1. Особенности CASE-технологии. История развития CASE-средств.

Особенности CASE-технологии

На пути к достижению комплексного подхода при разработке

программных средств (ПС) широкое применение получили CASE-средства

(Computer Aided Software Engineering), обеспечивающие поддержку

многочисленных технологий проектирования информационных систем,

охватывая всевозможные средства автоматизации и весь жизненный цикл

программного обеспечения (ПО). Диапазон CASE-средств очень велик, и

сегодня практически каждое из них

располагает мощной инструментальной

базой.

CASE-технология включает в себя методологию анализа,

проектирования, разработки и сопровождения сложных систем ПО,

поддержанную комплексом взаимосвязанных средств автоматизации.

Основная цель CASE-подхода – разделить и максимально автоматизироватьù

все этапы разработки ПС. Большинство CASE-средств основано на

парадигме *методология / метод / нотация / средство*.

**Методология** определяет шаги работы и их последовательность, а также

правила распределения и назначения методов. **Метод** – это систематическая

процедура генерации описаний компонентов ПО. **Нотация** предназначена

для описания структур данных, порождающих систем и метасистем.

**Средства** – это инструментарий для поддержки методов на основе принятой

нотации.

Основные преимущества применения CASE-средств:

- улучшение качества ПО за счет автоматического контроля проекта;

- возможность быстрого создания прототипа будущей системы, что позволяет уже на ранних стадиях разработки оценить результат;

- ускорение процессов проектирования и программирования;

- освобождение разработчиков от выполнения рутинных операций;

- возможность повторного использования ранее созданных

компонентов.

История

Основной причиной, по которой возникла необходимость в появлении систем автоматизированного проектирования в области информационных технологий, явился дисбаланс между производительностями труда в сфере производства и в сфере обработки информации. Причем разница была не в пользу последней. Начиная с 60-ых годов ХХ столетия, через каждые десять лет указанный разрыв значительно увеличивался.

На первых порах проблему пытались решить путем перевода людей из одной области труда в другую, что привело к падению общих темпов роста производительности труда и экономики. Кроме этого, ситуацию осложняли следующие факторы:

* Число различных классов технических систем удваивалось в среднем через каждые десять лет.
* Сложность изделий, вызванная увеличением количества их комплектующих, удваивалась через десять лет.
* Объем научно-технической информации удваивался через каждые восемь лет.
* Период создания новых изделий уменьшался в два раза через десять лет, и при этом сокращалось время их морального старения.

В результате совокупного действия перечисленных факторов объем разработок в области проектирования должен был возрастать приблизительно в десять раз через каждые десять лет. В связи с этим были начаты работы по автоматизации процессов проектирования. До недавнего времени концепция автоматизации труда базировалась на принципах геометрического моделирования и компьютерной графики, что позволяло охватить лишь стадии технического и рабочего проектирования. Системы компьютеризации труда конструкторов, технологов, программистов и менеджеров развивались автономно. В современных условиях необходима комплексная компьютеризация инженерной деятельности на всех этапах ЖЦ программного продукта. Комплексный характер CASE-средств заключается в том, что в них выполняются и сложные вычисления, и обработка большого объема данных. Это определяет направление дальнейшего развития CASE-средств как интегрированных интеллектуальных систем.

Современные CASE-средства делят на категории: тяжелые, средние, легкие. Категории определяются средствами, вложенными в систему, или усилиями, потраченными на ее освоение. Считается, что успех во многом зависит от того, на какие информационные ресурсы она ориентируется: активные или пассивные.

Активные информационные ресурсы составляет информация, доступная для автоматизированного хранения, поиска и обеспечивающая обработку данных. Иные формы информации являются пассивными ресурсами. Система автоматизированного проектирования должна взаимодействовать только с активными ресурсами.

1. Системная модель CASE-средств.

Любая техническая система, включая CASE-средство, может быть представлена следующим набором характеристик:

*S={Ind, P, Atr, Inp, Out, Str},* где

*Ind* – обозначение и наименование системы;

*P* – цели системы;

*Atr* – общесистемные характеристики;

*Inp* – входы системы;

*Out* – выходы системы;

*Str* – структура системы;

*Str={E, R}*, где *E* – компоненты системы, *R* – связи между компонентами.

*Обозначение и наименование системы*. Каждая коммерческая система должна иметь зарегистрированный товарный знак, который в совокупности с обозначением версии и модификации системы представляет собой обозначение системы. Наименование включает в себя ее функциональное описание.

*Цели системы* достигаются за счет ее технических функций, которые характеризуют способность преобразовывать входную информацию в выходную. В качестве целей системы чаще всего выступают такие характеристики, как: *трудоемкость, себестоимость, длительность цикла процесса, качество продукта.*

Уменьшение трудоемкости проектирования достигается за счет:

- автоматизации оформления документации,

- информационной поддержки и автоматизации принятия решений,

- параллельного проектирования.

Снижение себестоимости разработки происходит благодаря разумной экономии всех ресурсов.

Сокращение ЖЦ достигается за счет параллельного проектирования и создания виртуальных бюро.

Улучшение качества результатов проектирования обеспечивается путем:

- использования автоматизированного поискового и многовариантного проектирования,

- применения математических методов оптимизации параметров и структур объектов и процессов,

- привлечения стратегического проектирования.

*Общесистемные характеристики* участвуют в классификации CASE-средств по следующим признакам:

- прикладная область объектов проектирования,

- сложность проектируемых объектов,

- уровень автоматизации (менее 25% от общего количества автоматизированных процедур – низкоавтоматизированные CASE; от 25% до 50% автоматизированных процедур – среднеавтоматизированные CASE; более 50% автоматизированных процедур – высокоавтоматизированные CASE),

- комплексность автоматизации проектирования (зависит от стадий проектирования, которые охватывает система, здесь выделяют: *одноэтапные, многоэтапные и комплексные*),

- возможность работы в сетевом режиме и в Internet.

*Входы и выходы* системы зависят от ее функционального назначения и описываются в техническом задании на разработку.

*Структура* системы включает в себя ее функциональные составные части и связи между ними, а также зависит от комплексности CASE-средств.

1. Критерии развития CASE-средств.

Каждая техническая система, в том числе и CASE-система, характеризуется группой свойств, которые определяют меру совершенства и прогрессивности данной системы. Такие свойства называют критериями развития. Наборы критериев многих технических систем совпадают. Это вызвано тем, что все они являются искусственными и, в большинстве человеко-машинными системами. CASE-средства принадлежат к классу систем информатики, где преобразуемым операндом является информация. В результате ряд критериев для CASE-средств не являются актуальными, связанные с затратами материалов, энергии, экологией. Критерии, важные для CASE-средств, можно разделить на четыре группы:

функциональные,

технологические,

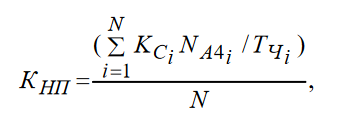
экономические,

эргономические.

Функциональный критерий рассматривается как интегральный показатель, зависящий от ряда частных функциональных критериев:

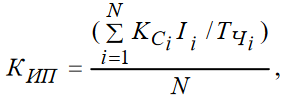
* *скорости обработки информации,*
* *интенсивности обработки информации,*
* *степени автоматизации труда,*
* *непрерывности процесса проектирования.*

Скорость обработки информации характеризуется двумя величинами: КНП – натуральным критерием производительности CASE-средства и КИП – информационным критерием производительности CASE-средства.



где КСi – коэффициент сложности работы i-ого проекта; NA4i – количество листов формата А4 i-ого проекта; ТЧ i – время в часах, затраченное на автоматический выпуск указанного количества листов i-ого проекта;N – количество проектов. КНП > 0, измеряется количеством листов, выпускаемых в часах, и возрастает с развитием CASE-средства. Если объем выполненной работы измеряется количеством информации в модели создаваемого объекта

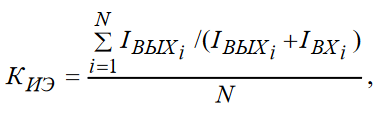
или процесса, то КНП можно заменить КИП .



где Ii – количество информации в принятых единицах измерения в i-ой модели. Независимо от критерия, которым измеряется, прежде всего производительность CASE-средства зависит от объема знаний и данных, заложенных в систему. С интенсивностью обработки информации в CASE-средства связан информационный критерий эффективности КИЭ. Он представляет собой

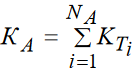
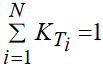
усредненное отношение объема выходной информации к суммарному

объему входной и выходной информации.

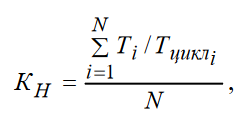


где IВЫХi – объем результирующих данных i-ого проекта, IВХi – объем исходных данн ых i-ого проекта, N – количество проектов. КИЭ принимает значения в диапазоне от 0 до 1 и является безразмерной величиной. Система считается тем совершеннее, чем меньше данных в нее нужно вводить. Предельным минимумом для этого является техническое задание.

Критерий совмещения функциональных операций имеет смысл для интерактивных CASE-средств и определяет допустимое число одновременно выполняемых на одной рабочей станции функциональных операций.

Критерий, характеризующий степень автоматизации CASE-средства – это критерий автоматизации , где NA – количество автоматизированных операций, а KTi – удельная трудоемкость i-ой проектной операции. Сумма удельных трудоемкостей всех операций равна 1, т.е.  где N – количество всех проектных операций. Следовательно, KA принимает значения в диапазоне от 0 до 1.

Критерий непрерывности процесса проектирования

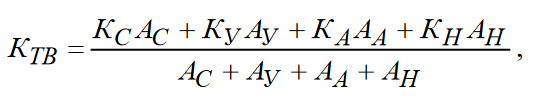


где Тi – чистая трудоемкость i-ого проекта в человеко-часах, а Тциклi – длительность календарного цикла i-ого проекта. Тциклi становится неоправданно большим из-за перехода с одной операции на другую, нехватки информации и т.д. С развитием CASE-средств стремятся к увеличению данного критерия.

Технологические критерии связаны с настоящими и будущими затратами на стадиях создания, развития и адаптации системы к решению конкретных задач.

Критерий трудоемкости разработки CASE-средства имеет две формы выражения  или , где ТC – суммарная трудоемкость разработки системы, КНП – натуральный показатель производительности, КИП – информационный показатель производительности. Данный критерий представляет собой монотонно убывающую величину, для уменьшения которой необходимо добиваться инвариантности ПС CASE-средств и использовать инструментальные средства автоматизации разработки программно-информационного обеспечения.

Критерий технологических возможностей отражает простоту разработки CASE-средства и подготовки ее к эксплуатации. Он является безразмерной величиной и определяется по формуле



где АС – количество стандартных программных модулей (ПМ),  – количество унифицированных ПМ, ÀÀ – количество адаптированных ПМ,АН – количество нестандартных ПМ, требующих разработки. К – весовые коэффициенты, причем КC =1 и  , Критерий технологических возможностей принимает значения в диапазоне от 0 до 1, и чем он больше, тем совершеннее CASE-средство.

В информатике принято различать старую и новую технологии. Для первой из них характерно доминирующее влияние нестандартных ПМ, когда что не позволяет добиться увеличения КТВ. Новая технология характеризуется тем, что  .Доминирующее влияние при создании CASE оказывают унифицированные и стандартные ПМ, а при подготовке к эксплуатации – адаптированные.

Экономический критерий CASE-средства служит для комплексного стоимостного учета положительного эффекта от автоматизации проектирования и основных затрат. В качестве этого показателя принято использовать КЭ – величину годового экономического эффекта от использования CASE-средства.



