне устоялась. В случае затруднений с переводом автор – во избежание неоднозначного понимания - использует «кальку». Неточный перевод чужих реалий в свои собственные порождает крайне тяжелые проблемы.

Сценарий (scenario) — специальная последовательность действий или взаимодействий между актором и системой. Его также называют *экземпляром прецедента* (use case instance). Может быть успешным либо неудачным в контексте содержательных задач использования данной системы.

Прецедент (**use case** – буквально, [отдельный] случай использования) — описания варианта использования системы в виде набора взаимосвязанных успешных и неудачных сценариев, описывающий возможную цель использования системы в контексте решения поставленных перед ней задач. По своему назначению, прецедент представляет собой - в текстовом или ином виде - рассказ об использовании системы в качестве инструмента, помогающим ее пользователям в решении их задач.

Мы будем рассматривать понятие прецедента с точки зрения (относительно, сравнительно) уже знакомого нам алгоритмического подхода.

Описание прецедентов. Диаграммы прецедентов - use case diagrams

Диаграммы прецедентов *частично* описывает use case – прецедент использования проектируемой системы, давая *частичное* описание *частичного* применения системы с точки зрения условного внешнего обозревателя (за которым – в идеале - стоит согласованная точка зрения участников работ). При этом описание фокусируется на том, ***что*** должна делать система по отношению к своему внешнему окружению (периферии), а не то на том, ***как*** она эта делает.

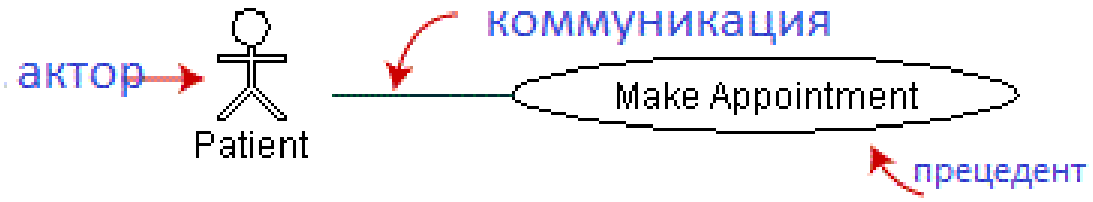
Иначе говоря, диаграмма есть *частичная* спецификация.

Напомним, в рассматриваемом подходе выделение основных (но не всех возможных) прецедентов на начальной стадии разработки играет ключевую роль, что отличает этот подход от чисто алгоритмического. При этом сначала обычно рассматриваются успешные сценарии (варианты).

Заметим, в случае достаточно крупных программных систем мы не используем термин «алгоритм», например – неуместно говорить об «алгоритме текстового редактора (редактирования текста)» и т.п. Причина ясна – в сложных случаях мы можем надежно описать (проанализировать, проверить и т.п.) некоторые *ветки* алгоритма верхнего уровня, но не сам законченный алгоритм, предполагающий законченное описание всех возможных веток.

Иначе говоря, сценарий – это и *частичная* алгоритмизация.

Пример. Запись пациента на прием в поликлинику.

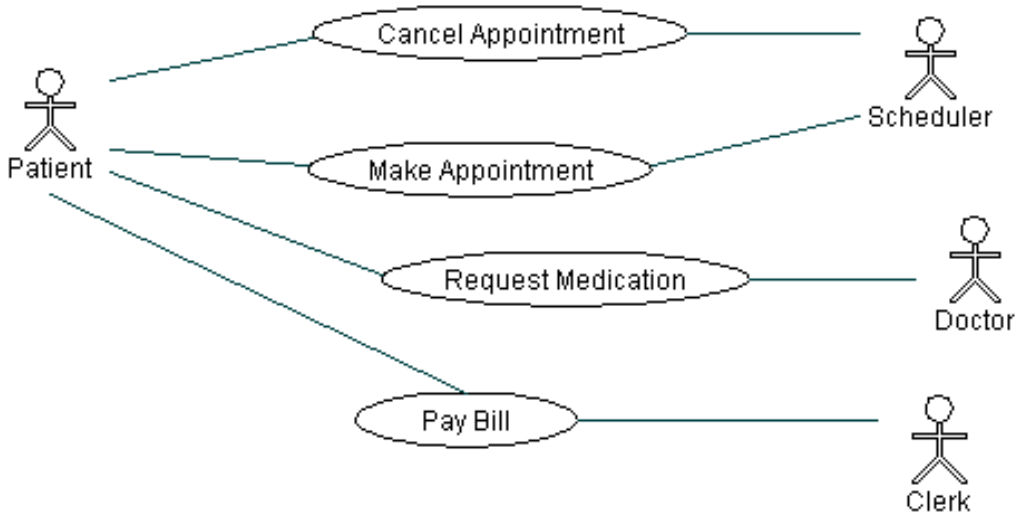


Данный простейший пример предполагает примерно следующий скрытый за ним сценарий прецедента (естественно сделать его комментарием к имени прецедента):

"Пациент звонит в поликлинику для записи к врачу в целях профилактики. Регистратор поликлиники находит в журнале регистрации ближайший свободный период, согласует время с пациентом и фиксирует его в журнале»

Make Appointment (согласовать назначение [у врача]) – имя прецедента. Patient (пациент) – роль человека. Связь между актором и прецедентом – ассоциация коммуникации (для краткости коммуникация – в виде обмена сообщениями)

Пример 2. Уточнение предыдущей диаграммы (посредством ее расширения).



Вторая диаграмма отражает больше деталей ситуации. Здесь появляются новые акторы: Scheduler – регистратор, Doctor – врач, Clerk – служащий, и новые прецеденты CancelAppointment – отменить прием, RequestMedication – запросить лечение и PayBill – оплатить счет.

Крупные программные среды UML-моделирования имеют средства поддержки версий. В таком случае мы можем считать представлением (view) группу диаграмм, относящихся либо к последней версии, либо ко всем. Второе, очевидно, предпочтительно.

Уже самый первый пример ставит перед нами «главный вопрос всех времен и народов»: «Ну и зачем мне эти диаграммы прецедентов?» Еще более популярный вариант: «Да вообще кому они нужны - все эти диаграммы!». В процессе обучения его лучше не скрывать в себе, но переадресовать себе самому, перефразируя и конкретизируя. Кому *именно* и когда *именно* и зачем *именно* они полезны?

Программист, конечно, увидит для себя возможности программной реализации - очертания [архитектуры, строения] будущей программы. Для понимания остальных возможностей нужно лишь представить себя в другой роли. В роли клиента – для оценки полезности использования системы в своей работе. В роли заказчика и постановщика – при обсуждении задачи. В роли тестера – при подготовке тестов прогона тех или иных веток выполнения программной системы.

Мораль стара – полезно встать на место другого, посмотреть на мир его глазами.

Но обратимся снова к конкретному примеру. Зачем нужно такое уточнение? Для человека, хорошо знакомого с предметной областью (доменом) – в данном случае, системой здравоохранения – в этой диаграмме нет новой информации. Для него она неявно содержится уже в первом прецеденте, выводиться из него. Однако, во-первых - не все люди одинаково хорошо знакомы с данным доменом.

Во-вторых – они могут иметь разные мнения как на сам домен, так и на то, какие его черты должны быть отражены в разрабатываемой системе. Наконец, мы можем в дальнейшем попросту забыть то, на чем договорились.

Проще говоря, для понимания необходимости диаграмм *вообще*, достаточно представить себя просто человеком, который не все знает, не все помнит, не все умеет и т.д. Короче – хорошо бы быть скромней…

Описание коммуникаций. Диаграммы последовательностей [системы]-[system] sequence diagrams

Диаграммы последовательностей некоторой системы объектов - артефакт модели (часть документированного описания) прецедентов, используемый для частичного описания *поведения* некоторой системы объектов в виде возможных, основных (успешных) и альтернативных сценариев.

Главное назначение таких диаграмм — отображение событий, передаваемых исполнителями системе через ее границы. Каждая из них дает схематическое описание *сценария прецедента* в виде последовательности событий, генерируемых внешними акторами - и *компактное, обозримое* (для анализа, контроля и т.п.) описание событий, генерируемые внутри самой системы. Иначе говоря – диаграмма пытается ответить скорее на вопрос: «какие главные события инициируются извне», чем как именно система реагирует на внешние сигналы. Не точно и полно - лишь по мере возможности.

Отметим здесь новую трактовку старого поведенческого подхода (бихевиоризм, от behavior - поведение), базового для кибернетики. Строение системы важно – но *не* само по себе, а лишь постольку, поскольку оно обеспечивает нужное поведение, верную реакцию на внешние сигналы. Не так важно, как устроена система – важнее, чтобы она имитировала нужное поведение. Устройство разнообразных т.н. «умных» машин «виртуально» - это касается машин как реальных, так и виртуальных. Проще говоря – оно не должно и не может походить на настоящее. А поведение – реально.

В кибернетических терминах, все внутренние подсистемы (связанные с описанием объектов данной системы) рассматриваются как "черный ящик", а сама система – как полупрозрачный, или «серый ящик». Как и ранее, то, что находиться вне ящика считается относящимся к *логике*, внутри ящика – к *реализации системы* (скрытой в устройстве ящика – касается ли она определения структуры связанных с ним данных или методов). В «полупрозрачном» случае, имеется в виду реализация самого верхнего уровня.

Вспомним, что классы и объекты в ООП описывают – соответственно, на более и менее абстрактном уровне – статическое представление модели. Они содержат указание на методы – именованную ссылку на определение правил (алгоритмов) *потенциального* поведения объектов, при этом скрывая фактическое поведение объектов, по отношению друг к другу. Проще говоря, они отвечают скорее на вопрос «как может вести себя [один] данный объект», а не на вопрос о том, как ведут себя объекты – вместе, в целом.

