# ТЕМА 1. АНАЛИЗ ТРЕБОВАНИЙ

Анализ требований – это процесс сбора требований к программному обеспечению (ПО), их систематизации, документированию, анализа, выявления противоречий, неполноты, разрешения конфликтов в процессе разработки программного обеспечения.

Анализ требований включает три типа деятельности:

- Сбор требований: общение с клиентами и пользователями, чтобы определить, каковы их требования.
- Анализ требований: определение, являются ли собранные требования неясными, неполными, неоднозначными или противоречащими; решение этих проблем.
- Документирование требований: требования могут быть задокументированы в различных формах, таких как простое описание, сценарии использования, пользовательские истории, или спецификации процессов.

**Типы требований:**

1. Требования клиентов

- Требования эксплуатации или развёртывания: Где система будет использоваться?
- Профиль миссии или сценарий: Как система достигнет целей миссии?
- Требования производительности: Какие параметры системы являются критическими для достижения миссии?
- Сценарии использования: Как различные компоненты системы должны использоваться?
- Требования эффективности: Насколько эффективной должна быть система для выполнения миссии?
- Эксплуатационный жизненный цикл: Как долго система будет использоваться?
- Окружающая среда: Каким окружением система должна будет эффективно управлять?

2. Функциональные требования

Функциональные требования идентифицируют задачи или действия, которые должны быть выполнены. Функциональные требования определяют действия, которые система должна быть способной выполнить, связь входа/выхода в поведении системы.

3. Нефункциональные требования

Нефункциональные требования – требования, которые определяют критерии работы системы в целом, а не отдельные сценарии поведения. Нефункциональные требования определяют системные свойства такие как про-

изводительность, удобство сопровождения, расширяемость, надежность, средовые факторы эксплуатации.

4. Производные требований

Требования, которые подразумеваются или преобразованы из высокоуровневого требования. Например, требование для большего радиуса действия или высокой скорости может привести к требованию низкого веса.

## Списки требований

Традиционный способ документировать требования – это создание списков требований. В сложной системе такие списки требований могут занимать сотни страниц.

Преимущества:

- Обеспечивает контрольный список требований.
- Обеспечивает договор между заказчиками и разработчиками.
- Для большой системы может обеспечить описание высокого уровня.

Недостатки:

- Большой объем. Фактически невозможно прочитать такие документы в целом и получить чёткое понимание системы.
- Такие списки требований перечисляют отдельные требования абстрактно, оторвано друг от друга и от контекста использования. Эта абстракция лишает возможности видеть, как требования связываются между собой или работают вместе.
- Не возможно верно расположить требования по приоритетам; в то время как список, действительно облегчает приоретизацию отдельных пунктов, удаление одного пункта из контекста может сделать весь сценарий использования или деловое требование бесполезным.
- Абстракция увеличивает вероятность неверного трактования требований; поскольку чем больше число людей будет их читать, тем большее будет число (различных) интерпретаций системы.
- Абстракция означает, что чрезвычайно трудно убедиться, что у вас есть все необходимые требования.

## Прототипы (опытные образцы)

Прототипы – макеты системы. Макеты дают возможность пользователям представить систему, которая ещё не построена. Опытные образцы помогают пользователям представить, на что будет похожа система, и облегчают пользователям принятие проектных решений, не дожидаясь окончания постройки системы. Ранние обзоры системы приводят к меньшему количеству изменений на поздних стадиях разработки и следовательно значительно уменьшают затраты.

Недостатки:

- Менеджерам, как только они видят опытный образец, сложно понять, что окончательный проект не будет разработан ещё некоторое время.
- Проектировщики часто чувствуют себя вынужденными использовать опытный образец в реальной системе, потому что они боятся «напрасно тратить время», начиная всё сначала.
- Опытные образцы преимущественно помогают с проектными решениями и дизайном пользовательского интерфейса. Однако они не могут сказать, какими были первоначальные требования.
- Проектировщики и конечные пользователи могут слишком сильно сосредоточиться на дизайне пользовательского интерфейса и слишком мало на производстве системы, которая служит бизнес-процессу.
- Прототипы отлично подходят для пользовательских интерфейсов, но мало полезны для сложной обработки данных или асинхронных процессов, которые могут вовлечь сложные обновления базы данных и/или вычисления.

## Сценарии использования

Вариант использования (англ. Use Case) – техника для документации потенциальных требований для создания новой системы или изменения существующей. Каждый вариант описывает один или несколько способов взаимодействия системы с конечным пользователем или другой системой, для достижения определённой цели. Варианты использования обычно избегают технического жаргона, предпочитая вместо этого язык конечного пользователя или эксперта в данной области. Создаются совместно специалистами по сбору требований и заинтересованными лицами.