ДС – общее изменение себестоимости проектирования в расчетном году, связанное с производительностью CASE-средства;

ЭК – годовая экономия от повышения качества проектных решений, в основе которых лежит новая информационная база;

ДК – дополнительные капиталовложения (в вычислительную технику), связанные с созданием и внедрением CASE-средства;

КЕ – предпроизводственные затраты на CASE-средство, связанные с трудоемкостью ее разработки;

ЕН – нормативный коэффициент

Рассматриваемый критерий может принимать как положительные, так и отрицательные значения. Однако он имеет тенденцию к возрастанию.

Критерий эргономичности CASE-средства равен отношению реализуемой эффективности системы к максимально возможной эффективности этой системы. Он представляет собой зависящую от времени функцию, стремящуюся к 1. Данный критерий можно трактовать как КПД человека в системе.

1. Понятие проекта. Масштаб проекта. Общие принципы управления проектом.

Проект – это уникальный комплекс взаимосвязанных мероприятий, направленных на достижение конкретной цели при определенных требованиях к срокам, бюджету и характеристикам ожидаемых результатов.

В этом определении следует обратить внимание на следующее:

1) Каждый проект характеризуется конкретной целью, ради которой он

затевается.

2) Каждый проект в чем-то уникален.

3) Любой проект ограничен по времени “жизни”.

4) Каждый проект характеризуется конкретными ресурсами,

выделенными на его выполнение.

Еще одним понятием, актуальным для проекта, является его масштаб.

Масштаб проекта – это совокупность цели проекта и планируемых для ее достижения затрат времени и средств. Другими словами, это своеобразное трехмерное пространство (цель-время-деньги), в котором живут участники проекта и сам проект. Опираясь на введенное понятие масштаба, можно сказать, что управление проектом направлено на сохранение его исходного масштаба – содержания и границ.

Управление проектом – это процесс планирования, организации и контроля состояния задач и ресурсов проекта, направленный на своевременное достижение цели проекта.

В ходе управления любым проектам должно быть обеспечено решение следующих задач:

- соблюдение директивных сроков завершения проекта;

- рациональное распределение материальных ресурсов и исполнителей

между задачами проекта, а также во времени;

Своевременная коррекция исходного плана в соответствии с реальным положением дел. Чтобы проект оказался успешным, в его реализации должны быть предусмотрены три главные фазы:

1) Формирование плана.

2) Контроль реализации плана и управление проектом.

3) Завершение проекта.

Чем качественнее будут реализованы эти фазы, тем выше вероятность успешного выполнения проекта в целом.

Автоматизация проектирования должна помогать справляться со всеми аспектами сложности при разработке ПО. Поэтому независимо от прикладной области, к которой относится решаемая задача, и от принятого уровня абстракции необходимо, чтобы используемое CASE-средство поддерживала выполнение следующих функций:

1. Ввод описания проекта в систему.

2. Просмотр-обход – предоставление пользователям возможности выбора нужных фрагментов проекта и формирования необходимых запросов.

3. Декомпозиция системы – возможность представления проекта в виде удобных для обработки частей-модулей.

4. Контроль соблюдения правил и норм – проверка соответствия проекта всем требованиям с учетом количественных измерений.

1. Моделирование – обработка проекта на выбранном уровне абстракции с целью лучшего понимания его поведенческих характеристик.

6. Синтез (программирование) – преобразование проекта из одного вида представления в другой, как правило, с переходом на более низкий уровень абстракции.

7. Управление – возможность всем участникам процесса проектирования оставаться в рамках выбранной инженерной методологии.

Целенаправленное управление проектом предназначено для пропорционального распределения ресурсов между работами по созданию ПС на протяжении всего цикла проектирования вплоть до внедрения системы в серийное производство или ее массового использования. В общем случае при проектировании необходимо создать в соответствии с принятым критерием эффективности оптимальную систему управления или обработки информации.

1. Три составляющие программного проекта: система обозначений, процесс и инструмент. Их роль и значение для проекта.

Успех любого программного проекта зависит от трех составляющих: системы обозначений (нотации, языка), процесса и инструмента. Одинаково нужны все три составляющие.

Система обозначений важна в любой модели. Это связующее звено между всеми составляющими процесса разработки проекта. Примером полной и надежной системы обозначений может служить унифицированный язык моделирования (Unified Modeling Language – UML). С его помощью можно описывать модели на любом этапе разработки ПС: от анализа требований до проектирования и реализации.

Современные проекты должны разрабатываться с помощью эффективных средств в соответствии со строгим графиком и обеспечивать возможность внесения изменений и адаптации к конкретным условиям и требованиям. ЖЦ проекта должен быть управляемым, что позволит гарантировать его непременное завершение. Примером процесса, подходящего для современных IT-проектов, является унифицированный процесс. Его методология основана на языке UML и находит поддержку в современных инструментах.

Ни одна из современных технологий разработки ПС не обходится без

какого-либо инструмента. Сегодня на рынке представлен достаточно

широкий спектр инструментов, использующих нотацию UML и объектно-

ориентированный подход к разработке ПО.

1. Типы и особенности современных программных проектов.

**Проект для постоянного заказчика**. Ситуация, при которой команда разработчиков в течение длительного времени обслуживает единственного заказчика.

**Продукт под заказ**. Ситуация, при которой команда разработчиков находит стороннего заказчика и договаривается с ним о разработки программного продукта, призванного решить те или иные проблемы заказчика.

**Тиражируемый продукт**. Ситуация, при которой команда разработчиков либо вообще не имеет конкретных заказчиков, либо довольно большое количество заказчиков желают иметь один и тот же продукт.

**Аутсорсинг**. Это одна из наиболее новых моделей производства ПО. Суть ее состоит в том, что между крупной (обычно) фирмой по производству ПО и другой иностранной фирмой заключается договор о субподряде.

*Проект для постоянного заказчика.* Самый благоприятный тип проекта для внедрения легких методологий, поскольку заказчик всегда доступен и не предъявляет сверхтребований к ПО. Однако необходимо учитывать количество разработчиков и степень их распределенности. Как правило, у таких команд не бывает необходимости в сертификации.

*Продукт под заказ.* Самый уязвимый тип проекта. Фирма целиком зависит от количества заключенных договоров. Постоянно идет поиск новых заказчиков. В таких условиях, конечно же, желательно наличие сертификата ISO. Сертификацию целесообразно проводить лишь при достижении определенной численности персонала, которой будет достаточно для внедрения тяжелой или средней технологии. Альтернативный вариант – одна из легких методологий.

*Тиражируемый продукт.* Самый устойчивый тип проекта. Выпуск такого проекта всегда характеризуется более низкими затратами на его производство по сравнению с выпуском единичных экземпляров. В данных условиях невозможно использование легкой методологии в чистом виде, т.к. нет возможности постоянно работать с заказчиком. В этом случае все зависит от способа управления командой, тактических и стратегических целей.

*Аутсорсинг*. Данный вид проектов характеризуется распределенной структурой и начальными предпосылками к утяжелению процесса, поскольку общение с заказчиком происходит в виде документов установленного образца. Если сторона фирмы-заказчика предоставляет команде свой технологический процесс, то у нее нет свободы выбора. В противном случае стоит остановить свой выбор на каком-либо из процессов средней тяжести.

1. Задачи и категории современных методологий создания программных проектов.

Под методологией понимается набор методов, практик, метрик и правил, используемых в процессе производства ПО. Главными задачами современной методологии и основанного на ней процесса являются следующие:

– облегчить процедуру введения новых людей в курс процесса производства;

– обеспечить взаимозаменяемость людей;

– распределить ответственности;

– демонстрировать видимый прозрачный процесс;

– создать учебную базу для сотрудников.

Методологии можно условно разбить на три категории: *тяжелые, легкие и средние.* Упрощенно каждая из них предназначена для работы в условиях больших, малых и средних проектов соответственно.

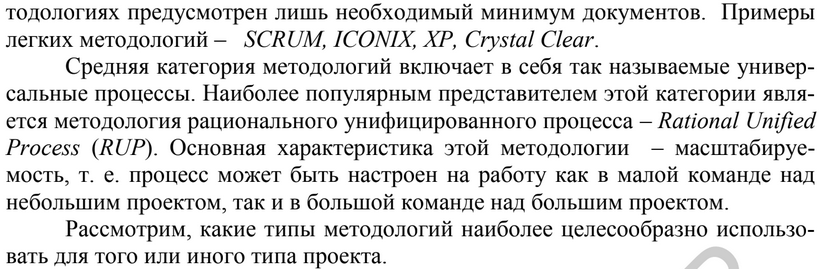
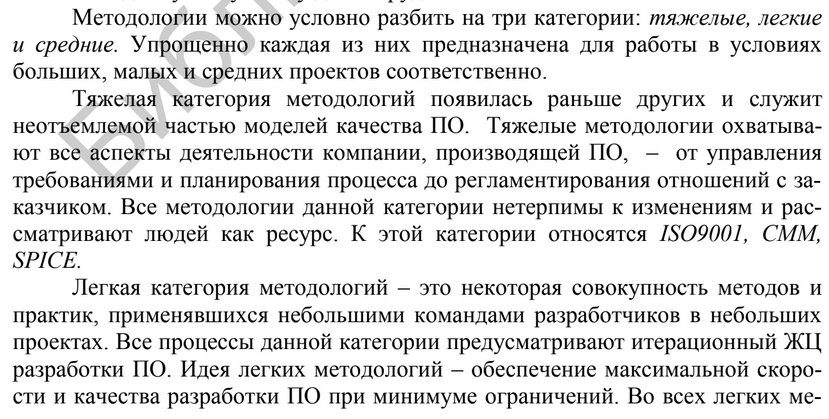
Тяжелая категория методологий появилась раньше других и служит неотъемлемой частью моделей качества ПО. Тяжелые методологии охватывают все аспекты деятельности компании, производящей ПО, – от управления требованиями и планирования процесса до регламентирования отношений с заказчиком. Все методологии данной категории нетерпимы к изменениям и рассматривают людей как ресурс. К этой категории относятся *ISO 9001, CMM, SPICE.*

Легкая категория методологий – это некоторая совокупность методов и практик, применявшихся небольшими командами разработчиков в небольших проектах. Все процессы данной категории предусматривают итерационный ЖЦ разработки ПО. Идея легких методологий – обеспечение максимальной скорости и качества разработки ПО при минимуме ограничений. Во всех легких методологиях предусмотрен лишь необходимый минимум документов. Примеры легких методологий - *SCRUM, ICONIX, XP, Crystal Clear*.

Средняя категория методологий включает в себя так называемые универсальные процессы. Наиболее популярным представителем этой категории является методология рационального унифицированного процесса – *Rational Unified Process (RUP*). Основная характеристика этой методологии – масштабируемость, т.е. процесс может быть настроен на работу как в малой команде над небольшим проектом, так и в большой команде над большим проектом.

Рассмотрим, какие типы методологий наиболее целесообразно использовать для того или иного типа проекта. (вопр. 6)

1. Взаимосвязь между методологией, размером задачи и командой разработчиков.



1. Целесообразность использования различных методологий для различных типов программных проектов.

Рассмотрим, какие типы методологий наиболее целесообразно использовать для того или иного типа проекта.

*Проект для постоянного заказчика.* Самый благоприятный тип проекта для внедрения легких методологий, поскольку заказчик всегда доступен и не предъявляет сверхтребований к ПО. Однако необходимо учитывать количество разработчиков и степень их распределенности. Как правило, у таких команд не бывает необходимости в сертификации.

*Продукт под заказ.* Самый уязвимый тип проекта. Фирма целиком зависит от количества заключенных договоров. Постоянно идет поиск новых заказчиков. В таких условиях, конечно же, желательно наличие сертификата ISO. Сертификацию целесообразно проводить лишь при достижении определенной численности персонала, которой будет достаточно для внедрения тяжелой или средней технологии. Альтернативный вариант – одна из легких методологий.