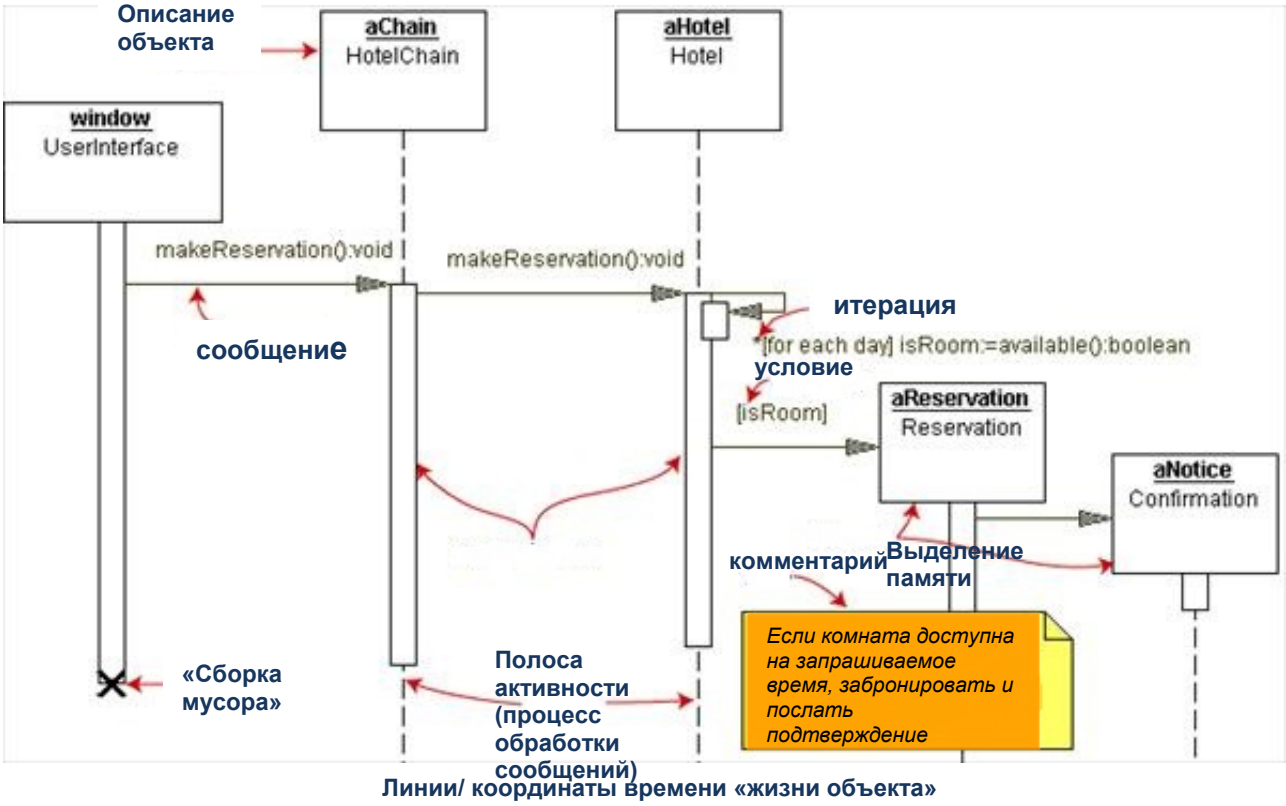
Поведение системы объектов рассматривается здесь как скоординированное взаимодействие объектов, фиксируемое в виде временно́й шкалы исполнения методов посылки сообщений - трассы сообщений. Течение времени отображается координатой «сверху вниз», посылающие сообщения объекты (источники) левее, получающие сообщения – правее (приемники сообщений). По возможности.

По процедурному программированию нам хорошо знакомо понятие *трассы [состояний]*, описывающей фактическое поведение объекта в виде последовательности смен его состояний. При этом каждое из состояний описывается в виде именованного набора значений переменных – характеристик состояния объекта на некий момент времени. В императивной (командной) модели управления подразумевается, что *некто* (исполнитель верхнего уровня) командует изменениями ниже стоящего объекта, вызывая связанные с ним процедуры в соответствии с некоторым правилом - алгоритмом, программой изменений подчиненного объекта. *Трасса вызовов* – и есть ветка исполнения алгоритма, представляемая в виде последовательности имен соответствующих процедур.

В данном случае речь снова идет о трассе вызовов – точнее, о трассе инициирующих такие вызовы сообщений. Но уже в более сложном случае системы объектов, изменяющих (через вызов метода объекта-источника) состояний объекта-приемника (а также, возможно, в качестве побочного эффекта – и состояний других объектов). Изменение состояния и трактуется теперь как «событие в жизни объекта».

Здесь по-прежнему есть внешние объекты, инициирующие внутренние процессы обмена сообщениями, но в более сложных для описания случаях нет единого главного «командира» и иерархии подчиненности.

Пример. Бронирование места в гостинице.



Внешний объект, инициирующий поток сообщений – окно регистрации Reservation window. Объект Reservation window посылает сообщение makeReservation() {забронировать } соответствующей службе сети отелей HotelChain. Та, в свою очередь пересылает сообщение makeReservation() отелю Hotel. Если в гостинице есть свободные номера, объекта Hotel вызывает методы Reservation {забронировать номер) и Confirmation {подтвердить заказ}.

Каждая из пунктирных вертикальных «линий жизни» (lifeline), представляет заполненное событиями время потенциального существования некоторого объекта – в виде изменения состояний других объектов и изменения его собственных состояний. Каждая стрелка описывает вызов метода посылки сообщения. Стрелка идет от отправителя к приемнику сообщения, выделяя полосу активности (период обработки сообщения - activation bar) на временной координате.

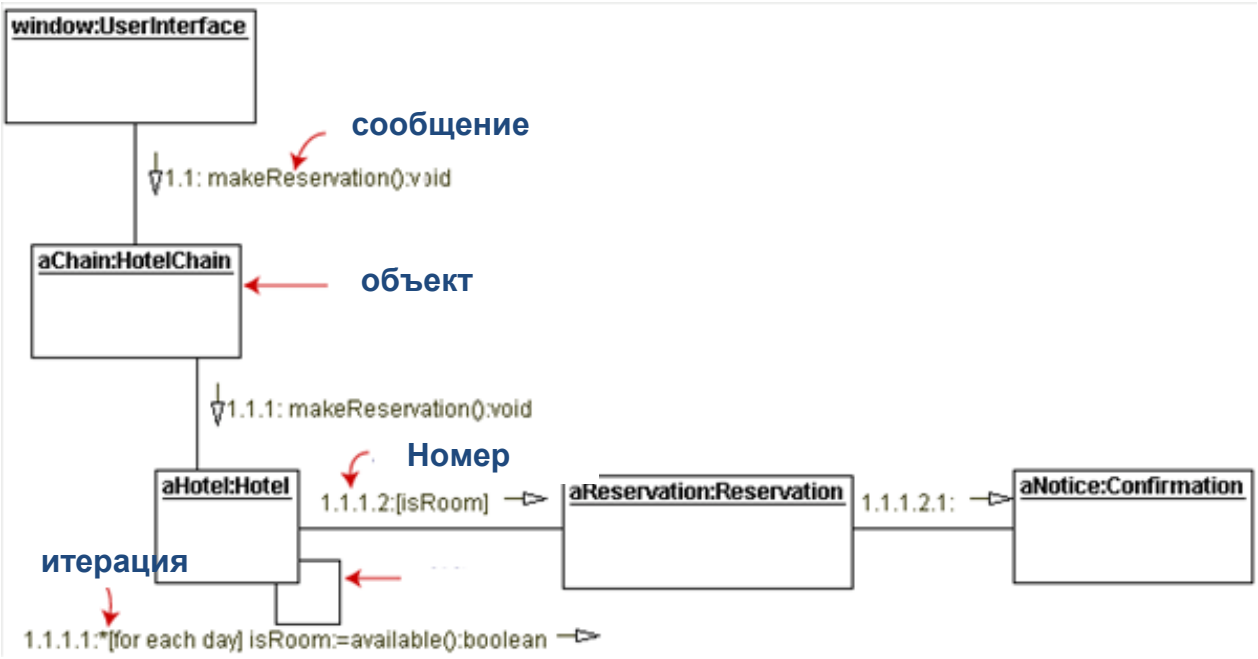
На диаграмме объект Hotel вызывает самого себя (self call) для определения доступности номера. Если условие IsRoom выполняется, Hotel создает объекты Reservation и Confirmation. Звездочка на вызове available означает итерацию – циклический вызов (для того, чтобы выяснить, доступен ли номер для всех дней предполагаемого пребывания гостя). Выражение в квадратных скобках означает условие (предикат).

Диаграмма включает в себя также поясняющий комментарий в виде текста на прямоугольнике с загнутым углом - возможный и, в любом неочевидном случае необходимый для любой иной UML диаграммы.

Диаграммы кооперации (взаимодействий) – collaboration[7] diagrams

Диаграммы кооперации, как и диаграммы последовательностей, относятся к диаграммам описания взаимодействия. Они содержат ту же информацию, что и диаграммы последовательностей – но в ином ракурсе, фокусируясь не на описании процессов коммуникации между объектами (в виде последовательности вызовов), но на определении роли объектов в качестве источника и приемника сообщений. Словом, здесь мы пытаемся ответить не на вопрос «как» происходит коммуникация фактически, но *кто* и кому может пересылать сообщения.

Пример. Тот же - бронирование места в гостинице.



[7] Collaboration – сотрудничество. В русском языке слово *коллаборационист* имеет негативный смысл, потому чаще говорят о диаграммах коопераций.

Прямоугольники, представляющие роли объектов, помечаются именем класса и/или объекта (в последнем случае имя класса отделяется от имени объекта двоеточием). Сообщения на диаграмме кооперации нумеруются. При этом сообщения нижнего уровня - посылаемые во время обработки некоторого сообщения - нумеруются префиксами, отделяемыми точками от номера сообщения высшего уровня – в соответствии с последовательностью этих вызовов.

Иначе говоря, таким хорошо известным нам образом (линейная запись дерева) кодируется факт следования или вложенности скрытых здесь процессов обработки сообщений – одного в другой. Как и ранее, они не могут перекрываться.

Описание строения. Диаграммы состояний. Statechart diagrams.

Диаграммы состояний, или автоматные диаграммы - пожалуй, наиболее хорошо нам знакомые. Или, вернее, *должны быть* нам знакомы - по меньшей мере, по вводному курсу дискретной математики.

Вспомним общие положения. Конечный автомат – модель возможного реагирования (в виде прямой и обратной реакции) объекта моделирования на внешние события - изменения во внешней среде (окружении, периферии, ситуации, контексте и т.п.)

Понятие автомата - центральное в понимании логической схемы функционирования компьютера как устройства, выполняющего пошагово *одну-единственную* операцию (как и любое иное), имитирующую операцию аппликации (применения функции к аргументу). Неявно, понятие автомата лежит в начале основных программистских концепций – например, трактовке программ как преобразователя потоков и/или последовательных файлов. Но в данном случае, нам стоит вспомнить концепции более ранние. Не о компьютерах или программах, но о не менее знаменитой собаке академика Павлова…

В общем случае, автомат реагирует на такие изменения среды

- явно определяемым изменением состояния своего внутреннего, невидимого извне и скрытого внутри него устройства; такие внутренние состояния в теории автоматов именуются, но не описываются.
- изменением состояния среды – неявным, поскольку наличие таких состояний среды подразумевается, но не сами состояния не определяются.