Варианты использования – наиболее важный инструмент для моделирования требований с целью спецификации функциональных возможностей разрабатываемого программного обеспечения или системы в целом. Они могут содержать дополнительное текстовое описание всех способов, которыми пользователи могут работать с программным обеспечением или системой. Полнота функциональных требований к разрабатываемой системе достигается спецификацией всех вариантов использования с соответствующими сценариями, отражающими все пожелания и потребности пользователей к разрабатываемой системе.

## Спецификация требований программного обеспечения

Спецификация требований программного обеспечения (англ. Software Requirements Specification, SRS) – законченное описание поведения системы, которую требуется разработать. Включает ряд пользовательских сценариев (англ. Use cases), которые описывают все варианты взаимодействия между пользователями и программным обеспечением. Пользовательские сценарии являются средством представления функциональных требований. В дополне-

ние к пользовательским сценариям, спецификация также содержит нефункциональные требования, которые налагают ограничения на дизайн или реализацию (такие как требования производительности, стандарты качества, или проектные ограничения).

Структура SRS (Рекомендуемая стандартом IEEE 830)

1. Введение
   1.1. Цели
   1.2. Соглашения о терминах
   1.3. Предполагаемая аудитория и последовательность восприятия
   1.4. Масштаб проекта
   1.5. Ссылки на источники
2. Общее описание
   2.1. Видение продукта
   2.2. Функциональность продукта
   2.3. Классы и характеристики пользователей
   2.4. Среда функционирования продукта (операционная среда)
   2.5. Рамки, ограничения, правила и стандарты
   2.6. Документация для пользователей
   2.7. Допущения и зависимости
3. Функциональность системы
   3.1. Функциональный блок X (таких блоков может быть несколько)
      3.1.1. Описание и приоритет
      3.1.2. Причинно-следственные связи, алгоритмы (движение процессов, workflows)
      3.1.3. Функциональные требования
4. Требования к внешним интерфейсам
   4.1. Интерфейсы пользователя (UX)
   4.2. Программные интерфейсы
   4.3. Интерфейсы оборудования
   4.4. Интерфейсы связи и коммуникации
5. Нефункциональные требования
   5.1. Требования к производительности
   5.2. Требования к сохранности (данных)
   5.3. Критерии качества программного обеспечения
   5.4. Требования к безопасности системы
6. Прочие требования
   6.1. Приложение А: Глоссарий
   6.2. Приложение Б: Модели процессов и предметной области и другие диаграммы
   6.3. Приложение В: Список ключевых задач

## Видение проекта

Описание Видения Продукта не обязательно должно быть большим и сложным. Формат, предложенный Романом Пичлером и описанный в его книге “Agile Product Management with Scrum”, предлагает ответить на пять простых вопросов:

1. Кто будет использовать продукт? Кто его покупает? Кто использует? Кто целевая аудитория?
2. Какие нужды пользователей продукт удовлетворяет? Какую пользу приносит продукт? Как он облегчает жизнь пользователям?
3. Каковы критические атрибуты, чтобы удовлетворить эти нужды и сделать продукт успешным? В каких областях продукт должен быть выдающимся?
4. Чем продукт похож на существующие аналоги и чем он от них отличается? Есть ли аналоги в нашей организации? Есть ли аналоги на рынке?
5. Какой у нас есть срок и бюджет, чтобы запустить продукт?

Структура Vision

1. Введение
   1.1 Цель
2. Назначение
   2.1 Определение проблемы
   2.2 Определение назначения изделия
3. Описания пользователей и совладельцев
   3.1 Демография рынка
   3.2 Пользовательская среда
   3.3 Профили пользователей
   3.4 Ключевые потребности совладельцев/пользователей
   3.5 Альтернативы
4. Краткий обзор программы
   4.1 Перспектива программы
   4.2 Сводка возможностей
   4.3 Предположения и зависимости
5. Возможности программы
6. Другие требования программы
   6.1 Применяемые стандарты
   6.2 Системные требования
   6.3 Эксплуатационные требования
   6.4 Внешние требования
7. Требования к документации
   7.1 Руководство пользователя
   7.2 Интерактивная справка

7.3Руководства по установке, конфигурированию, файл ReadMe
7.4Маркировка и пакетирование

**Техническое задание**

Техническое задание – технический документ(спецификация), оговаривающий набор требований к системе и утверждённый как заказчиком/ пользователем, так и исполнителем/ производителем системы. Такая спецификация может содержать так же системные требования и требования к тестированию.