*Тиражируемый продукт.* Самый устойчивый тип проекта. Выпуск такого проекта всегда характеризуется более низкими затратами на его производство по сравнению с выпуском единичных экземпляров. В данных условиях невозможно использование легкой методологии в чистом виде, т.к. нет возможности постоянно работать с заказчиком. В этом случае все зависит от способа управления командой, тактических и стратегических целей.

*Аутсорсинг*. Данный вид проектов характеризуется распределенной структурой и начальными предпосылками к утяжелению процесса, поскольку общение с заказчиком происходит в виде документов установленного образца. Если сторона фирмы-заказчика предоставляет команде свой технологический процесс, то у нее нет свободы выбора. В противном случае стоит остановить свой выбор на каком-либо из процессов средней тяжести.

1. Унифицированный процесс разработки программных средств.

Прежде всего унифицированный процесс – это процесс разработки ПО, т.е. это множество различных видов деятельности, необходимых для преобразования требований пользователей в программную систему.

Однако унифицированный процесс – это больше, чем единичный процесс. *UP* – это обобщенный каркас процесса, который может быть специализирован для широкого круга программных систем, различных областей применений, уровней компетенции и разм

еров проекта. Его основными принципами являются:

– итерационный и инкрементный подход к созданию ПО;

– управление вариантами использования;

– построение системы на базе архитектуры ПО.

Первый принцип является определяющим. В соответствии с ним разработка системы выполняется в виде нескольких краткосрочных мини-проектов фиксированной длительности (от 2 до 6 недель), называемых итерациями. Каждая итерация включает в себя свои собственные этапы анализа требований, проектирования, реализации, тестирования, интеграции и завершается созданием работающей системы.

Итерационный цикл основывается на постоянном расширении и дополнении системы в процессе нескольких итераций с периодической обратной связью и адаптацией добавляемых модулей к существующему ядру системы. Система постоянно разрастается, поэтому такой подход называют итерационным и инкрементным.

Вариант использования – это часть функциональности системы, необходимая для получения пользователем значимого для него, ощутимого и измеримого, результата. Варианты использования обеспечивают функциональные требования. Все варианты использования в совокупности составляют модель вариантов использования, которая описывает полную функциональность системы.

Варианты использования в *UP* – это не только средство описания требований к системе. Они направляют далее весь процесс ее разработки. Основываясь на модели вариантов использования, разработчики создают все последующие модели.

Поскольку варианты использования управляют процессом, они не выделяются изолированно, а разрабатываются совместно с созданием архитектуры системы. Следовательно, варианты использования управляют архитектурой системы, которая, в свою очередь, оказывает влияние на их выбор. И архитектура системы, и варианты использования развиваются по мере хода жизненного цикла.

Созданная архитектура является основой всей дальнейшей разработки. В будущем неизбежны незначительные изменения в деталях архитектуры, однако серьезные изменения маловероятны.

1. Основные и дополнительные элементы объектно-ориентированного подхода.

Главное отличие объектного подхода от структурного заключается в объектной декомпозиции системы. При этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы – в терминах обмена сообщениями между объектами. Концептуальной основой объектного подхода является объектная модель, главными элементами которой считаются: абстрагирование, инкапсуляция, модульность, иерархия. Кроме того, имеются три дополнительных элемента: типизация, параллелизм, устойчивость. Рассмотрим все элементы с точки зрения создания ПО с помощью CASE-средств.

*Абстрагирование* – это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование позволяет отделить существенные особенности поведения объекта от деталей их реализации. Выбор правильного набора абстракций – это главная задача объектно-ориентированного подхода.

*Инкапсуляция* – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы разделить интерфейс и внутреннюю реализацию объекта.

*Модульность* – это свойство системы, связанное с ее декомпозицией на ряд внутренне сильно связанных, но слабо связанных между собой модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

*Иерархия* – это ранжированная или упорядоченная система абстракций. Иерархия по номенклатуре – это структура классов, а иерархия по составу – это структура объектов.

*Типизация* – это ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов.

*Параллелизм* – это свойство объектов находиться в активном или пассивном состоянии и различать между собой активные и пассивные объекты.

*Устойчивость* – это свойство объекта существовать во времени и в пространстве вне зависимости от процесса, породившего данный объект.

Еще два важных понятия объектно-ориентированного подхода – это полиморфизм и наследование. *Полиморфизм* можно интерпретировать, как способность класса принадлежать более чем одному типу. *Наследование* означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов. Система изначально строится с учетом ее эволюции, которую позволяют реализовать полиморфизм и наследование: потомки могут добавлять в родительские классы новые структуры данных и методы.

Благодаря применению абстрагирования, модульности и полиморфизма на всех стадиях разработки ПС существует согласованность между моделями всех этапов ЖЦ, когда модели ранних стадий могут быть сравнены с моделями реализации.

1. История появления, особенности и назначение унифицированного языка моделирования UML.

Методы объектно-ориентированного анализа и проектирования включают в себя язык моделирования и описание процессов моделирования. Язык моделирования UML (Unified Modeling Language) – это нотация, которая используется методом для описания проектов. *Нотация*  представляет собой совокупность графических объектов, которые используются в моделях. Она является синтаксисом языка моделирования. *Процесс* – это описание шагов, которые необходимо выполнить при разработке проекта.

Авторами UML являются основоположники объектно-ориентированного подхода: Буч, Рамбо, Якобсон. UML объединяет в себе методы объектного подхода, дополняя их новыми возможностями. UML не привязан к какой-либо конкретной методологии или ЖЦ и может использоваться со всеми существующими методологиями.

Создание UML началось в 1994 году, а в 1995 появилась первая спецификация языка. В 2000 году появилась версия UML 1.4 как существенное расширение UML, достигнутое добавлением семантики действий. Это было серьезным достижением, поскольку сделало спецификацию UML полной в вычислительном отношении, что обеспечило возможность создавать UML-модели исполняемыми. В 2005 году была завершена спецификация UML 2.0.

Основная идея UML – возможность моделировать ПО и другие системы как наборы взаимодействующих объектов. В UML-модели есть два аспекта:

– *статическая структура* – описывает, какие типы объектов важны для моделирования системы и как они взаимосвязаны;

– *динамическое поведение* – описывает ЖЦ этих объектов и то, как они взаимодействуют друг с другом для обеспечения требуемой функциональности системы.

Основные цели создания унифицированного языка моделирования:

1. Предоставить пользователям готовый к применению выразительный язык визуального моделирования, позволяющий разрабатывать осмысленные модели и обмениваться ими.

2. Предусмотреть механизмы для расширения базовых концепций.

3. Обеспечить независимость UML от конкретных языков программирования и процессов разработки.

4. Создать формальную основу для понимания языка моделирования.

5. Стимулировать рост рынка объектно-ориентированных инструментальных средств.

6. Интегрировать лучший практический опыт.

Семантика языка UMLпредставляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML. Семантика определяется для двух видов объектных моделей: структурных моделей и моделей поведения. Структурные модели, известные также как статические модели, описывают структуру сущностей или компонентов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения. Модели поведения, называемые иногда динамическими моделями, описывают поведение или функционирование объектов системы, включая их методы, взаимодействие и сотрудничество между ними, а также процесс изменения состояний отдельных компонентов и системы в целом.

Для решения столь широкого диапазона задач моделирования разработана достаточно полная семантика для всех компонентов графической нотации. Требования семантики языка UML конкретизируются при построении отдельных видов диаграмм.

В настоящее время UML принят в качестве стандартного языка моделирования и получил широкую поддержку в индустрии ПО. UML взят на вооружение самыми известными производителями ПО: IBM, Microsoft, Hewlett-Packard, Oracle. Большинство современных CASE-средств разрабатывается на основе UML.

1. Требования к программному обеспечению. Первичные и детальные требования. Функциональные и нефункциональные требования.

**Требования к программному обеспечению** — совокупность утверждений относительно атрибутов, свойств или качеств [программной системы](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), подлежащей реализации. Создаются в процессе разработки требований к [программному обеспечению](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) (ПО), в результате [анализа требований](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7_%D1%82%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B9).

(из Википедии)

Делятся на первичные и детальные.

Первичные - документируют потребности заказчика, на понятном заказчику языке. Требования заказчика помещаются в системную спецификацию.

Детальные - требования оформлены и структурированы, более детализированы по отношению к первичным. Требования разработчика содержат детальные описания функций и ограничений системы - это оформляется в спецификацию анализа.

По характеру требования делятся на функциональные и нефункциональные.

Функциональные описывают поведение системы и функции, которые она должна выполнять. Исходят из анализа предметной области. Разнообразные варианты поведения при разных входных данных и внешней среде.

Нефункциональные описывают требования к характеристикам системы и внешнего окружения. Могут накладывать ограничения на функции системы или условия разработки.

Нефункциональные делятся на

Требования к программной системе - свойства и характеристики системы

(скорость работы, производительность, потребление памяти, надежность, переносимость и т.п.)

Организационные - о взаимодействии разработчика и заказчика (стандарты, методы и средства разработки; сроки; документация)

Внешние - относятся к факторам внешней среды и ее взаимодействия с системой (юридические обязательства, этические требования)

1. Назначение, особенности и построение диаграммы *Use Case*.

Диаграмма *Use Case* позволяет создать список операций, которые выполняет система. Часто *Use Case* называют диаграммой функций, так как на основе набора таких диаграмм создается список требований к системе и определяется множество выполняемых ею функций. Данный тип диаграмм используется при описании бизнес-процессов автоматизируемой предметной области, определении требований к будущей программной системе.

Моделирование системы начинается с анализа требований к ней, что напрямую связано с определением функций создаваемой системы. Поэтому часто диаграмму Use Case называют диаграммой функций. Основными инструментами этой диаграммы являются Use Case (варианты использования) и Actor (действующие лица, актеры). Вместе они определяют сферу применения создаваемой системы. Причем первые описывают все то, что происходит внутри системы, а вторые – то, что происходит снаружи. На диаграмме Use Case и Actor объединяются при помощи соответствующих связей. Еще одно название Use Case – прецедент, а актеров могут называть артефактами.

Несмотря на общие подходы к разработке диаграмм, в каждой CASE-среде имеются свои отличительные особенности.

Например, в Rational Rose предусмотрены дополнительные значки, предназначенные для построения модели производства. В целом они являются производными инструментов Use Case и Actor.

В Rational XDE не включены специальные значки для бизнес-анализа, что отражает сокращенный процесс разработки .NET-приложений, однако это не мешает полноценно использовать диаграмму Use Case для определения требований к системе.

В Enterprise Architect предусмотрены паттерны, позволяющие быстро спроектировать диаграмму функций конкретной системы.

<http://fkn.ktu10.com/?q=node/2236>

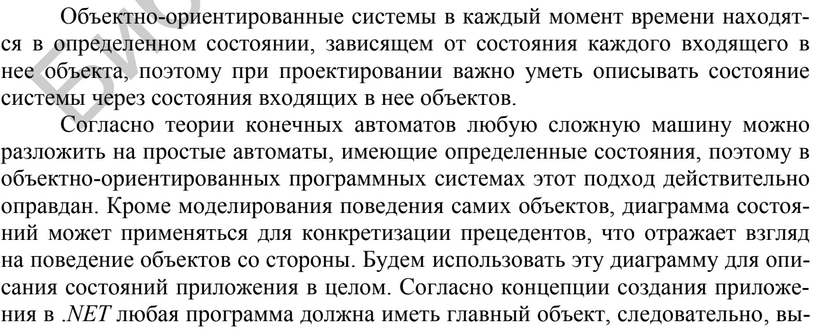
1. Назначение, особенности и построение диаграммы *Deployment*.