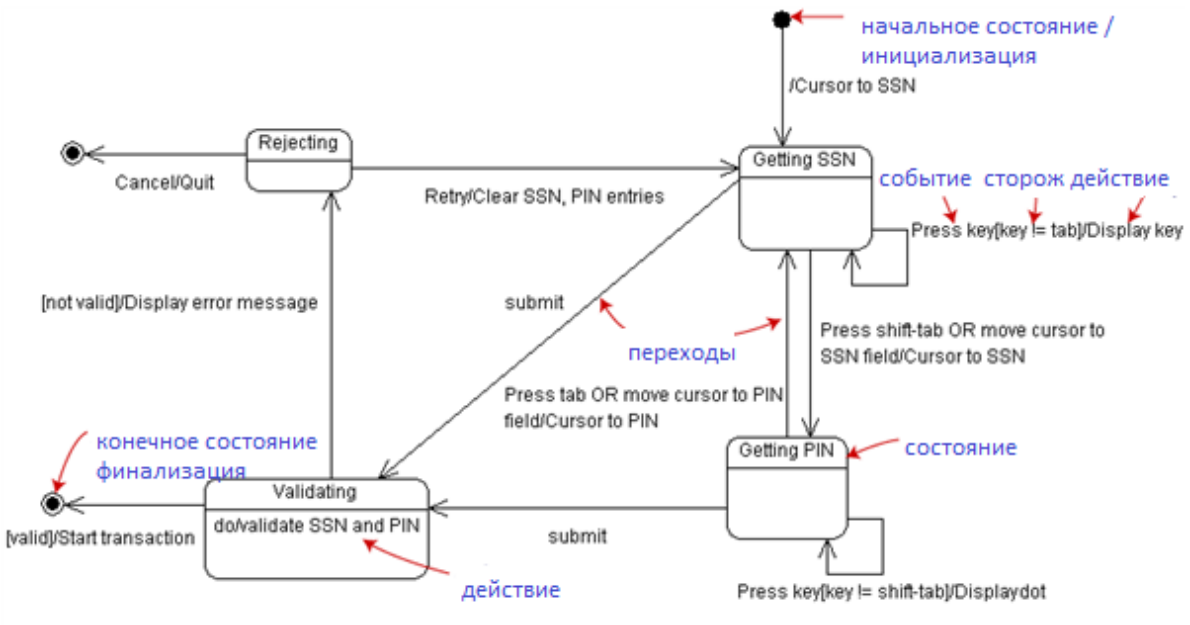
Обратной связью называется изменение поведения (внутреннего состояния) автомата, возникающее (косвенно) в результате его собственных действий по изменению среды.

В явном виде, определяется *коммуникация* автомата со средой, в виде

- множества возможных входных и выходных сообщений. Их роль в теории автоматов выполняют элементарные сигналы, буквы (в ООП, естественно – объекты).
- двух функций (или одного оператора), определяющих по текущим входному сообщению и внутреннему состоянию выходное сообщение и следующее внутреннее состояние. Соответственно называемых функцией выхода и функцией перехода (transition).

  Если (и *когда*) входные сообщения отсутствуют, автомат описывает *источник* (или отправитель, sender) подготовленных им ранее выходных сообщений. Если (и *когда*) отсутствуют выходные сообщения, автомат описывает приемник (адресат, получатель) и обработчик выходных сообщений.

Пример. Авторизация он-лайн пользователя банковской системы.



Авторизация предполагает примерно следующий сценарий:

- ввод – набор на клавиатуре - [предположительно] действующего (активного, верного) общего идентификатора пользователя - вроде номера паспорта; в данном случае – это номер клиента системы социального страхования SSN (social security number),
- ввод [предположительно] действующего персонального идентификатора пользователя как клиента данного банка PIN (personal id number)
- отсылку данной информации на проверку (validation).

Описание возможных [внутренних] состояний авторизации начинается с именования. Getting SSN – получить SSN, Getting PIN – получить PIN, Validating – проверка и Rejecting – отказ, в случае неудачи сценария «по умолчанию». Далее определяются переходы из состояния в состояние, для *каждой пары* состояний и *всевозможных комбинаций* сообщений.

Что описывается в теории множеств в терминах фундаментального понятия *декартового произведения* множеств.

Существенным отличием диаграмм состояний в UML (по сравнению с классической теорией автоматов) является возможность описания *структурных состояний*, т.е. состояний, которые сами являются системами. Тонким моментом является здесь переход сообщений через границы системы. Контуры принятого здесь UML подхода намечены нами при обсуждении понятия *прецедента.*

Но возможности языков всегда обусловлены необходимостью решения некоторой проблемы и прежде чем «влезать внутрь ящика (черного или серого)» стоит задуматься – какой именно? Классическая теория сложности вычислений определяет «информационный взрыв» - обилие информации, с которой человек не может справиться (manage) в терминах экспонент (например 2^n). Что не подлежит сомнению, но… В практике программирования уже полиномы (в данном случае, минимально n^2) – уже в общем случае *не управляемы – не контролируемы, анализируемы, проверяемы и т.п.* Для того, чтобы взглянуть в лицо хаосу, достаточно вообразить себе автомат со 50 состояниями. Кажется, немного… если бы не 2500 возможных переходов.

Это касается *любых* диаграмм – бинарных и иных графов, *вне зависимости* от интерпретации, типа задачи описания и способа ее решения. Надеюсь, сейчас для нас становиться куда более предметным не только понимание происхождения UML, но и всех иных структурных методов в

программировании и в целом – назначение математических формализмов. А ты говорил - «пустая абстракция, оторванная от жизни»… Чьей жизни? - Только не от жизни описателя моделей.

Обратим внимание на интересную особенность примера. Авторизация – не вещь (физический объект), не система таких объектов, но *процесс*. И вместе с тем, конечно – объект нашего текущего внимания и описания.

Диаграммы деятельности – activity diagrams.

«Плавательные дорожки» - swimlines

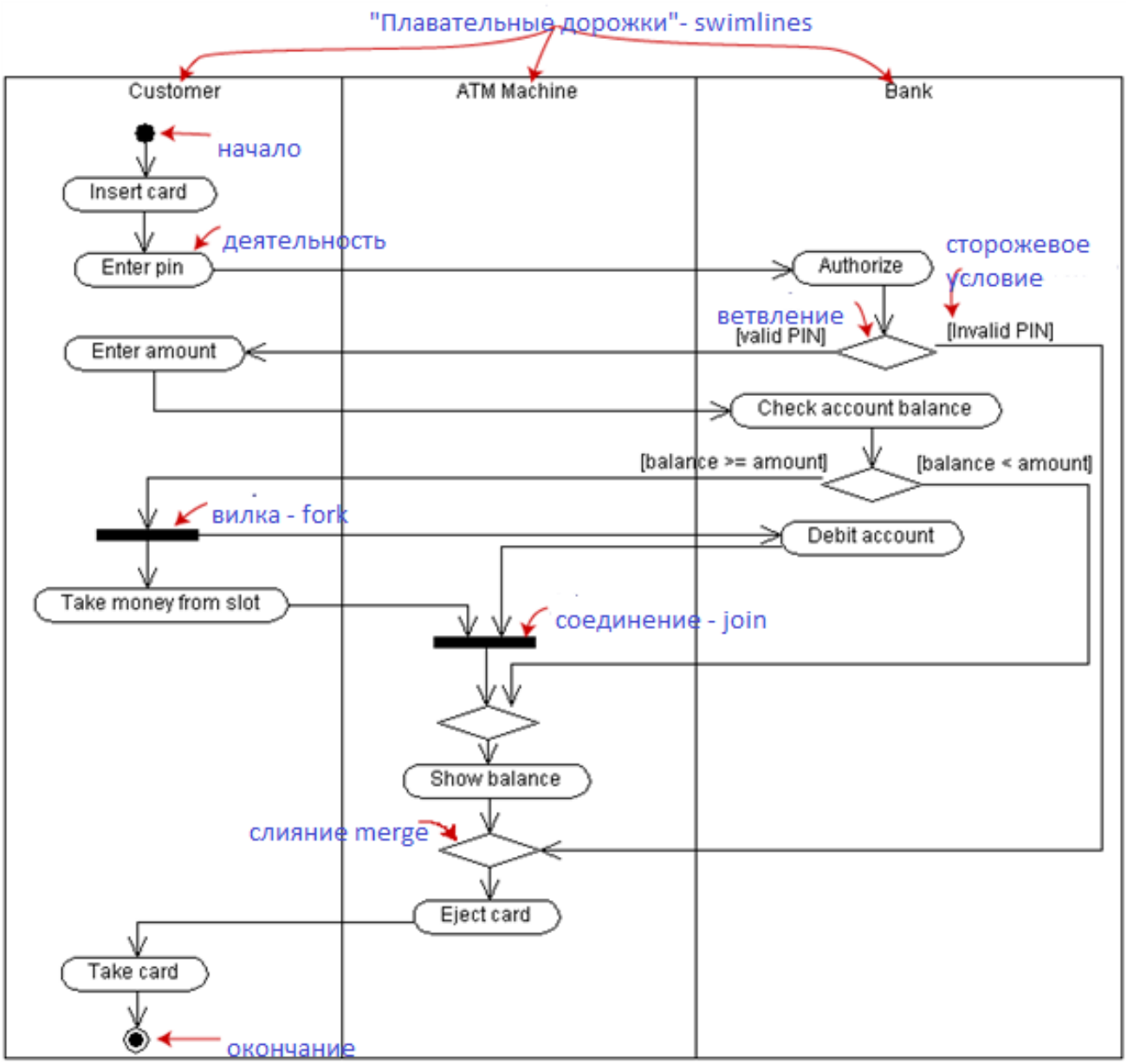
gkf

Диаграммы деятельности – по сути своей, хорошо знакомые нам блок-схемы с рядом особенностей, отражающих изменение процедурного подхода под влиянием ООП и тех идей частичности описаний, которые мы связали ранее с итеративным подходом.

В особенности, трактовкой алгоритмов как совокупного описания множества успешных и неудачных сценариев прецедентов – в сложных случаях описываемых лишь частично на каждой стадии разработки.

Наиболее заметные отличия видны на следующем примере.

Пример. Схема работы банкомата (ATM Machine) по обслуживанию клиента (Customer) банка (Bank).



По-прежнему условия (предикаты) изображаются ромбами, методы ([пользовательские] операторы) – прямоугольниками с закругленными концами.

Сторожевое условие (guard condition) трактуется здесь как проверка необходимости выполнения неправильного хода событий – т.е. неудачного сценария. По сути – необходимости обработки логического исключения (exception) (вызванного сбоем «времени логики», а не «времени исполнения»).

Плавательные дорожки (swimlines) разделяют «пловцов» - объекты, чьи методы исполняются. Как видно из примера, фактически они трактуются здесь как ***субъекты деятельности***, исполняющие некоторую общую работу.

В резком контрасте с ранним императивным и более поздним процедурным подходами. В первом объект *исполняет* команду, во втором – действия *выполняются.* В обоих случаях - без явного указания субъекта (актора), инициирующего (начинающего, активирующего) действие (операцию). Блок-схемы (flowcharts, т.е. диаграммы потоков) выражают поток передачи управления – с неявным субъектом управления и явным объектом управления. В данном случае уместнее говорить о субъектах – в рамках распределения и передаче полномочий и *ответственности* за ту или иную часть работы.

Продолжая разбор этих случаев, можно допустить две интерпретации примера, в зависимости от конкретной предметной области.

- Банкомат и банк вместе составляют систему-сервер, выполняющую работу для инициирующего ее клиента (актора) – имеющего приоритет, отраженный в порядке следования субъектов.
- Все субъекты исполняют некоторую общую работу, преследуя интересы взаимовыгодного сотрудничества. Клиент инициирует работу не потому, что он имеет абсолютный приоритет – но лишь потому, что «в этот раз кто-то все-таки должен начать первым». Иными словами – в других диаграммах возможен иной порядок расположения субъектов.

Конечно, для данной предметной области больше подходит первая, характерная для современных «клиент-серверных» подходов интерпретация.

Мы привыкли рассматривать подобные диаграммы как законченные описания. Но в контексте рассматриваемого подхода, существует еще одна важная трактовка диаграмм, исходящая из возможностей описателя. Можно предположить, что мы находимся на той стадии разработки, когда взаимоотношение субъектов еще не определены или нам неизвестны. *Как-то* (пока) надо описать.

Немедленным следствием возникающего здесь «разделения труда» между несколькими акторами, является необходимость согласования возникающего параллелизма выполнения действий. На диаграмме такое согласование обозначается новым типом ветвления – вилка (fork) и соединения (join) процессов.

Параллелизм – крайне сложная тема, допускающая много толкований. В контексте необходимости описания предметной области, естественно продолжить тему разделения труда между субъектами исходя из необходимости достижения общего результата. В этом «многопроцессорном» варианте порядок действий неважен.

Исходя из возможностей описателя, возможна иная трактовка. Мы знаем, что данный процесс когда-то должен начаться и когда-то завершиться. Но [пока] не знаем точно порядка действий.

Задача описания предметной области.

Модель предметной области отображает концептуальные классы – основные, с точки зрения моделирующего, классы понятий, относящиеся к предметной области (не к программной реализации).

В случае моделирования бизнеса – деятельности крупных предприятий - обычно говорят о бизнес-логике, бизнес-моделировании и т.п. На языке UML модель предметной области представляется в виде набора *диаграмм классов,* на которых свойства и методы проименованы, но в общем случае не реализованы. Иначе говоря – по мере возможности, определены интерфейсы классов.

Формально говоря, в рассматриваемом подходе построение диаграмм классов в основных своих принципах мало отличается от принятого в «обычном ООП». Самую существенную разницу привносят рассмотренные нами выше мотивы разделения труда. По мере возможности, программист *соучаствует* в предыдущих этапах, но *отвечает* за этап программной реализации. Внутри него, он волен выделять классы, исходя из необходимости решения внутренних проблем программной реализации – в рамках своей роли, он этого делать не может. Классы предметной области заданы реалиями предметной области.

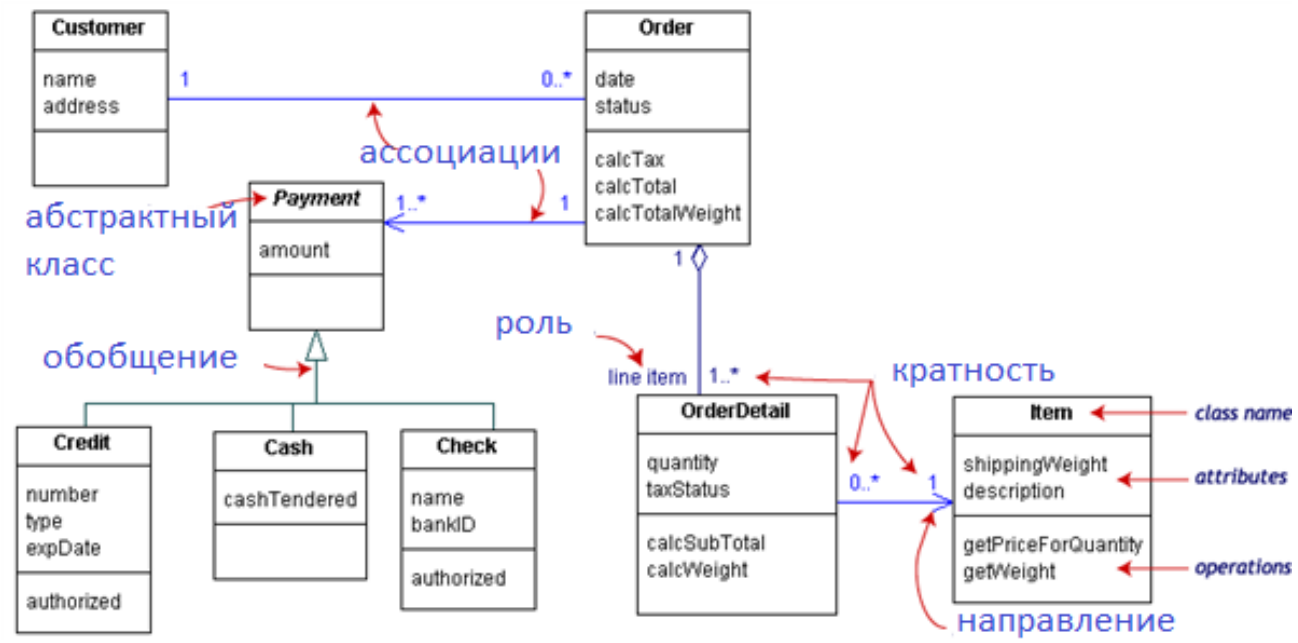
Правда, в предлагаемом подходе это уже не означает односторонней подчиненности – по мере острой необходимости, он (изредка) - в качестве *обратной связи* (feedback) - может просить об изменении задачи в силу невозможности выполнить задачу в заданных условиях – например, при заданных ресурсах времени (т.е. в срок).

Напомним (в качестве пищи для размышлений), что исторически понятие [петли] обратной связи (feedback loop) появилось в кибернетике (точнее, в теории автоматов) именно в связи с необходимостью формального описания сложного поведения, характерного для *живых* существ. А не *реальных автоматов*, наиболее частых (в силу простоты) примеров в сегодняшней литературе по программированию.

Модель предметной области отображает:

- объекты предметной области или концептуальные классы;
- *ассоциации* между концептуальными классами;
- атрибуты и операции концептуальных классов (имена свойств и методов).

Пример диаграммы классов. Схема оплаты заказов.



Пример подразумевает примерно следующий (успешный) сценарий оплаты (payment) заказа (order) клиентом (customer) некоторой торгово-транспортной компании: «Клиент, заказавший список товаров (см. item – характеризацию свойств товара в отдельном пункте заказа) может оплатить заказ кредитной карточкой (credit), наличными (cash) или банковским чеком (check)»

Обратим внимание, что диаграммы классов, помимо самих классов, описывают в виде графа и разнообразные реальные связи (association) между классами.

Иначе говоря – при описании структуры классов мы применяем идущий от теории множеств и отношений (relation) *реляционный* подход, рассмотренный нами ранее при освоении реляционных баз данных. Отметим здесь возможность указания кратности связей, описывающих точнее (по сравнению с «один» и «много»), сколько объектов (экземпляров, instance) одного класса может находиться в данном отношении с другим классом. 0..1 - не более одного, 1 - ровно 1, 1* - не менее одного и т.п.

Некоторую путаницу здесь обычно вносит разнообразие типов связей. Здесь связь (association) – любая возможная связь между классами, вне зависимости от того, какое дальнейшее отношение между объектами оно подразумевает.

Одни из них – как и ранее - относятся к описанию структуры данных. Таковы, например, связь Customer-Order и связь «часть-целое» (aggregation, отношение агрегирования) между классами Order и OrderDetail.

Другие (generalization – обобщение) – к описанию взаимосвязи классов по наследованию. См. в примере связь класса Payment с подклассами Cash, Check и Credit. Это статическая (фиксированная) связь *между классами*, подразумевающая возможность *динамической* (изменяемой) связи между *объектом и классом*. В каждый момент времени объект относим к некоторому конкретному классу – но затем эта принадлежность объекта может измениться. Что собственно и отличает понятие класса в ООП от типа в процедурном программировании.

Закладки на будущее. Помимо «просто связей», своеобразной особенностью UML является возможность определения иных, пользовательских типов связей, а также группировки классов в *пакеты* – по иным, отличным от рассмотренных нами ранее соображениям. В том числе, будущим, пока нам неизвестным.

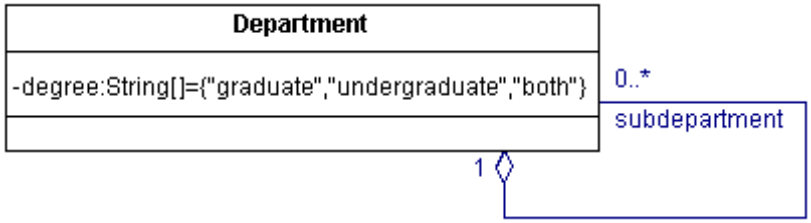
Обычно пакеты (равно как изначально модули и компоненты) связывают с решением специальных задач обработки программ в качестве данных (например, компиляции), но мы вольны их использовать и по каким-то иным, оригинальным соображением (если они действительно оригинальны).

Диаграммы объектов. *

Диаграммы объектов используются значительно реже диаграмм классов. Фактически, мы спускаемся здесь до уровня программной реализации. Применять их полезно, когда графическое представление помогает обратить внимание участников разработки на те особенности реализации, которые тяжело распознать по тексту программы.