Диаграмма Deployment предназначена для анализа аппаратной части системы. В распределенных системах, какими часто бывают Web-приложения, этот тип диаграмм очень важен, поскольку именно здесь определяется, на каком сервере сети будет работать конкретный компонент или Web-служба и с какими другими сетевыми устройствами будет осуществляться взаимодействие.

При помощи *Deployment* проектировщик может провести анализ необходимой аппаратной конфигурации, на которой будут работать процессы системы, и описать их взаимодействие между собой. Этот тип диаграмм также позволяет анализировать взаимодействие процессов, работающих на разных компьютерах сети.

Современные CASE-инструменты позволяют наглядно показать топологию сети, поскольку в них включены стереотипы большинства распространенных сетевых устройств.

1. Назначение, особенности и построение диаграммы *Statechart*.

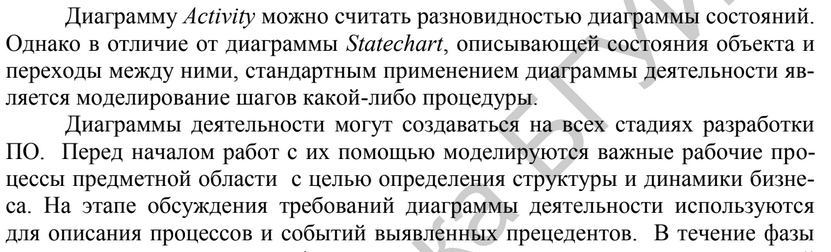


Одна из диаграмм состояний, показывает статичные состояния объекта и переходы между ними (активити показывает *действия*).

Диаграмма состояний *Statechart* предназначена для изучения состояний объектов и условий переходов между ними. Модель состояний позволяет представить поведение объекта при получении им сообщений и взаимодействии с другими объектами. Кроме ранее предложенных способов создания диаграмм, *Statechart* можно построить из контекстного меню рассматриваемого класса. После активизации диаграммы, станет доступным набор ее инструментов.

Для повышения информативности состояниям и событиям, переводящим объекты из одних состояний в другие, присваиваются имена. С состоянием объекта могут быть связаны события и действия. Разница между ними заключается в том, что действие осуществляется самим классом, для которого строится диаграмма *Statechart*, то есть вызывается метод данного класса, а посылка сообщения направлена на объект другого класса, чей метод вызывается при помощи сообщения. Для того чтобы получить доступ к действиям и событиям, нужно через контекстное меню объекта перейти в окно его спецификации, где выбрать вкладку *Actions*, а в ней – кнопку *Insert*. На рис. 6.1 состояние «Создан кладовщиком при получении товара» связано с действием «Создан в двух экземплярах». Построение диаграммы завершается добавлением на нее значка – End State, который отражает окончание работы. Направление перехода может быть установлено только в End State. Однако нет ограничений на количество самих элементов и переходов в них.

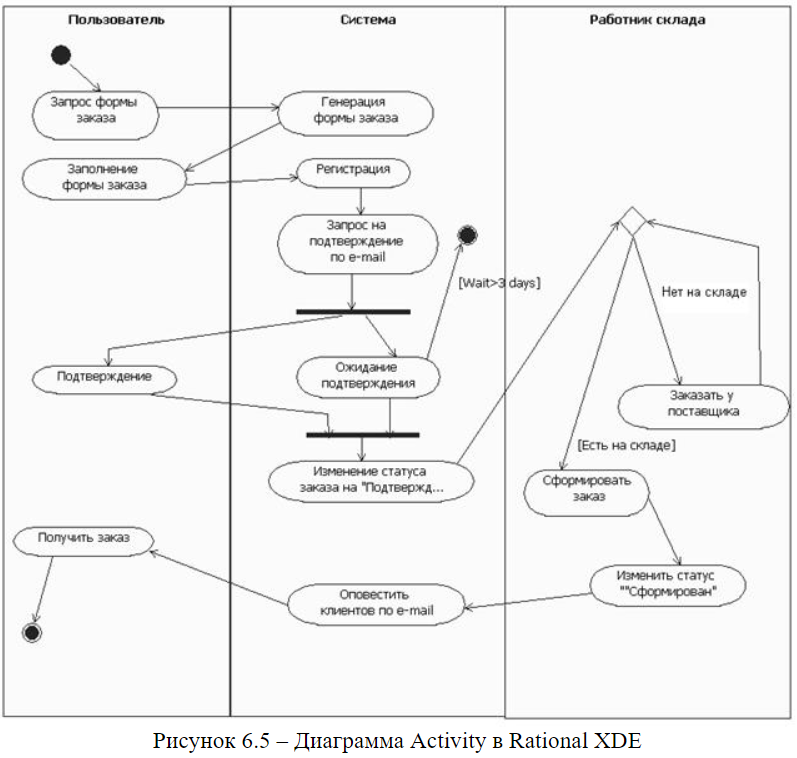
1. Назначение, особенности и построение диаграммы *Activity*.



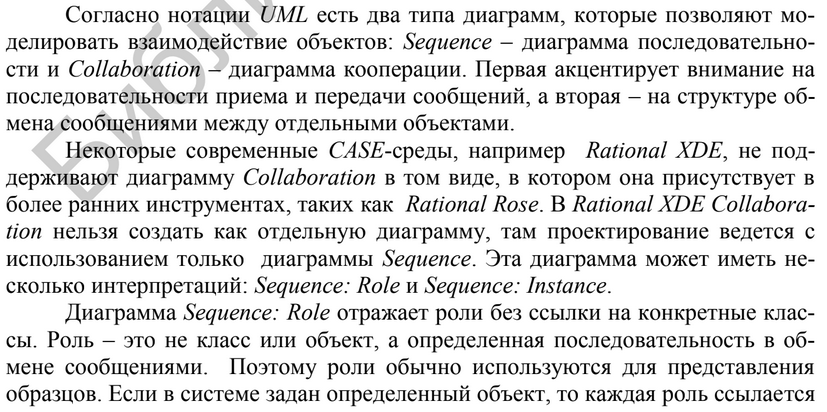
Для построения *Activity* доступны те же способы, что и для *Statechart*. При создании новой диаграммы состояний будет предложено выбрать из двух возможных вариантов: *Statechart Diagram* и *Activity Diagram*. После активизации *Activity* станет доступным набор ее инструментов. Как и *Statechart diagram*, *Activity* начинается значком *Start State* и завершается значком *End State*. Одноименный с диаграммой значок обозначает выполнение определенных действий в течение жизни объекта. В отличие от , обычно обозначающего ожидание какого-либо события, показывает непосредственное действие. Деятельности соединяются на диаграмме значком – *State Transition* (переход состояния). Кроме этого, показывает получение и обработку сообщения объектом. Переход состояния может происходить между *Action*-*Action*, *State*-*State*, *State*-*Action*, *Action*-*State*. Возможна установка нескольких переходов между двумя состояниями или действиями. Каждый такой переход уникален и показывает реакцию объекта на определенное сообщение. Поэтому нельзя создать несколько переходов между двумя состояниями с указанием одного и того же сообщения.

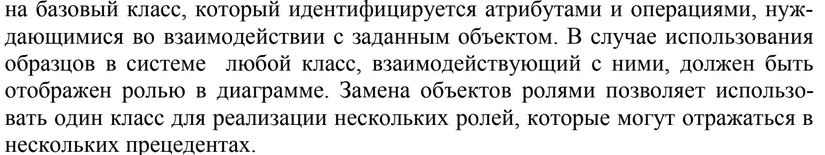
Уникальным инструментом диаграммы *Activity* является значок – *Swimlanes* (плавательные дорожки), который позволяет моделировать последовательность действий различных объектов и связи между ними. С его помощью можно строить бизнес-процессы организации, отражая на диаграмме различные подразделения и объекты. *Swimlanes* помогают показать роли каждого участника бизнес-процесса. Для этого необходимо переместить соответствующие значки активности или состояний в определенную часть диаграммы, отделенную от остальных *Swimlanes*.

Для построения диаграмм *Activity* имеется еще несколько важных инструментов. Значок - *Decision* (решение), позволяет показать зависимость дальнейшей работы от внешних условий и решений. Этот инструмент аналогичен командам языка программирования if или case и может иметь больше двух выходов, но обычно используют выбор из двух переходов, определенных булевыми выражением. Значки и – *Synchronizations* (синхронизация) позволяют определить независимо выполняемые действия. При этом действия разделяются на несколько выполняемых независимо, и только по завершении всех действий объект продолжает работу.



1. Назначение, особенности и построение диаграммы *Sequence*.





sequence - последовательность

collaboration - структура (наличие взаимосвязей между клиентами и серверами вообще)

Диаграмма взаимодействия создается одним из способов, которые были предложены в предыдущих разделах. Выбор какого-либо из них повлечет за собой открытие окна, в котором предложено два типа диаграмм: *Sequence* и *Collaboration*. После активизации диаграммы *Sequence* станет доступным набор ее инструментов.

Рассмотрим построение диаграммы Sequence на примере сценария работы кладовщика с карточкой товара и накладной (рис. 7.1), в нем использованы основные инструменты Sequence. Значок – *Object* (объект) позволяет включить объект в диаграмму. Каждый из них является реализацией какого-нибудь класса, поэтому в объекте можно указать соответствующий ему класс. Для повышения наглядности в примере использованы стереотипы классов. Кладовщик представлен как business worker, Накладная – как business entity, Карточка товара – как *business entity*. Значок – *Message* (сообщение) предназначен для передачи сообщения от одного объекта к другому. Классы должны позволять отправку или прием сообщений. Инструмент – *Message to self* (сообщение самому себе) показывает, что отправитель сообщения является одновременно и его получателем. В этом случае объект выполняет функции и сервера, и клиента. Для реализации объектов в окне параметров доступна опция *Persistence*, отражающая время жизни объекта. Это время от его создания до уничтожения. Обычно объекты существуют в пределах их области видимости и автоматически уничтожаются системой, когда выходят за эту область. Таким образом, можно настроить следующие параметры жизни объекта:

·*Persistent* – область видимости объекта превышает время жизни (сервер поставщика).

·*Static* – элемент существует на всем протяжении работы программы.

·*Transient* – время жизни объекта и области видимости совпадают, этот вариант присваивается по умолчанию и подходит в случае нашего примера.

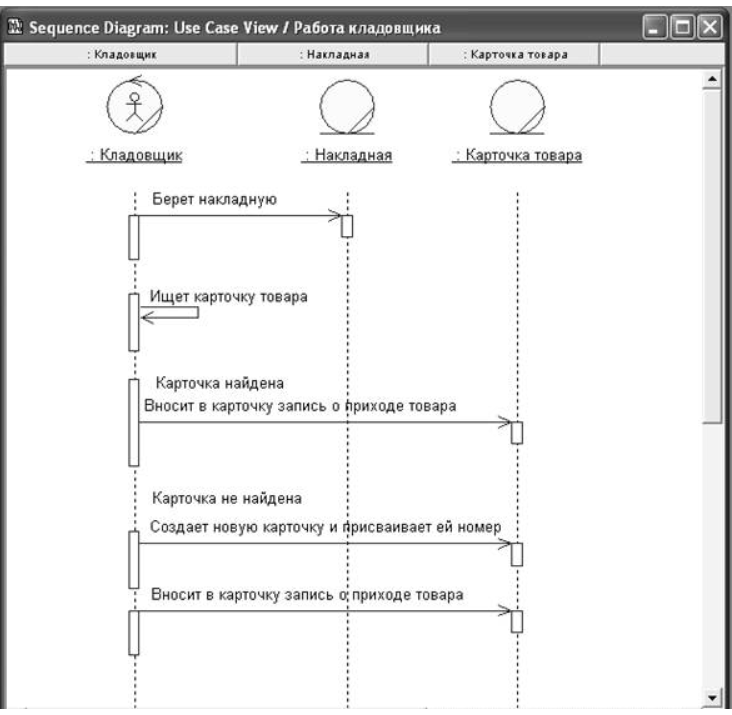


Рисунок 7.1 – Пример работы кладовщика с карточкой товара и накладной

Горизонтальные линии на рис. 7.1, проведенные от объекта к объекту, означают передачу сообщений между ними. Свойства задаются в окне спецификации с помощью двух групп кнопок.

1. *Synchronization* определяет порядок обмена сообщениями и может быть выбрана из следующих вариантов:

·*Simple* – простая посылка сообщения.

·*Synchronous* – операция происходит только в том случае, когда клиент посылает сообщение, а сервер может принять сообщение клиента.

·*Timeout* – клиент отказывается от выдачи сообщения, если сервер в течение определенного времени не может его принять.

·*Balking* – операция происходит только в том случае, когда сервер готов немедленно принять сообщение, иначе клиент не выдает сообщение.

·*Procedure Call* – клиент вызывает процедуру сервера и полностью передает ему управление.

·*Return* – возврат из процедуры.

·*Asynchronous* – клиент выдает сообщение, и, не ожидая ответа сервера, продолжает выполнение своего программного кода.

2. *Frequency* определяет частоту обмена сообщениями:

·*Periodic* – сообщения поступают от клиента с заданной периодичностью.

·*Aperiodic* – сообщения поступают от клиента нерегулярно.

**rational xde**

Диаграмма *Sequence: Role* отражает роли без ссылки на конкретные классы. Роль – это не класс или объект, а определенная последовательность в обмене сообщениями. Поэтому роли обычно используются для представления образцов. Если в системе задан определенный объект, то каждая роль ссылается на базовый класс, который идентифицируется атрибутами и операциями, нуждающимися во взаимодействии с заданным объектом. В случае использования образцов в системе любой класс, взаимодействующий с ними, должен быть отображен ролью в диаграмме. Замена объектов ролями позволяет использовать один класс для реализации нескольких ролей, которые могут отражаться в нескольких прецедентах. Имя роли отображается с косой чертой; в имени роли через двоеточие может быть представлено имя класса, реализующего эту роль.

На рис. 7.4 приведен *Toolbox* диаграммы *Sequence*.

Значок *Lifeline Actor* позволяет создать на диаграмме роль актера с указанием линии жизни. Фактически на диаграмме создается отображение актера, а линия жизни дополняет это отображение.

Однако не всегда необходимо создавать новых актеров в диаграмме. В разрабатываемой модели уже созданы актеры *Пользователь*, *Покупатель* и *Администратор*, поэтому при моделировании реализации прецедента *Регистрация* можно воспользоваться уже созданными элементами.

Инструмент *LifeLine* позволяет создать на диаграмме роль для программного объекта с линией жизни. В отличие от *LifeLine Actor* при создании этого элемента класс не добавляется, а его место в названии остается пустым. Абстрактную роль *Система* будет иллюстрировать в модели взаимодействия незарегистрированного пользователя с виртуальным магазином.

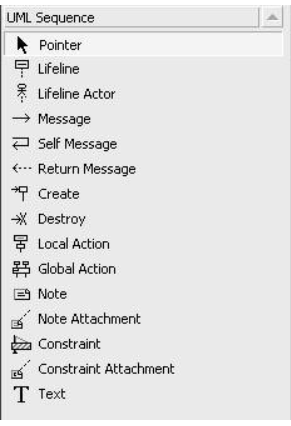


Рисунок 7.4 – Инструменты диаграммы *Sequence*

В диаграмме *Sequence: Instance* линия жизни может явно указать для объекта момент его создания и уничтожения.

Для отражения обмена сообщениями между незарегистрированным пользователем и окном регистрации создаются роли *Незарегистрированный пользователь* и *Окно регистрации*.

Инструмент *Message* позволяет создать отображение сообщения, передаваемого от одного объекта или роли к другому. Передача сообщения означает передачу управления объекту-получателю.

Значок *Return Message* показывает на диаграмме возврат управления из вызванной подпрограммы на сервере клиенту. На диаграмме такое сообщение отправляется от *Пользователя* к *Окну регистрации*.

Значок *Create* предназначен для показа сообщения, при помощи которого один объект создает другой. При этом созданный объект не получает управления, т.е. фокус активности остается у отправителя сообщения.

Значок *Destroy* отражает момент уничтожения программного объекта.

Окончательный вариант диаграммы Sequence показан на рис. 7.5.

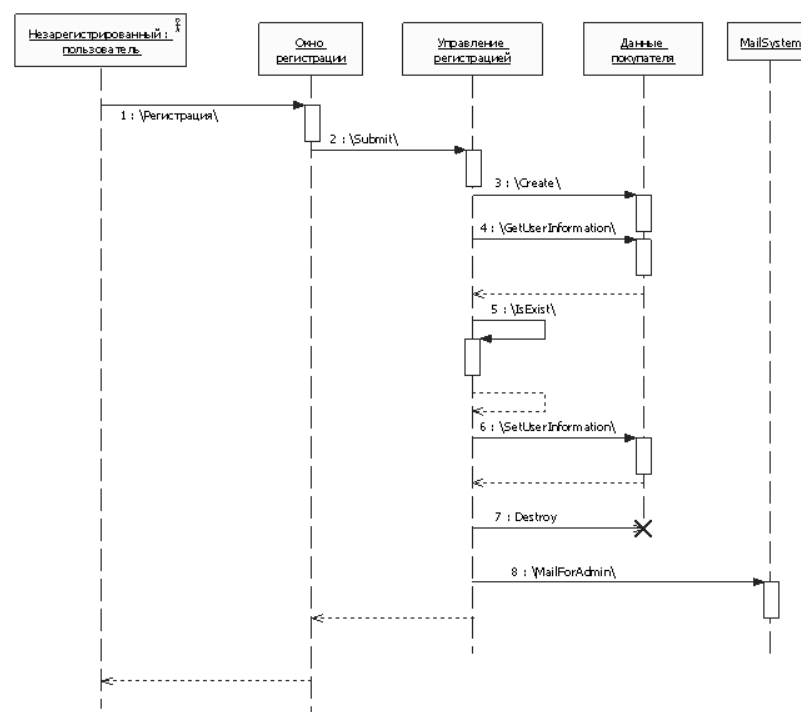


Рисунок 7.5 – Диаграмма Sequence в Rational XDE

Здесь происходят следующие события. *Незарегистрированный пользователь* хочет зарегистрироваться и нажимает кнопку в *Окне регистрации*. Окно регистрации выдает сообщение Submit для объекта управляющего регистрацией, который, в свою очередь, создает объект доступа к данным, выдавая сообщение Create, а затем запрашивает у созданного объекта данные пользователя для проверки, не был ли зарегистрирован такой пользователь ранее. Это происходит внутри обработчика сообщения *IsExist*, которое объект выдает самому себе. После получения данных и удостоверения того, что записи об этом пользователе еще нет в системе, отправляется сообщение о сохранении данных о пользователе *SetUserInformation*, после чего объект уничтожается.

Сообщение *MailFormAdmin* дает команду на отправку оповещения администратору о том, что в системе зарегистрировался новый пользователь, причем обработка этого сообщения происходит асинхронно, т.е. без ожидания завершения почтовой системы, поскольку отправка почты может занимать довольно длительное время.

1. Назначение, особенности и построение диаграммы *Collaboration*.

Второй тип диаграмм взаимодействия – это *Collaboration*. Данная диаграмма отличается от *Sequence* тем, что здесь не акцентируется внимание на последовательности передачи сообщений, а отражается наличие взаимосвязей между клиентами и серверами вообще. Поскольку на Collaboration для демонстрации сообщений не применяется временная шкала, диаграмма получается более компактной и оптимально подходит для представления взаимодействий сразу всех объектов. Однако такое представление является мгновенным снимком системы в некотором состоянии, так как объекты создаются и уничтожаются на всем протяжении работы программы. В связи с этим появляются такие понятия, как время жизни и область видимости объектов.

А вот как это строить хз что корректно вставлять, зачем учить как пользоваться программой…школьная инфа с вордом и поверпоинтом

Удобной возможностью работы в *Rational Rose* является то, что на основе Sequence-диаграммы можно создавать *Collaboration* и наоборот. Для построения *Collaboration* нужно, находясь в диаграмме *Sequence*, выбрать *Menu:Browse =>Create Collaboration diagram*. При построении с помощью значка Interaction diagram  и выбора Collaboration в диалоговом окне создается пустая диаграмма, и в нее не перенесутся уже существующие объекты. На рис. 7.2 представлена диаграмма *Collaboration*, построенная на основе диаграммы *Sequence* (см. рис. 7.1).

Для построения приведенной диаграммы *Collaboration* были использованы ее основные инструменты. Значок *Object* позволяет создавать объекты, которые имеют состояния и поведение. Значок  – *Object Link* (связь объекта) отражает взаимодействие объектов посредством показа их связей. При этом один из объектов может посылать сообщение другому. Значок – *Link To Self* показывает, что объект имеет обратную связь с самим собой. Значок – *Link Message* (передача сообщения) позволяет отразить связь, которая подразумевает обязательную передачу сообщения. У каждой связи *Link Message* есть соответствующие свойства, которые позволяют настроить область видимости для связанных объектов. Для задания области видимости объектов используется диалоговое окно, приведенное на рис. 7.3. Сервер описывается в блоке *Supplier visibility*, клиент – в блоке *Client visibility*. В них доступны значения для выбора:

· Unspecified – не определено, это значение присваивается по умолчанию.

· Field – объект включен в другой объект.

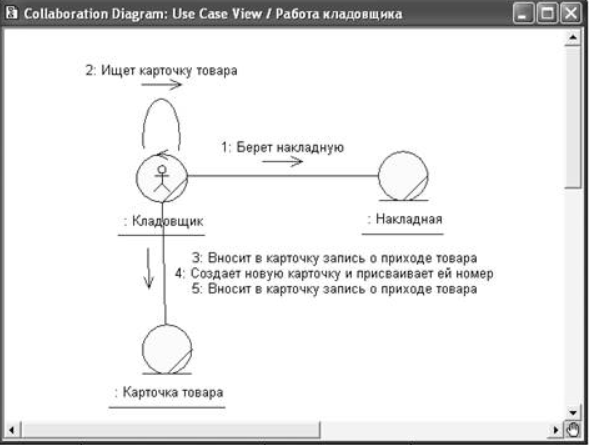


Рисунок 7.2 – Пример работы кладовщика с карточкой товара и накладной

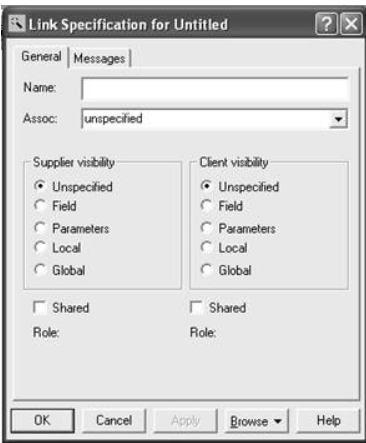


Рисунок 7.3 – Задание области видимости для Link Message

· *Parameter* – объект передается параметром в другой объект.

· *Local* – объект локально определен в границах другого объекта.

· *Global* – объект глобален по отношению к другому объекту.

*Rational XDE* не поддерживает диаграмму *Collaboration* в том виде, в котором она присутствует в среде Rational Rose. *Collaboration* нельзя создать как отдельную диаграмму. Проектирование ведется с использованием только диаграммы *Sequence*.

1. Назначение, особенности и построение диаграммы *Component*.