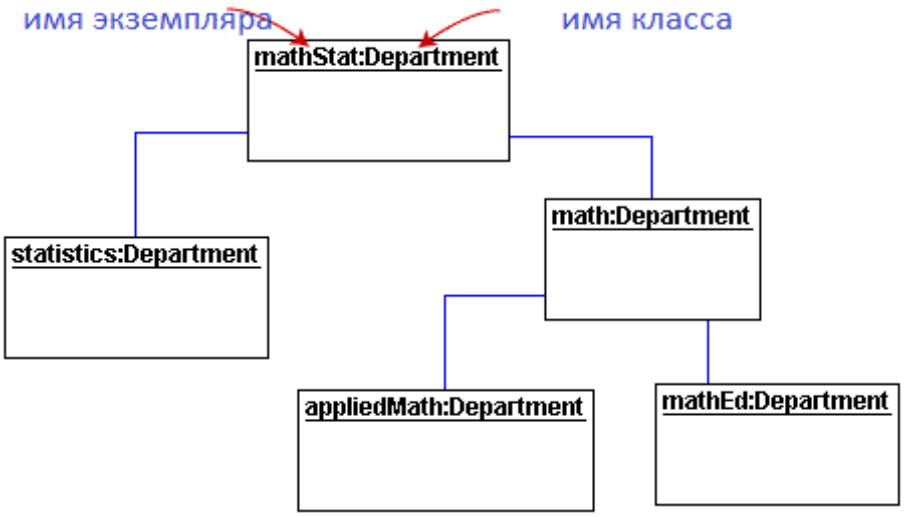
В качестве неформальной мотивации, легко представить температуру в коллективе при возвращении из отпуска программиста, не посвятившего коллег в особенности реализации. Как известно, по закону Паркинсона (он же закон падения бутерброда) необходимость коррекции программного кода происходят как раз во время отсутствия написавшего его программиста! Во всяком случае, такие примеры надолго запоминаются…

Пример 1. Рекурсивные связи



Здесь предполагается иерархия классов, описывающих подразделения (departments) университета.

Пример 2. Унаследованные статические связи.



mathStat, math, statistics, appliedMath, mathEd – объекты, ссылки на класс Department

Вероятно, более сложным и важным примером является описание далеко не всегда очевидных возможностей изменения в программе принадлежности одного объекта к различным классам – относительно иерархии наследования.

Диаграммы компонент и диаграммы развертывания –

Component and deployment diagrams

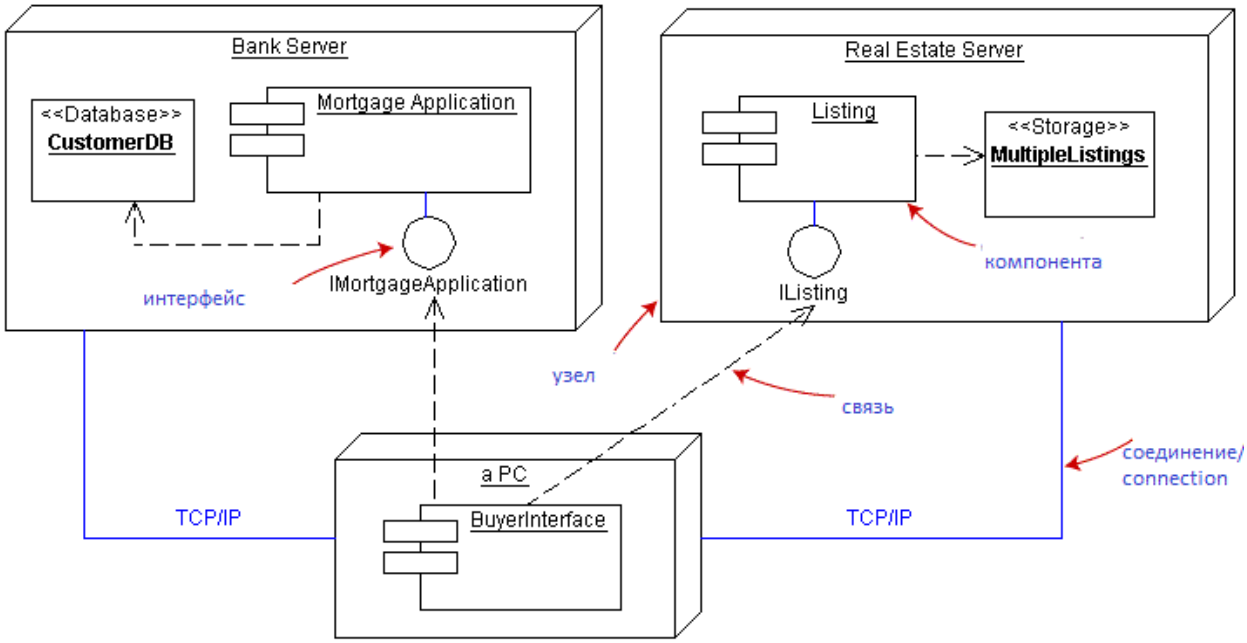
Компонента [архитектуры программной системы] – модуль, часть [текста] программы, традиционно связываемый с автоматизацией, алгоритмической обработкой программного кода (программы как данные). При использовании UML они трактуются как физические аналоги диаграмм классов и описываются аналогичным образом. Иначе говоря, диаграммы компонентов - формальный аналог диаграмм классов, предназначенный для описания программной архитектуры, строения программной системы.

И в этом качестве – не столь важны для прикладного программиста. Но еще раз обратим здесь внимание на один важный момент. Мы привыкли, что задачи программирования ставятся заказчиком - задаются извне программирования. Но, несомненно, весьма большая часть задач рождается внутри самого программирования – касается ли они software или hardware, программной или аппаратной его части. Так, изначально модули трактовались скорее как единицы компиляции (compilation, сборки) – но проложили путь объектному программированию в качестве *универсального* способа моделирования. То, что в модульном программировании трактуется как именованная константа (имя модуля обозначает фиксированный интерфейс и его реализацию), в объектном становиться именем переменной (имя объекта обозначает переменный интерфейс и реализацию).

В компонентном программировании понятие компоненты идет еще дальше и имеет более четкое определение. Просто – именованный, но [пока] никак не реализованный интерфейс. Родившись в среде чисто системных задач (особенно важно здесь вспомнить задачу поддержки версий), она становиться здесь инструментом решения *любых* задач (повышенной сложности).

Диаграммы развертывания предназначены главным образом для описания *физической конфигурации* (архитектуры) программного и аппаратного обеспечения. Так, следующий пример показывает взаимосвязи между программными и аппаратными компонентами программной системы по обработке транзакций по сделкам с недвижимостью.

Пример. Физическая среда поддержки продажи недвижимости.



Здесь Real Estate Server и Bank Server – сервера агентства по продаже недвижимости и ипотечного банка, предоставляющего ссуду своему клиенту (customer) в ответ на его заявление (application). PC – клиентский компьютер, позволяющий пользователю разрабатываемой нами программной системы делать запросы как в банк, так и агентство, в соответствии с имеющейся в его базе данных списком (listing) предложений.

Аппаратные устройства представляются узлами (nodes). Каждая программная компонента связана с некоторым узлом и представляется на диаграмме в виде прямоугольника с двойной петлей в верхнем левом углу. Отдельным значком (кружок) выделен *интерфейс* – набор сервисных функций (возможностей поддержки) предоставляемых соотвествующимим системами клиенту.

Концептуальное понимание термина прозрачно. Клиент не может знать сложное внутренное устройство систем (хотя и должен знать, что оно сложно и потому ценить труд разработчиков). Больше того. Не будучи высоквалифицированным специалистом, он *не должен* вмешиваться в их функционирование – причем даже тогда, когда для этого есть *физическая возможность*. Это дает косвенное определение моральных ограничений, честного клиента и попросту – *культурного человека*. Не хакера.

Однако, для нормального функционирования всей системы он вправе получить доступ к интерфейсу – описание которого ему, скорее всего, известно лишь в форме инструкции пользователя в *неформальном* текстовом или графическом виде, т.е. на *обычном* человеческом языке. Проще говоря, кто-то все-таки должен ему обяснить азы - «что, как и почему».

По сути единственным существенным нововедением здесь является специальное обозначение символа для *интерфейса*, рассматриваемого отдельно от его программной или физической реализации (см. обсуждение выше). По всей видимости, Unified Modelling Language еще нуждается в дальнейшей унификации. Что ж, все люди – люди, в том числе и создатели языков моделирования.

Но закончить наш обзор UML я хотел бы не этим – но обращением к читателю.

Глава 4. Пример разработки игрового приложения.[8]

Вместо введения, от автора примера: *Возможно, может быть, иногда, скорее всего,…* – вот что определяет подходы ОО АП.

1. *Концептуализация системы:* идея приложения – игра на развитие памяти.
2. *Аналитическая модель* – это точное, четкое представление задачи, позволяющее отвечать на вопросы и строить решения.
3. *Проектная модель* – это реализация решений задач, понятых на этапе анализа.

Краткое описание игры: на экране появляются фигуры различной формы и цвета, игрок щелкает по последней появившейся фигуре, если выбор сделан правильно, то появляется новая фигура. Суть игры - в том, что чем больше фигур на экране, тем труднее определиться с выбором.

Сложность. Игра должна поддерживать стандартные варианты сложности (например, легкий, средний, трудный). Пользователь должен иметь возможность настроить свою сложность игры.

Уровни. Вне зависимости от типа сложности игра должна поддерживать уровни. Уровень определяет максимальное количество фигур на экране. Т.е. количество фигур – это функция от уровня.

Подсказки. В случае затруднения выбора фигуры игрок может выбрать подсказку.

База данных. По возможности в системе должна храниться информация об игроках.

Дополнительные требования. Необходимо все делать так, чтобы игра была масштабируемой, т.е. пользователь мог подключать к своей игре различные варианты сложности.

Воспользуемся итерационным подходом по разработке приложения. На первой итерации реализуем только небольшую часть требований, так как это обеспечит наиболее раннюю обратную связь.

Модель вариантов использования, прецедентов

Рассмотрим прецеденты *Игра* и *Настройка сложности.*

Основными артефактами при моделировании прецедентов являются диаграммы прецедентов, прецеденты, диаграммы последовательностей прецедентов. Многие новички в области ООА/П при моделировании прецедентов делают акцент лишь на диаграммах прецедентов, что по своей сути не приносит существенной пользы.

---

[8] Автор главы – Анастасия Сабирзянова. Я внес лишь несущественные стилистические правки – поскольку мне нравиться ее живой и честный стиль изложения процесса разработки. Понятно, при желании она могла бы слукавить… Ведь Анастасия отлично закончила факультет ВМК КГУ и имеет уже достаточно большой опыт практического разработки коммерческих приложений. Впрочем - наверное, именно поэтому и не смогла… Н.Б.

Необходимо понимать, что диаграммы отображают лишь *имена*, названия *прецедентов*, а не сами прецеденты.