Диаграмма компонентов позволяет создать физическое отражение текущей модели. *Component* показывает организацию и взаимосвязи программных компонентов, представленных в файлах различных типов, а ее связи отражают зависимости одного компонента от другого. В текущей модели может быть создано несколько диаграмм компонентов для отражения пакетов, компонентов верхнего уровня или описания содержимого каждого пакета компонентов.

Для систем, состоящих из большого количества классов, целесообразно строить диаграмму компонентов, когда определены все связи классов и структура наследования. Но поскольку на всем протяжении проектирования системы, вплоть до выхода готового программного продукта в диаграммы будут вноситься изменения, оправдано создание *Component* для нескольких классов, чтобы получить практику работы с данным типом диаграмм.

В *Rational Rose* заложена возможность работы с программными библиотеками, их можно как создавать, так и пользоваться уже готовыми. Необходимо только указать, какие классы, в каких компонентах будут находиться. Для того чтобы обеспечить минимальные трудозатраты на разработку и сопровождение, тесно связанные между собой классы собираются в библиотеки.

Диаграмму компонентов можно построить двумя способами: с помощью меню *Browse=>Component diagram* или воспользовавшись значком *Component* *diagram*  на панели инструментов. После чего будет активизировано диалоговое окно выбора диаграммы, посредством которого создается, удаляется, переименовывается диаграмма.

Рассмотрим построения Component diagram на примере системы обслуживания банкоматов архитектуры клиент-сервер (рис. 8.1), на диаграмме показаны компоненты клиента. Система реализуется на языке программирования Visual С++. У каждого класса имеется свой собственный заголовочный файл и файл с расширением .срр, поэтому каждый класс преобразуется в свои собственные компоненты на диаграмме, представляющие тело и заголовок класса. Темные

компоненты соответствуют файлам тела класса на Visual С++, прозрачные компоненты – заголовочным файлам классов языка Visual С++.

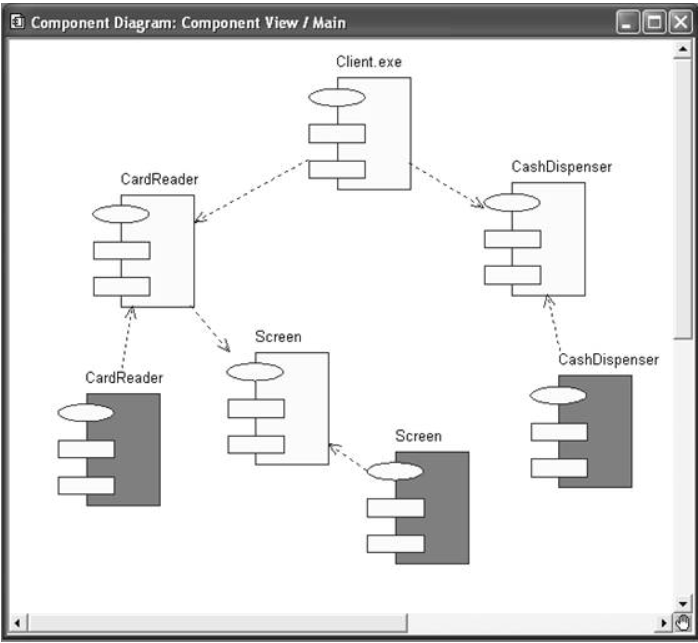
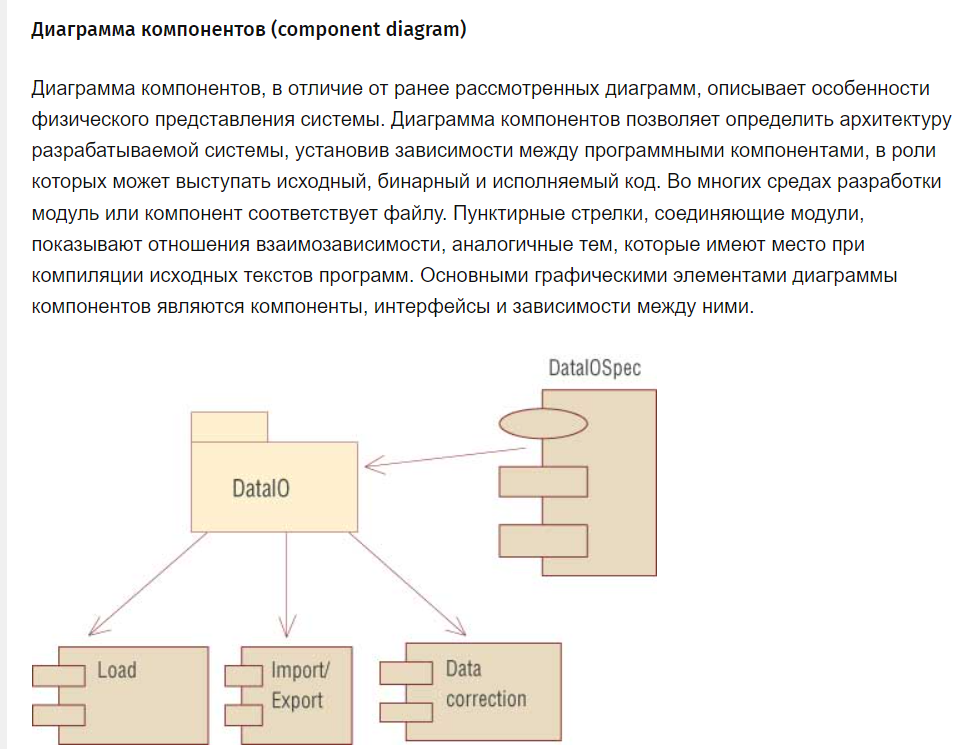


Рисунок 8.1 – Диаграмма компонентов в Rational Rose

Для работы исполняемого файла *Client.exe* необходимы заголовочные файлы *CardReader* и *CashDispenser*, для того чтобы класс *CardReader* мог быть скомпилирован, класс *Screen* должен уже существовать. В свою очередь, заголовочные классы *CardReader, CashDispenser* и *Screen* используются для компиляции соответствующих им файлов на языке *Visual С++.* После компиляции всех классов может быть создан исполняемый файл *Client.exe.*

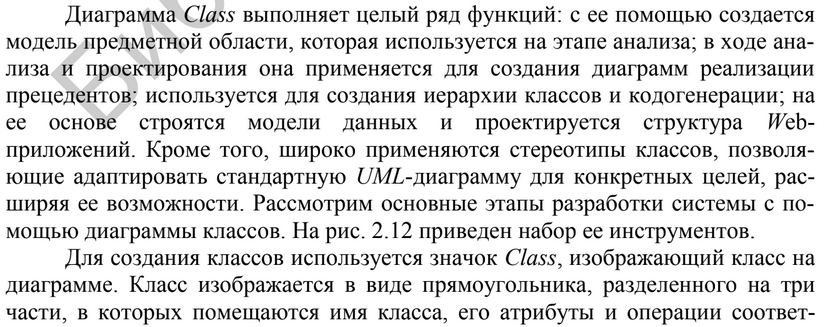
Для построения приведенной диаграммы компонентов были использованы некоторые из ее инструментов. Значок *Package specification* позволяет отобразить определение пакета, а значок *Package body*  выполняет описание тела пакета. Обычно эти инструменты связаны между собой. Здесь *Package specification* – заголовочный файл с расширением .h, а *Package body* – файл с расширением .cpp. С помощью значка dependency устанавливаются связи между компонентами. Этот тип связи показывает, что классы, содержащиеся в компоненте-клиенте, наследуются, содержат элементы, используют или каким-либо другим образом зависят от классов, которые экспортируются из компонента-сервера.

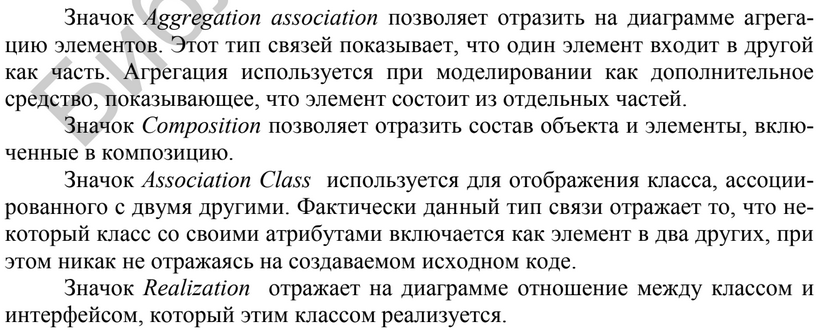
Строка инструментов диаграммы компонентов содержит еще несколько элементов, позволяющих отражать программную реализацию системы. Значок - Component (компонент) на диаграмме представляет собой модуль программного обеспечения, такой как исходный код, двоичный файл, выполняемый файл, динамически подключаемые библиотеки. Компоненты могут использоваться для показа взаимосвязи модулей на этапе компиляции или выполнения программы, а также показывать, какие классы используются для создания определенных компонентов. Значок - *Package* (пакет) позволяет отобразить пакет, который объединяет группу компонентов в модели. Значок - *Main program* (главная программа) позволяет добавить в модель компонент, обозначающий главную программу. Значок - *Subprogram body* (тело подпрограммы) позволяет добавить в модель компонент, обозначающий тело подпрограммы. Значки и - *Task specification* / *body* (определение / тело задачи) позволяют отобразить независимые потоки в многопотоковой системе.

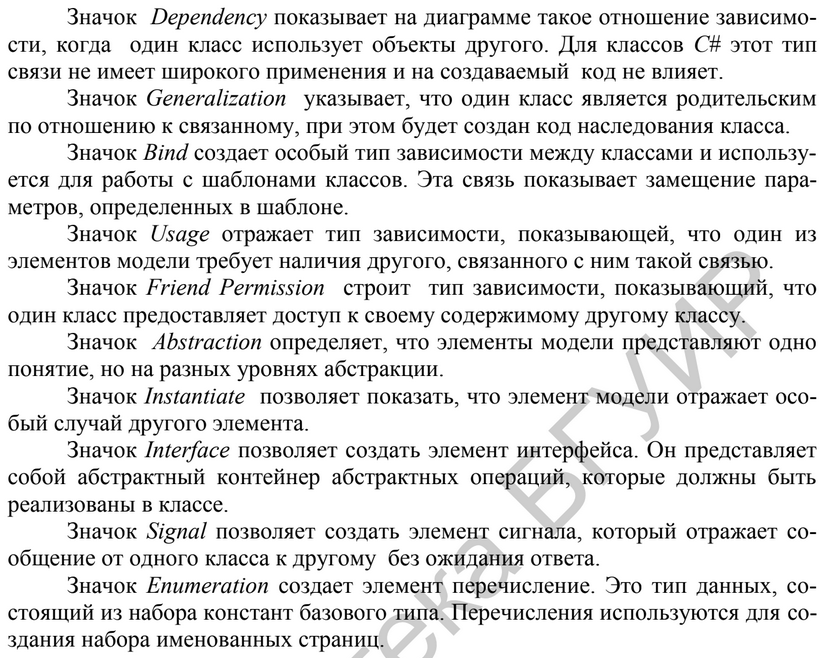


1. Назначение, особенности и построение диаграммы *Class*, виды и особенности связей между классами на диаграммах.









это не из методы дальше, вроде чуть понятнее.

В большинстве случаев классы взаимодействуют друг с другом, что отображается при помощи различного вида связей, влияющих на получаемый при генерации код. В диаграмме классов различают следующие виды связей:

·*Unidirectional association* (однонаправленная ассоциация).

·*Dependency* (зависимость).

·*Association class* (ассоциированный класс).

·*Generalization* (наследование).

·*Realization* (реализация).