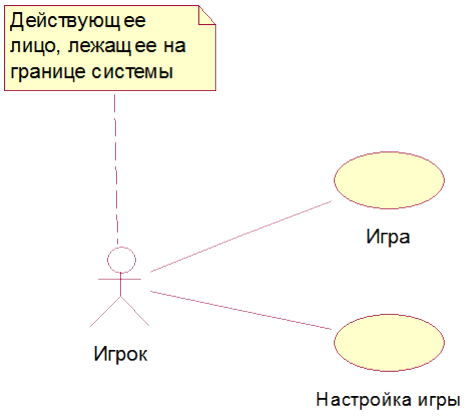


Рисунок 1. Диаграмма прецедентов

Напомним, что *прецедент* – это рассказ об использовании системы действующим лицом, который в дальнейшем можно для большей наглядности спроектировать в диаграммы последовательностей прецедентов.

Идентификация действующих лиц. Единственным действующим лицо является Игрок. Игрок не является частью системы, потому как система не может управлять его действиями, мы может ожидать только некоторую последовательность действий от игрока.

Идентификация начальных событий. Определим, какие события инициируют прецеденты *Игра* и *Настройка сложности*. В данном случае начальным событием является запрос соответствующих услуг, предоставляемых системой.

Идентификация конечных событий. Также следует определить конечные события. Для прецедента Настройка сложности таковым будет установка параметров сложности игры, здесь все понятно. Для прецедента *Игра* ситуация иная: вариант использования *Игра* может продолжаться до тех пор, пока игра не будет выиграна или проиграна. Конечным событием прецедента Игра выберем событие, когда игроку надоест данный сервис, т.е. выход из игры.

Предварительные рекомендации: в качестве прецедентов следует выбирать только полные транзакции, несущие смысл для пользователя системы. Хотя ведущие методологи уже определили стиль написания прецедентов, это вовсе не означает, что мы должны строго следовать ему (или не следовать лишь по причине несогласия). Не забудем, что прецеденты отвечают на вопрос “что”, а не на вопрос “как”, т.е. не надо в прецеденте описывать каким образом достигается тот или иной результат.

Прецедент *Игра.*

Основной исполнитель. Игрок.
Заинтересованные лица и их требования. Игрок хочет посредством игры развить свою память.
Предусловия. Сложность игры определена, выбран уровень.
Результат. Игра пройдена. Игра не пройдена. Игрок повысил свой уровень.
Основной успешный сценарий.
1. Игрок запускает игру.
2. Система отображает фигуру.

3. Игрок щелкает по фигуре, которая, по его мнению, появилась последней на экране.
4.
а) Если игрок ошибся в выборе фигуры, то игра заканчивается проигрышем.
б) Если игрок сделал верный выбор, то система переводит игрока на новый уровень, при выполнении всех условий. В любом случае, возвращаемся на этап 3.

Прецедент *Настройка сложности.*

Основной исполнитель. Игрок.
Заинтересованные лица и их требования. Игрок хочет выбрать одну из стандартных сложностей или настроить свою.
Предусловия. нет
Результат. Сложность игры определена.
Основной успешный сценарий.
1. а. Игрок выбирает одну из предустановленных сложностей игры.
1. б. Игрок настаивает свои метрики: количество типов фигур, количество цветов фигур, таймер.
2. Нажимает ОК.

Многим коллегам может показаться абсурдность моделирования такой маленькой задачи. Но в том-то и прелесть, что маленький пример лишь иллюстрирует принципы, не отвлекая нас от самих принципов, рассматриваемого в данном пособии. Иначе, увлекшись самой игрой (предметной областью) можно забыть, то зачем мы здесь сегодня собрались.

Теперь *спроектируем* прецеденты в диаграммы последовательностей прецедентов. Хотя в UML и нет такого понятия, оно позволит нам воссоздать некое логическое звено в рассуждениях по построению моделей.

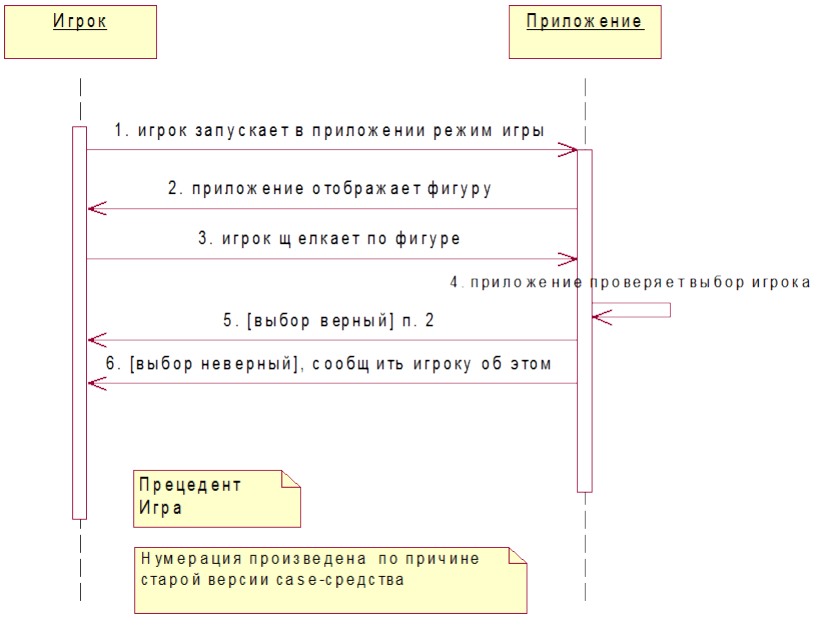


Рисунок 2. Диаграмма последовательностей прецедента *Игра*

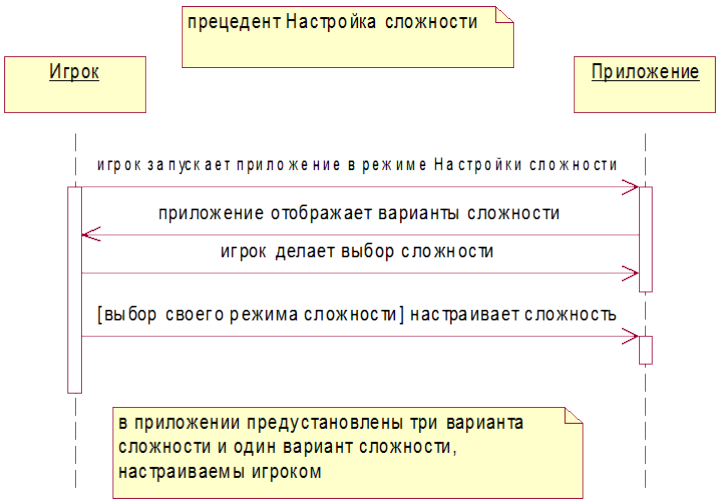


Рисунок 3. Диаграмма последовательностей прецедента *Настройка сложности*

Анализ приложения

В анализе предметной области нет необходимости, потому как, приложение не имеет аналогов в реальном мире. Проведем анализ приложения: выделим аспекты программного приложения, видимые пользователю и отражающие его точку зрения. Но анализ предметной области можно провести по аналогичной схеме.

Модель классов приложения.

Одним из артефактов анализа приложения являются диаграммы классов приложения. Это означает, что в моделях нежелательно отображать классы, относящиеся к уровню реализации (списки, деревья и т.п. абстрактные типы данных).

Классы. В качестве источника потенциальных классов выделим прецеденты, но это не единственный источник. Обычно классы соответствуют существительным.[9] Например, из предложения – *система (экран) отображает фигуру,* можно отобрать потенциальные классы *Система* (экран), *Фигура.*

Проблема: одна часть речи может плавно перетекать в другую. Не стоит долго мучиться над вопросом *соотношения ключевых слов со списком потенциальных классов.* Модели классов будут уточняться со временем, и абсолютно корректную модель построить сразу, чаще всего, невозможно. Хотя это и не означает, что это бесполезное занятие. [10]

Выбираем потенциальные классы для нашего примера: *игрок, игра, фигура, сложность, уровень, эллипс, прямоугольник.*

---

[9] Что верно – при учете необходимости трактовать в таких случаях иные части речи как существительные. Вспомним пример «Авторизация», где мы трактуем операцию *авторизовать* как процесс. Н.Б.

[10] *Ключевые слова*: имеется в виду *глоссарий* – компактный список основных терминов, описывающих предметную область. В остальном - мнение Анастасии хорошо ложиться в концепцию ОО АП. Модель – не только и не столько конечный результат разработки. Это текущий результат понимания проблем, с ней связанных. Отсюда – необходимость и полезность проб. Н.Б.

Замечание: вообще говоря, для каждого прецедента обычно приходится строить свою модель классов. Но так как приложение небольшое, то построим единую модель классов для обоих прецедентов.

Свойства. Определим свойства классов. Свойства – это такие концепции, которые не обладают собственной индивидуальностью. В данном случае, например, цвет фигуры (но если бы мы строили графический редактор, то цвет, скорее всего, обладал бы такой индивидуальностью). Такие свойства можно найти в списке потенциальных классов.

Типичной ошибкой при выделении свойств является запись в свойства классов объектов, которые представляют собой сложные объекты. Например, можно в качестве свойства класса *игра* указать сложность, но класс *сложность* обладает своим уникальным поведением, для него важна индивидуальность, поэтому эти два класса лучше связать между собой посредством ассоциации.

Замечание: если вы строите модель приложения или предметной области, то также не стоит указывать в качестве свойств внутренний идентификатор. Такого рода атрибут не имеет значения в реальном мире, а приносит лишь удобство для реализации.[11]

Ассоциации. Далее между выбранными классами установим связи – ассоциации. Связи можно извлечь из тех же прецедентов, глаголы – это *потенциальные* связи. Иначе говоря – как и ранее - проблема понимания слов естественного языка в том или ином качестве остается.

Например, игра определяется сложностью.