*Unidirectional association* – это один из важных и сложных типов связи. Она показывает, что один класс включается в другой как атрибут по ссылке или по значению. На рис 8.7 приведен пример связи Unidirectional association.

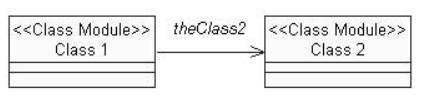


Рисунок 8.7 – Однонаправленная ассоциация

Создаваемый код класса зависит от установленных спецификаций связи. При активизации окна спецификаций открывается ее вкладка General, где содержится следующая информация о связи:

· *Name* – имя связи.

· *Parent* – имя пакета, которому принадлежит связь.

· *Stereotype* – стереотип.

· *Role A/Role B* – имя роли, с которой один класс ассоциируется с другим.

· *Element A/Element B* – имя класса, который ассоциирован с данной ролью.

На вкладке *Detail* указываются дополнительные свойства связи, такие как:

· *Name Direction* – имя связанного класса.

· *Constraints* – выражение семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии.

Вкладка *Role General* отражает настройки переменной, которая будет включена в класс. Поскольку направление связи на рис. 8.8 от *Class*1 к *Class*2, то *Role A* – это *Class*2, а *Role В* – это *Class*1. Рассмотрим вкладку *Role A* General. Она имеет следующие поля:

·*Role* – имя переменной для класса.

·*Element* – имя класса, для которого создается переменная.

·*Export Control* – доступ к элементу; имеется четыре переключателя: *Public, Protected, Private, Implementation*, которые указывают, в какой секции была создана переменная.

Вкладка *Role Detail* детализирует установки для связи и имеет поля:

· *Role* – имя переменной класса.

·*Element* – имя класса, для которого создается переменная.

·*Constraints* – выражение семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии.

·*Multiplicity* – ожидаемое количество объектов данного класса, которые задаются числом, отображаемым рядом со стрелкой связи или буквой «n», указывающей, что количество не лимитировано.

·*Navigable* – направление, в котором действует связь. На какой класс будет направлена стрелка связи, тот и будет включаться в другой. Для того чтобы изменить направление связи, достаточно снять флажок с вкладки *Role A Detail* и установить его во вкладке *Role B Detail*. В случае, когда сняты флажки на обеих вкладках, ни один элемент не будет включен в другой, на диаграмме этому будет соответствовать просто линия.

· *Aggregate* – один класс содержит другой. Для того чтобы показать, что класс *Class*2 входит в класс *Class*1, необходимо установить этот флажок во вкладке *Role B Detail*. При этом стрелка связи на диаграмме приобретает ромб с обратной стороны стрелки (рис. 8.8).

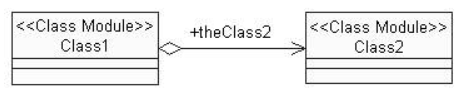


Рисунок 8.8 – Агрегирование класса

· *Static* – общность реквизита для всех объектов данного класса. При этом после инициализации к нему можно обращаться, даже если еще не было создано ни одного объекта класса. Static применяется, чтобы переменные такого типа не тиражировались при создании нового объекта класса.

· *Friend* – указанный класс является дружественным, то есть имеет доступ к защищенным методам и атрибутам.

· *Key/qualifier* – атрибут, который идентифицирует уникальным образом единичный объект, что не влияет на генерацию кода.

Тип связи *Dependency* позволяет показать, что один класс использует объекты другого. Это может осуществляться при передаче параметров или вызове операций класса. В таком случае генератор кода *Rational Rose* включает заголовочный файл в класс, который использует операторы или объекты другого класса. Графическое изображение этого вида связи показано на рис. 8.9.



Рисунок 8.9 – Связь Dependency

Тип связи Association class используется для отображения свойства ассоциации. Свойства сохраняются в классе и соединяются связью *Association* (рис. 8.10). Этот тип не имеет своих спецификаций. Ассоциация предназначена для задания дополнительных атрибутов у связи. Она обозначает, что некоторый класс со своими атрибутами включается как элемент в два других, хотя при генерации кода это не отображается.

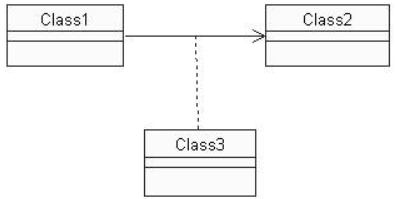
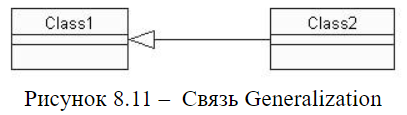


Рис. 8.10 – Связь Association class

Тип связи *Generalization* позволяет указать, что один класс является родительским по отношению к другому, при этом будет создан код наследования класса. Пример такой связи показан на рис. 8.10.



Тип связи *Realization* позволяет показать, что один класс является реализацией, т.е. создан на основе шаблона другого. В Rational Rose для обозначения класса шаблона используется понятие параметризованный класс. Графическое изображение этого типа связи показано на рис. 8.12.



Рисунок 8.12 – Связь Realization

На практике чаще других используются два вида связей: *Unidirectional association* для агрегирования включения ссылок на классы и *Generalization* для создания иерархии наследования.

Ниже на рис. 8.13 приводится пример фрагмента диаграммы классов.

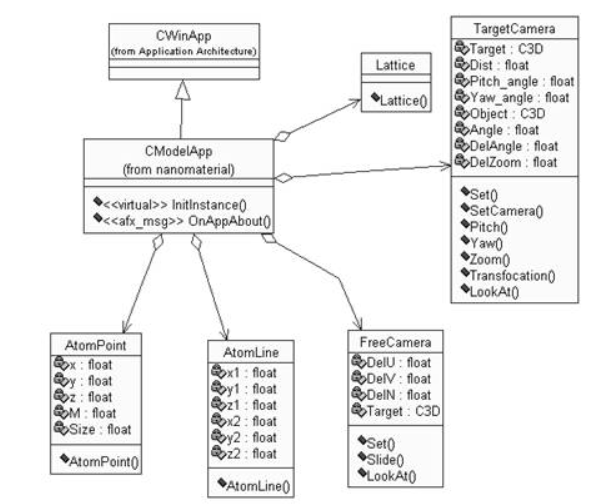


Рисунок 8.13 – Фрагмент диаграммы классов

На ней отражается внутренняя структура системы, описывается наследование и взаимное положение классов относительно друг друга путем определения между ними связей различных типов. В случае программирования на языке *Microsoft Visual C++* разработчик получает дополнительные возможности – доступ ко всей иерархии классов библиотеки *MFC*. Тогда с помощью мастера создания приложений автоматически строится шаблон приложения, в котором классы наследуются из библиотеки *MFC*. На рис. 8.13 показано, что главный класс приложения *CModelApp* наследуется из библиотечного класса *CWinApp*. Связи *CModelApp* с остальными классами показывают, что они содержатся в главном классе приложения.

На диаграмме вместе с атрибутами и операциями представлены следующие классы: *AtomPoint*; *AtomLine*; *Lattice*; *TargetCamera* и *FreeCamera*.

1. Понятие шаблонов проектирования и их классификация. Шаблоны в нотации языка UML.

При реализации проектов по разработке программных систем и моделированию бизнес-процессов встречаются ситуации, когда решение проблем в различных проектах имеют сходные структурные черты. Попытки выявить похожие схемы или структуры в рамках объектно-ориентированного анализа и проектирования привели к появлению понятия шаблона (паттерна), которое из абстрактной категории превратилось в непременный атрибут современных CASE-средств.

Паттерны различаются степенью детализации и уровнем абстракции. Используется следующая общая классификация паттернов по категориям их применения:

· Архитектурные паттерны;

· Паттерны проектирования;

· Паттерны анализа;

· Паттерны тестирования;

· Паттерны реализации.

Архитектурные паттерны (Architectural patterns) – множество предварительно определенных подсистем со спецификацией их ответственности, правил и базовых принципов установления отношений между ними.

Архитектурные паттерны предназначены для спецификации фундаментальных схем структуризации программных систем. Наиболее известными паттернами этой категории являются паттерны GRASP (General Responsibility Assignment Software Pattern). Эти паттерны относятся к уровню системы и подсистем, но не к уровню классов. Как правило, формулируются в обобщенной форме, используют обычную терминологию и не зависят от области приложения.

Паттерны проектирования (Design patterns) – специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними. Паттерны проектирования описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте. Наиболее известными паттернами этой категории являются паттерны GoF (Gang of Four), названные в честь Э. Гаммы, Р. Хелма, Р. Джонсона и Дж. Влиссидеса, которые систематизировали их и представили общее описание. Паттерны GoF включают в себя 23 паттерна. Они не зависят от языка реализации, но их реализация зависит от области приложения.

Паттерны анализа (Analysis patterns) – специальные схемы для представления общей организации процесса моделирования. Паттерны анализа относятся к одной или нескольким предметным областям и описываются в терминах предметной области. Наиболее известными паттернами этой группы являются паттерны бизнес-моделирования ARIS (Architecture of Integrated Information Systems), которые характеризуют абстрактный уровень представления бизнес-процессов. В дальнейшем паттерны анализа конкретизируются в типовых моделях с целью выполнения аналитических оценок или имитационного моделирования бизнес-процессов.

Паттерны тестирования (Test patterns) – специальные схемы для представления общей организации процесса тестирования программных систем. К этой категории паттернов относятся такие паттерны, как тестирование черного ящика, белого ящика, отдельных классов, системы. Паттерны этой категории систематизировал и описал М. Гранд. Некоторые из них реализованы в инструментальных средствах, наиболее известными из которых является IBM Test Studio. В связи с этим паттерны тестирования иногда называют стратегиями или схемами тестирования.

Паттерны реализации (Implementation patterns) – совокупность компонентов и других элементов реализации, используемых в структуре модели при написании программного кода. Эта категория паттернов делится на следующие подкатегории: паттерны организации программного кода, паттерны оптимизации программного кода, паттерны устойчивости кода, паттерны разработки графического интерфейса пользователя и др. Паттерны этой категории описаны в работах М. Гранда, К. Бека, Дж. Тидвелла и др. Некоторые из них реализованы в популярных интегрированных средах программирования в форме шаблонов создаваемых проектов. В этом случае выбор шаблона программного приложения позволяет получить некоторую заготовку программного кода.

Паттерны проектирования в нотации языка UML

В сфере разработки программных систем наибольшее применение получили паттерны проектирования GoF, некоторые из них реализованы в популярных средах программирования. При этом паттерны проектирования могут быть представлены в наглядной форме с помощью рассмотренных обозначений языка UML. Паттерн проектирования в контексте языка UML представляет собой параметризованную кооперацию вместе с описанием базовых принципов ее использования.

При изображении паттерна используется обозначение параметризованной кооперации языка UML (рис. 10.1), которая обозначается пунктирным эллипсом. В правый верхний угол эллипса встроен пунктирный прямоугольник, в котором перечислены параметры кооперации, которая представляет тот или иной паттерн.



Рисунок 10.1 – Изображение паттерна в форме параметризованной кооперации

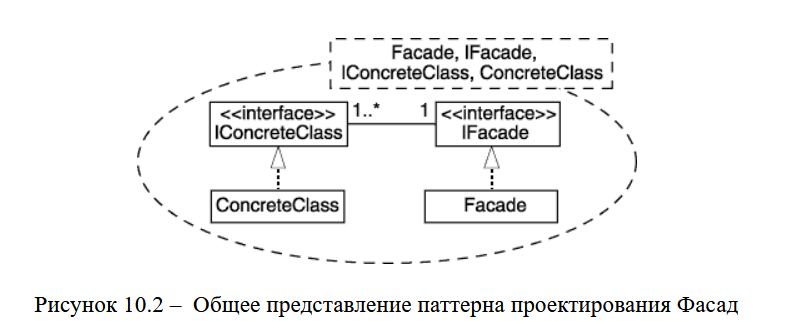
В последующем параметры паттерна могут быть заменены различными классами, чтобы получить реализацию паттерна в рамках конкретной кооперации. Эти параметры специфицируют используемые классы в форме ролей классов в рассматриваемой подсистеме. При связывании или реализации паттерна любая линия помечается именем параметра паттерна, которое является именем роли соответствующей ассоциации. В дополнение к диаграммам кооперации особенности реализации отдельных паттернов представляются с помощью диаграмм последовательности.