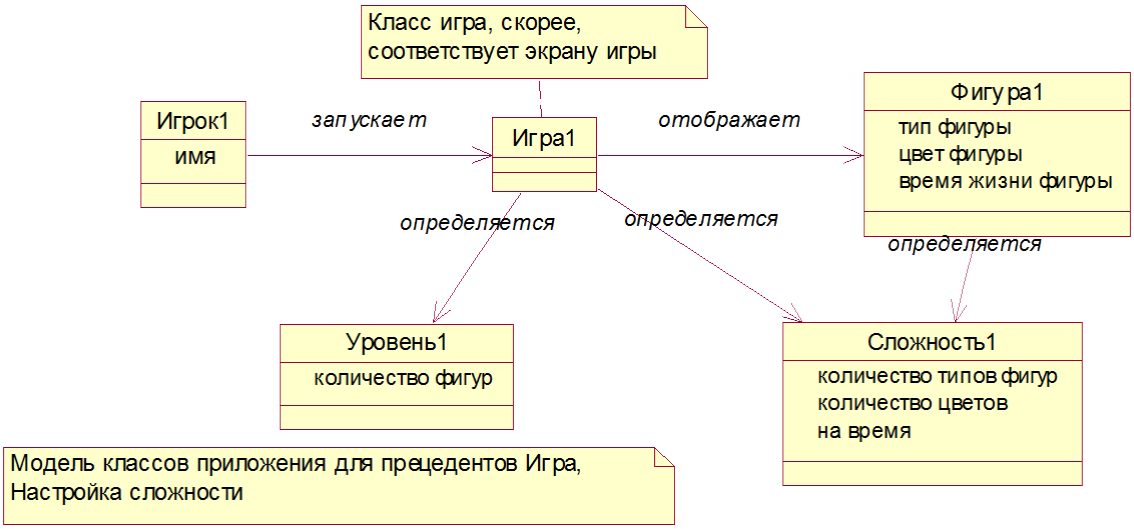


Рисунок 4. Диаграмма классов приложения

---

[11] Не всегда, не абсолютно. Описание предметной области имеет приоритет – *по умолчанию*, при прочих равных условиях. Но напомним, главной задачей итеративного метода является (более-менее) равномерное продвижение *всех* видов работ. Внутренний идентификатор – например, первичный ключ таблицы БД - также может быть выбран далеко не случайно, но исходя из задач программной реализации. Можно представить себе крайнюю *исключительную* ситуацию, когда из-за проблем реализации приходиться ограничивать описание предметной области. Разница – в том, что обоснования требуют единичные исключения, а не общие правила. Н.Б.

На диаграмме присутствуют две ассоциации - *игра* ***определяется*** *сложностью, фигура* ***определяется*** *сложностью.*

Зная шаблон проектирования *Низкое связывание*, вижу потенциальную ошибку, что одна из этих ассоциаций лишняя, но какая именно - решу на этапе проектирования. [12]

Операции. При моделировании классов сразу можно выделить наиболее очевидные операции. Например, фигуру можно прорисовать.

Рекомендация: вообще говоря, диаграмма классов строится параллельно с диаграммой взаимодействия, на диаграмме взаимодействия отображаются сообщения, посылаемые между объектами. Сообщение – это вызов функции класса. Так что операции классов лучше брать из диаграмм взаимодействия. [13]

Обобщения. Далее организуем структуру иерархии классов по наследованию путем выявления общей структуры. Общую структуру можно выделить у классов *Эллипс, Прямоугольник* и т.д. в класс *Фигура*. К такому обобщению можно прийти из двух соображений. Обобщение снизу вверх: в суперклассе *Фигура* должны быть определены общие черты для классов *Эллипс, Прямоугольник* (координаты, цвет, время жизни). Конкретизация сверху вниз: выделим из описания игры именные группы, состоящие из различных обстоятельств с указанным существительным. Например, «фигура прямоугольник» или «фигура эллипс».

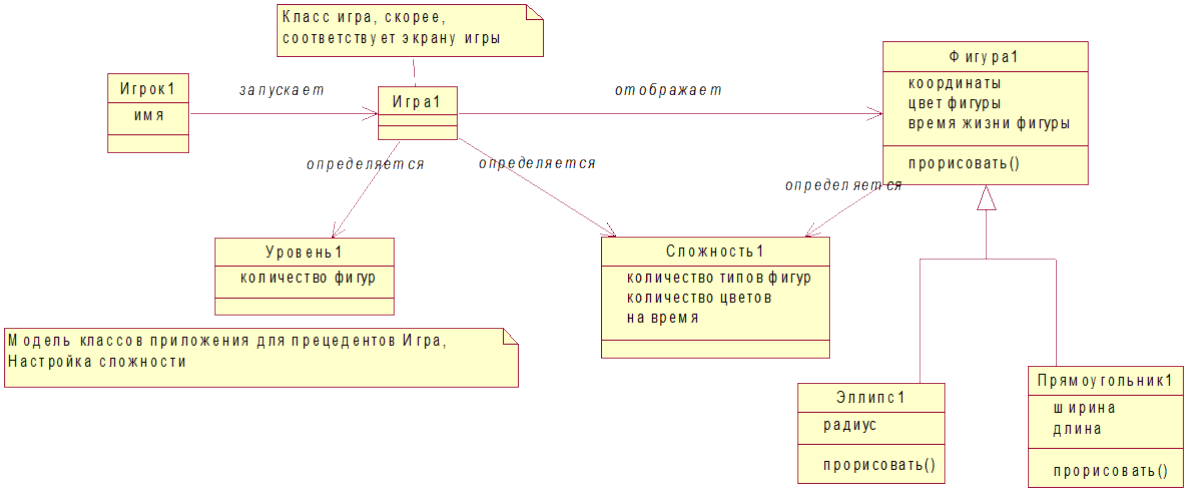


Рисунок 5. Диаграмма классов приложения с обобщением

Замечание: возможно, это обобщение даст в дальнейшем точку расширения типов фигур, отображаемых на экране.

Резюме: модели классов полезны не только для определения структур данных. Прослеживание моделей классов позволяет выразить некоторые виды поведения, т.е. прецеденты.

---

[12] Мы не затрагивали в данном пособии крайне полезное понятие *шаблона проектирования* (к тому же, если честно - я не помню такого шаблона). Пока – до знакомства с ним по литературе (см. например [Larman]) читатель здесь и далее может читать его так: «*ведущие разработчики с большим опытом рекомендуют создать в данной ситуации следующий класс».* Н.Б.

[13] В простых случаях. В общем случае, сообщение *ссылается* на вызов, и соотношение сообщений и операций приходиться задавать особо. Н.Б.

Модель состояний приложения

Еще одним артефактом этапа анализа приложения являются диаграммы состояний. Диаграммы состояний требуются для объектов предметной области, которые имеют нетривиальное, в том числе циклическое поведение. Большинство же классов в нашем случае не требует использования диаграмм состояний, для их описания достаточно списка операций. Например, таковым является класс Фигура - для данного приложения неважно, в каком состоянии находиться фигура.

Выделим классы, обладающие разными состояниями. Таковым, например, является класс *Игра*. Этот класс может находиться в состояниях *Игры, Победы* и т.д.

Выделим состояния класса *Игры*. Для этого мысленно представим себе объект класса *Игра,* пытаясь понять различия этого объекта в состояниях *Игра, Победа, Проигрыш*. В состоянии Игра объект прорисовывает фигуру, а в остальных состояниях – нет. Иначе говоря, ассоциация между классами *Игра* и *Фигура* присутствует в состоянии *Игра*, а в состояниях *Победа* и *Проигрыш* эта ассоциация отсутствует (тем самым состояния *Победа* и *Проигрыш* для меня неотличимы).

По причине маленького масштаба примера, может показаться, что нет необходимости в данных моделях. На практике в таком случае можно отнести их к программной реализации, но здесь возникает желание привести пример моделирования диаграммы состояний. [14]

Выделение событий. После выделения состояний необходимо выделить события, которые вызывают переход между состояниями.
Категоризация событий:

- Событие сигнала – это событие получения или отправки сигнала.
- Событие изменения – это событие, вызванное выполнением логического выражения.
- Событие времени – это событие, вызванное достижения момента абсолютного времени или истечением временного интервала.

Событие изменения *Выполнение условий игры* переводит объект класса *Игра* из состояния *Игры* в состояние *Победа*. Событие изменения *Невыполнение условий игры* переводит объект класса *Игра* из состояния *Игры* в состояние *Проигрыш*. Тем самым разные события приводят объект в разные состояния.

Событие – это точки на линии времени, а состояния – это интервалы.

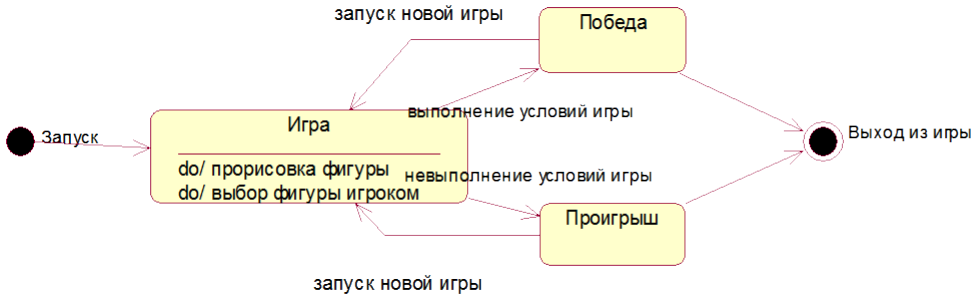


Рисунок 6. Диаграмма состояний для класса *Игра*

[14] Равно как и желание предупредить возможные ошибки на более ранней стадии. Н.Б.

Внутри состояний также могут происходить события, но эти события не вызывают переходов между состояниями. Например, в состоянии Игра происходят события прорисовки фигур, события выбора игроком фигуры.

Диаграмма состояний объекта – это автомат (граф), вершинами которого являются состояния, а ребрами – события.

Резюме: на этом мы заканчиваем процесс проведения анализа предметной области приложения. После этого следует еще раз пересмотреть полученные модели, подкорректировать при необходимости. Основная цель анализа – определение проблемы, не давая особых преимуществ какому-либо варианту реализации.

Проектирование приложения

Проектирование системы – это выбор высокоуровневой стратегии решения задач.

Модель взаимодействий приложения

Вернемся к прецедентам, потому как модели взаимодействий строятся на модели прецедентов. Артефакт диаграммы последовательностей прецедентов отвечали на вопрос “что”, в данном разделе построим диаграммы последовательностей, отвечающие на вопрос “как” при решении задач варианта использования Игра.

Обратимся к сценарию прецедента Игра, инициализация которого происходит после события запуска этого сервиса Игроком.

Инициализация игры. Согласно шаблонам проектирования необходим класс-контроллер, который будет принимать на себя “огонь” событий от пользователя. Пусть класс **Игра** отвечает за запуск игры.

Создание фигуры включает в себя последовательность действий: случайное определение параметров будущей фигуры и создание фигуры согласно параметрам.

Определение значений атрибутов будущей фигуры. Так как фигура определяется атрибутами (тип, цвет, положение), а значения атрибутов должны быть случайны, то необходим класс, который будет отвечать за случайное образование значений атрибутов – класс *Алгоритм*. Значения атрибутов зависят от сложности игры. Иначе говоря, здесь мы замечаем связывание классов *Алгоритм* и *Сложность* - т.е. значения атрибутов есть функция от *Сложности.* Тем самым *Сложность* передается классу *Алгоритм* в качестве параметра. Значит, *Сложность* надо передать в качестве параметра при запуске игры.

Создание фигуры. Параметры будущей фигуры определены, но какой класс будет отвечать за создание фигуры? Здесь применим шаблон проектирования Creator, согласно которому создавать будет тот класс, который обладает большей информацией об объекте, таковым является класс Алгоритм. Класс Алгоритм возвратит классу Игра положение новой фигуры. После создания фигуры класс переходит в режим ожидания выбора пользователем фигуры.