Паттерны проектирования позволяют решать различные задачи, с которыми постоянно сталкиваются проектировщики объектно-ориентированных приложений. Ниже представлен полный список паттернов проектирования GoF и краткое описание назначения каждого из них

В качестве примеров рассматриваются два паттерна проектирования, которые нашли наибольшее применение при проектировании программных систем: паттерны Фасад и Наблюдатель.

1. Шаблон “Фасад” и его обозначение в нотации языка UML.

Паттерн Фасад предназначен для замены нескольких разнотипных интерфейсов доступа к определенной подсистеме некоторым унифицированным интерфейсом, что существенно упрощает использование рассматриваемой подсистемы. Общее представление паттерна проектирования Фасад может быть изображено с помощью следующей диаграммы параметризованной кооперации ( рис. 10.2).

Изображенная параметризованная кооперация содержит 4 параметра: класс Facade (Фасад), интерфейс IFacade, интерфейсы IConcreteClass и конкретные классы ConcreteClass, в которых реализованы интерфейсы IConcreteClass. Пунктирная линия со стрелкой в форме треугольника служит для обозначения отношения реализации (не путать с отношением обобщения классов).

При решении конкретных задач проектирования данный паттерн может быть конкретизирован. В этом случае вместо параметров изображенной кооперации должны быть указаны классы, предназначенные для решения отдельных задач.

Ниже приведен пример, который иллюстрирует использование паттерна Фасад для выполнения операций по заданию и считыванию адресов из базы данных сотрудников. Фрагмент соответствующей диаграммы классов содержит 2 класса: Адрес и интерфейс к операциям этого класса IАдрес (рис. 10.3). При задании адреса нового сотрудника необходимо обратиться к этому интерфейсу и последовательно выполнить операции: задатьУлицу(), задатьДом(), задатьКорпус(), задатьКвартиру(), используя в качестве аргумента идентификационный номер нового сотрудника. Для получения информации об адресе сотрудника, необходимо также обратиться к этому интерфейсу и последовательно выполнить операции: прочитатьУлицу(), прочитатьДом(), прочитатьКорпус(), прочитатьКвартиру(), используя в качестве аргумента идентификационный номер интересующего сотрудника.

Очевидно, отслеживать при каждом обращении правильность выполнения этих последовательностей операций неудобно. С этой целью к данному фрагменту следует добавить еще один интерфейс, реализацию паттерна Фасад для рассматриваемой ситуации. Соответствующий фрагмент модифицированной диаграммы классов будет содержать 4 класса (рис. 10.4), изображенные таким образом, чтобы иллюстрировать реализацию параметризованной кооперации.

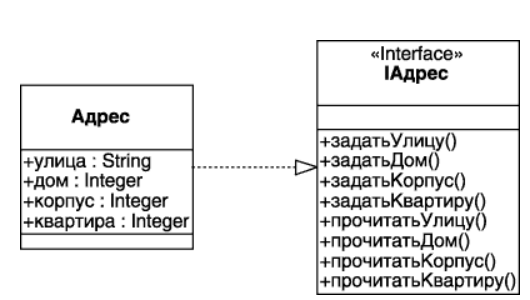


Рисунок 10.3 – Фрагмент диаграммы классов до применения паттерна Фасад

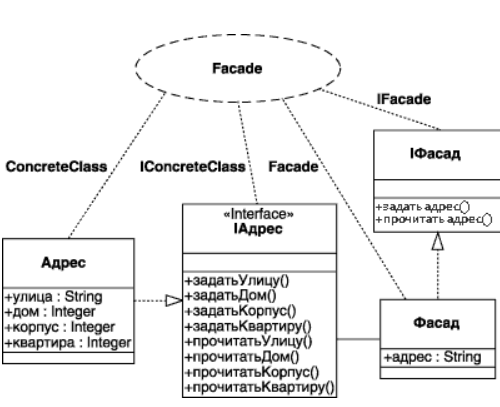


Рисунок 10.4 – Конкретная реализация паттерна проектирования Фасад

При задании адреса нового сотрудника в этом случае достаточно обратиться к интерфейсу IФасад и выполнить единственную операцию: задатьАдрес(), используя в качестве аргумента идентификационный номер нового сотрудника. Для получения информации об адресе сотрудника также достаточно обратиться к этому интерфейсу и выполнить единственную операцию: прочитатьАдрес(), используя в качестве аргумента идентификационный номер интересующего сотрудника. Реализацию данных операций следует предусмотреть в классе Фасад. Взаимодействие объектов этих классов может быть представлено с помощью диаграммы последовательности (рис. 10.5).

Аналогичная диаграмма последовательности может быть построена для выполнения операции по чтению адреса. Использование паттерна Фасад обеспечивает для клиента не только простоту доступа к информации об адресах, но и независимость представления объектов класса Адрес от запросов клиентов. Это обстоятельство особенно актуально при изменении формата представления информации или смене соответствующей базы данных. В этом случае потребуется внести изменения только в реализацию операций класса Фасад.

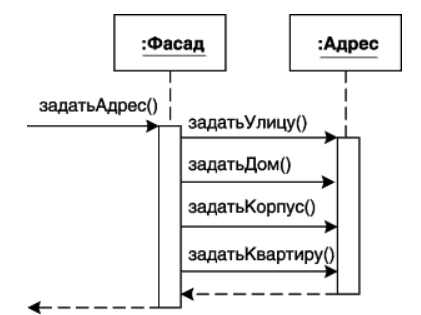
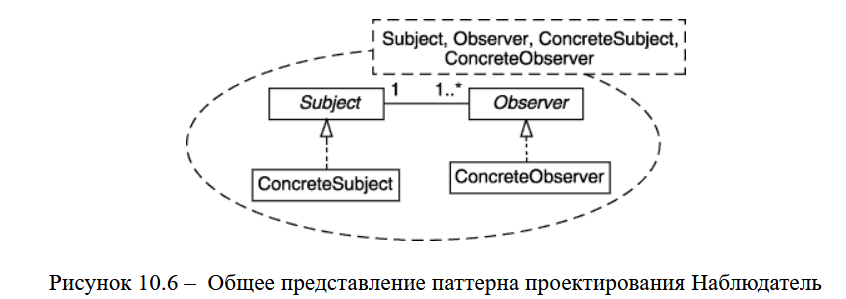


Рисунок 10.5 – Диаграмма последовательности для выполнения операции задания адреса

1. Шаблон “Наблюдатель” и его обозначение в нотации языка UML.

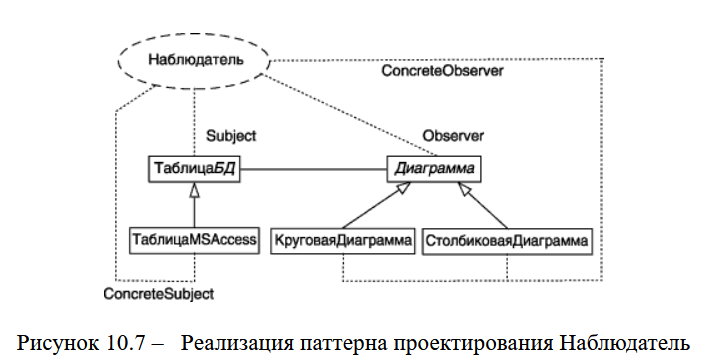
Паттерн Наблюдатель предназначен для контроля изменений состояния объекта и передачи информации об изменении этого состояния множеству клиентов. В общем случае паттерн Наблюдатель также может быть изображен в виде параметризованной кооперации (рис. 10.6).



Изображенная параметризованная кооперация содержит 4 параметра: абстрактный класс Subject (Субъект), класс ConcreteSubject (Конкретный Субъект), абстрактный класс Observer (Наблюдатель) и класс ConcreteObserver (Конкретный Наблюдатель). Пунктирная линия со стрелкой в форме треугольника служит для обозначения отношения обобщения классов.

При решении конкретных задач проектирования данный паттерн также может быть конкретизирован. В этом случае вместо параметров изображенной кооперации должны быть указаны классы, предназначенные для решения отдельных задач.

Рассмотрим пример, который иллюстрирует использование паттерна Наблюдатель для отслеживания изменений в таблице БД и отражении этих изменений на диаграммах. Для определенности можно использовать таблицу БД и две диаграммы – круговую и столбиковую. Фрагмент соответствующей диаграммы классов содержит 5 классов ( рис. 10.7).



В этом случае за субъектом Таблицей MS Access может "следить" произвольное число наблюдателей, причем их добавление или удаление не влияет на представление информации в БД. Класс Таблица MS Access реализует операции по отслеживанию изменений в соответствующей таблице, и при их наличии сразу информирует абстрактного наблюдателя. Тот в свою очередь вызывает операции по перерисовке соответствующих диаграмм у конкретных наблюдателей, в качестве которых выступают классы Круговая Диаграмма и Столбиковая Диаграмма.

Использование паттерна Наблюдатель не только упрощает взаимодействие между объектами соответствующих классов, но и позволяет вносить изменения в реализацию операций классов субъекта и наблюдателей независимо друг от друга. При этом процесс добавления или удаления наблюдателей никак не влияет на особенности реализации класса субъекта.

В заключении следует отметить, что язык UML представляет собой нотацию для визуального моделирования программных систем и бизнес-процессов. В то же время описание языка UML не содержит сведений относительно того, каким образом и в какой последовательности следует разрабатывать канонические диаграммы при выполнении конкретных проектов. Соответствующая информация относится к области методологии проектирования программных систем.

1. Декомпозиция программной системы на модули. Принцип модульности. Оценка сложности программной системы через принцип модульности. Затраты на работу с модулями.
2. Определение модуля. Связность и сцепление модулей. Типы связности и сцепления модулей.
3. Создание модели предметной области программной системы с помощью диаграммы классов.
4. Создание модели анализа с помощью диаграммы классов. Различные стереотипы для классов и их назначение.
5. Декомпозиция системы. Правила выделения подсистем.
6. Особенности создания шаблона приложения в среде *Rational Rose* с использованием библиотеки *MFC*. Структура и классы приложения.
7. Функциональные возможности *Rational Rose*: модуль *Component Assignment Too*l, компонент *Model Assistant*, обновление кода по модели и модели по коду.
8. Особенности генерации исходного кода в среде *Rational XDE*. Способы синхронизации модели.
9. Назначение, возможности, особенности использования модуля *Web Modeler* в *Rational Rose*.
10. Возможности и особенности построения *Web*-модели в среде *Rational XDE*.
11. Понятие проектных рисков. Действия по управлению рисками.
12. Статический и динамический аспекты *Rational Unified Process (RUP)*.
13. Принципы и стадии разработки ПС в технологии *Rational Unified Process.*
14. Содержание и результаты первой и второй стадий в технологии *Rational Unified Process.*
15. Содержание и результаты третей и четвертой стадий в технологии *Rational Unified Process*.
16. Этапы создания программных средств в технологии *Oracle*.
17. Процессы создания программных средств в технологии *Oracle.*
18. Сравнительный анализ технологий создания ПС *Rational Unified Process*, *Oracle*
19. Понятия CASE-средство, CASE-система, CASE-технология, CASE-индустрия и различия между ними.