Выбор фигуры. Обработку этого события примет на себя класс-контроллер Игра. Проверку верности выбора фигуры игроком производи сам класс-контроллер, так как он обладает всей информацией для этого. Если Игрок сделал верный выбор, то система опять прорисовывает новую фигуру, иначе игра переходит в состояние Проигрыша.

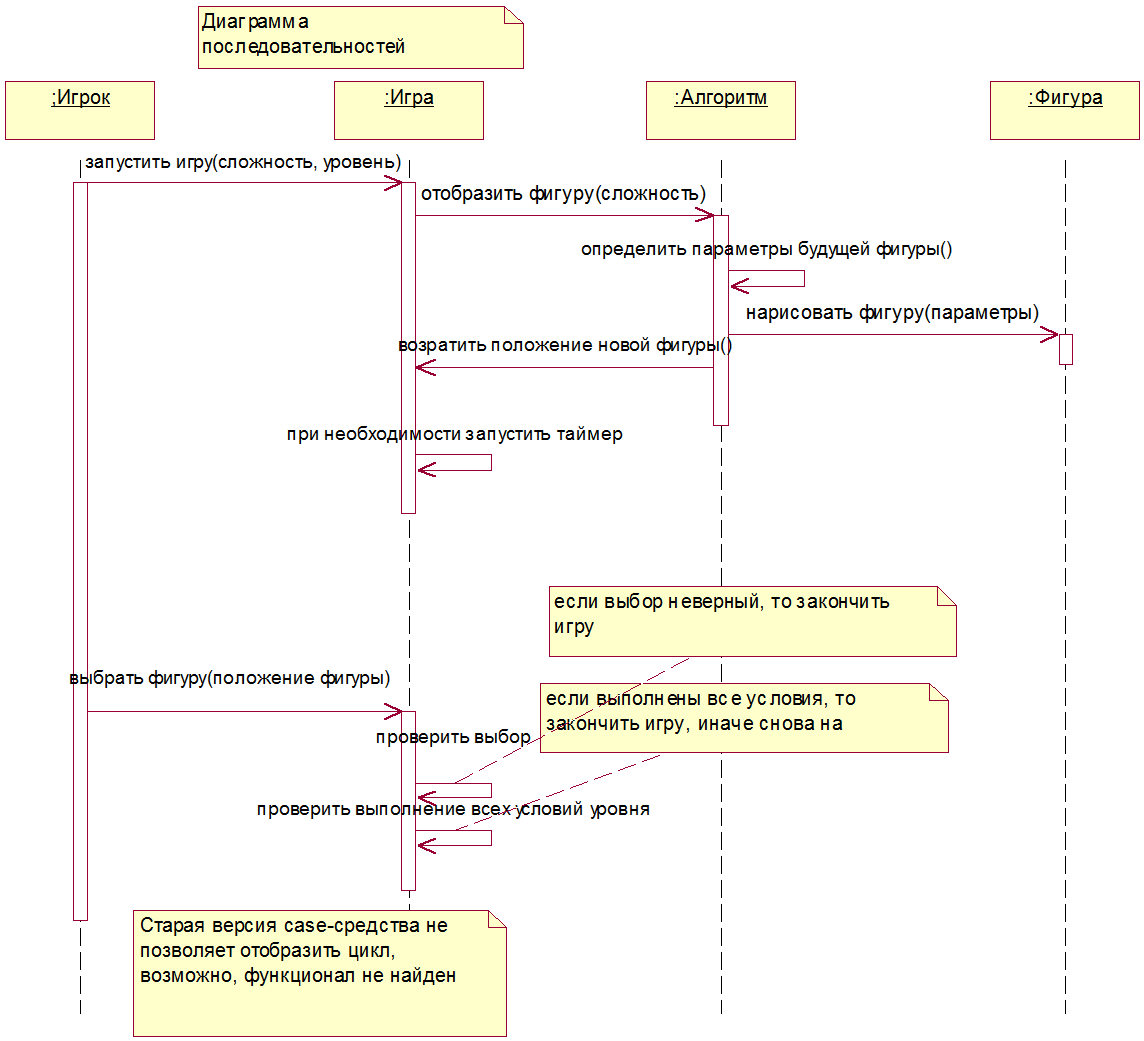


Рисунок 7. Диаграмма последовательностей прецедента *Игра*

Резюме: не все со мной могут согласиться при принятом здесь подходе к анализу и проектированию данной конкретной системы. Вообще говоря, он отражает достаточно субъективный взгляд на вещи.[15]

---

[15] Я не случайно выделил выше *субъекты* действий. В главном, я солидарен с Анастасией в ее понимания сути подхода, выраженном ранее и во введении к примеру. Все разработчики – субъекты, или попросту – люди. Не боги, но и не роботы. А каждый человек имеет свой личный подход к решению проблем, зависящий от теоретической квалификации, практического опыта и многих иных вещей – крайне полезных для понимания проблем, но не предопределяющих однозначно решения. Жизнь сложна – языки просты.
Болезни современного программирования, которую и пытается лечить UML – идут от обратного. Не разобравшись в проблемах, мы часто заранее уверены в однозначности решения. Каждый – своего собственного. Оттого-то зачастую температура «в среднем по больнице» поднимается выше нормальной 36.6 - Н.Б.

ПОСЛЕСЛОВИЕ ДЛЯ ХАКЕРОВ. ОСНОВНОЙ РЕСУРС.

Конечно, не все из сказанного выше было для тебя на 100% ясно и на 100% полезно - в смысле 100% готовности *завтра же* приступить к разработке сложных программных систем 100% надежности. Вроде так актуальных сегодня систем задач 100% защиты информации «от где-то спрятавшихся злобных хакеров». Я не верю, что стопроцентные гарантии здесь возможны - и нужны вообще. Во всяком случае, эта скромная методичка не ставила перед собой недостижимых целей. Это всего лишь *инструкция для еще не квалифицированного пользователя (клиента) языка UML*. Написанная, по мере возможности, «по-человечески».

Никто не совершенен – ни ты, «клиент образования», ни я, твой «сервер». Ты недоучился или я недоучил, теперь уже поздно искать виноватых. Здесь мы сделали лишь первую итерацию в процессе подготовки. Конечно, для достаточно надежного программирования сложных систем нужно еще углубляться, специализироваться, еще многое знать и уметь. Это ясно. Но данное пособие – о другом. Здесь я хотел сделать не шаг вперед, но *шаг назад* – показать тебе, что сначала нужно суметь заново понять *уже пройденное*. Зачастую – недопонятое, пройденное *мимо.*

Хакер – вовсе не злобный преступник. По крайней мере – изначально. Напротив, хакером называют и энтузиаста программирования. Тогда хакеры - это ты и я. Что ж, без интереса к работе – жизнь скучна. Мы же все-таки люди, не роботы. Все не идеальны, все индивидуальны.

Но все мы поначалу беремся сделать 1) все 2) сразу 3) идеально. И это нормально – пока не сильно касается других, близких и дальних. Момент истины наступает, когда наш идеальный сценарий заканчивается неудачей. Тогда одни начинают учиться работать всерьез, а те, что покруче - искать виноватых. «Недоучили, не так и не тому учили» – в общем, недодали ресурсов. Милый, а ты сам – сильно вкладывался? Не скрыты ли за твоей непоколебимой верой в безграничную мощь собственной интуиции надежда на авось и… обычная лень, нежелание трудиться?

Hack-work – это поденщина, рутинная и халтурная работа, а «крутой хакер» – человек крайне ненадежный. Бесплатно или за хорошие деньги, но если такой возьмется за разработку уже не игрушечных систем – реального вреда будет не меньше, чем от любого «злобного хакера» (как правило – виртуального). Расплачиваться же тогда будут все. Реально. И интересно тогда уже точно никому не будет.

Дело тут не в самих деньгах, договорах и иных артефактах. Это лишь обозначения общественного договора, знаки доверия и договоренностей между людьми. Просто на будущее вещи приходиться фиксировать – чтобы не забыть прошлого. Основной ресурс – ресурс человеческих взаимоотношений. Если не приумножать его, тогда действительно – кому нужна вся эта математика? It’s a deal -договорились?

Я надеюсь, теперь ты лучше понимаешь то, на что и кого именно я делал свой расчет. Вот такая математика…

СПИСОК ЛИТЕРАТУРЫ

1. [Booch,Rumbaugh,Jacobson] **Гради Буч, Джеймс Рамбо, Ивар Джекобсон – Язык UML. Руководство пользователя.** Издательство ДМК Пресс, 2007 г., 496 с. *Классика от создателей UML.*
2. [Larman] **Крэг Ларман – Применение UML 2.0 и шаблонов проектирования.** Издательство Вильямс, 2008 г., 736 с. *Отражает доминирующий сегодня инженерный подход к ОО АП.*
3. **Дж. Рамбо, М. Блаха - UML 2.О. Объектно-ориентированное моделирование и разработка.** 2-е изд. — СПб.: Питер, 2007. — 544 с. *То же, с чуть более практическим уклоном.*
4. **Rational University** – материалы академической программы корпорации IBM (см. http://www.ibm.com/ru/software/info/students/): Essentials of visual modeling, Fundamentals of Rational Rose. *Сокровищница примеров, тестов и лабораторных работ.*

Дополнительная литература.

1. **Мартин Фаулер. UML. Основы.** Издательство Символ-Плюс, 2006 г., 192 с. *Краткий справочник.*
2. **Rational University** – материалы академической программы корпорации IBM (см. http://www.ibm.com/ru/software/info/students/): Mastering Object-Oriented Analysis and Design, Managing the Management of Iterative Development и другие.
3. **Бертран Мейер. Объектно-ориентирование конструирование программных систем**. Издательство Русская редакция 2005 г., 1204 с. *Отражает более классический взгляд на современную ситуацию. Требует хорошей математической подготовки.*

Дополнительная литература - для преподавателей.

Ф.А. Новиков. Описание практической работы студентов (ЛП) по дисциплине «Анализ и проектирование на UML» - кафедра «Технологии программирования», Санкт-Петербургский государственный университет информационных технологий, механики и оптики, Санкт-Петербург, 2007

Выше автор ориентировался на *индивидуальную* подготовку. Но единственный надежный способ практической проверки понимания изложенных выше концепций ОО АП – особенно, «критерия 36.6» - коллективная, *командная разработка*. Методическая разработка Ф. Новикова дает здесь хороший старт - в виде схемы проведения соответствующих лабораторных работ. Хотя автор лично рекомендовал бы для их выполнения задания игрового характера, не претендующие на серьезность. Например - шахматы, шашки, иные настольные игры.