Техническое задание позволяет:

- Исполнителю – понять суть задачи, показать заказчику «технический облик» будущего изделия, программного изделия или автоматизированной системы;
- Заказчику – осознать, что именно ему нужно;
- Обеим сторонам – представить готовый продукт;
- Исполнителю – спланировать выполнение проекта и работать по намеченному плану;
- Заказчику – требовать от исполнителя соответствия продукта всем условиям, оговоренным в ТЗ;
- Исполнителю – отказаться от выполнения работ, не указанных в ТЗ;
- Заказчику и исполнителю – выполнить попунктную проверку готового продукта(приемочное тестирование – проведение испытаний);
- Избежать ошибок связанных с изменением требований (на всех стадиях и этапах создания, за исключением испытаний).

Структура ТЗ

1. Общие сведения
   1.1. Полное наименование системы и ее условное обозначение;
   1.2. Шифр темы или шифр (номер) договора;
   1.3. Наименование предприятий (объединений) разработчика и заказчика (пользователя) системы и их реквизиты;
   1.4. Перечень документов, на основании которых создается система, кем и когда утверждены эти документы;
   1.5. Плановые сроки начала и окончания работы по созданию системы;
   1.6. Сведения об источниках и порядке финансирования работ;
   1.7. Порядок оформления и предъявления заказчику результатов работ по созданию системы (ее частей), по изготовлению и наладке отдельных средств (технических, программных, информационных) и программно-технических (программно-методических) комплексов системы.
2. Назначение и цели создания системы
3. Характеристика объекта автоматизации

3.1. Краткие сведения об объекте автоматизации или ссылки на документы, содержащие такую информацию;
3.2. Сведения об условиях эксплуатации объекта автоматизации и характеристиках окружающей среды.

4. Требования к системе
   4.1. Требования к системе в целом;
   4.2. Требования к функциям (задачам), выполняемым системой;
   4.3. Требования к видам обеспечения.
5. Состав и содержание работ по созданию системы
   5.1. Перечень документов по ГОСТ 34.201, предъявляемых по окончании соответствующих стадий и этапов работ;
   5.2. Вид и порядок проведения экспертизы технической документации (стадия, этап, объем проверяемой документации, организация-эксперт);
   5.3. Программа работ, направленных на обеспечение требуемого уровня надежности разрабатываемой системы (при необходимости);
   5.4. Перечень работ по метрологическому обеспечению на всех стадиях создания системы с указанием их сроков выполнения и организации-исполнителей (при необходимости).
6. Порядок контроля и приемки системы
   6.1. Виды, состав, объем и методы испытаний системы и ее составных частей (виды испытаний в соответствии с действующими нормами, распространяющимися на разрабатываемую систему);
   6.2. Общие требования к приемке работ по стадиям (перечень участвующих предприятий и организаций, место и сроки проведения), порядок согласования и утверждения приемочной документации;
   6.3. Статус приемочной комиссии (государственная, межведомственная, ведомственная).
7. Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу системы в действие
   7.1. Приведение поступающей в систему информации (в соответствии с требованиями к информационному и лингвистическому обеспечению) к виду, пригодному для обработки с помощью ЭВМ;
   7.2. Изменения, которые необходимо осуществить в объекте автоматизации;
   7.3. Создание условий функционирования объекта автоматизации, при которых гарантируется соответствие создаваемой системы требованиям, содержащимся в ТЗ;
   7.4. Создание необходимых для функционирования системы подразделений и служб;
   7.5. Сроки и порядок комплектования штатов и обучения персонала.
8. Требования к документированию

8.1. Согласованный разработчиком и заказчиком системы перечень подлежащих разработке комплектов и видов документов, соответствующих требованиям ГОСТ 34.201 и научно-технической документации отрасли заказчика; перечень документов, выпускаемых на машинных носителях; требования к микрофильмированию документации;

8.2. Требования по документированию комплектующих элементов межотраслевого применения в соответствии с требованиями ЕСКД и ЕСПД;

8.3. При отсутствии государственных стандартов, определяющих требования к документированию элементов системы, дополнительно включают требования к составу и содержанию таких документов.

9. Источники разработки: документы и информационные материалы (технико-экономическое обоснование, отчеты о законченных научно-исследовательских работах, информационные материалы на отечественные, зарубежные системы-аналоги и др.), на основании которых разрабатывалось ТЗ и которые должны быть использованы при создании системы.

# ТЕМА 2. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАМНОГО ОБЕСПЕЧЕНИЯ

Наиболее распространёнными проблемами, возникающими в процессе разработки ПО, считают:

- *Недостаток прозрачности.* В любой момент времени сложно сказать, в каком состоянии находится проект и каков процент его завершения. Данная проблема возникает при недостаточном планировании структуры (или архитектуры) будущего программного продукта, что чаще всего является следствием отсутствия достаточного финансирования проекта: программа нужна, сколько времени займёт разработка, каковы этапы, можно ли какие-то этапы исключить или сэкономить следствием этого процесса является то, что этап проектирования сокращается.
- *Недостаток контроля.* Без точной оценки процесса разработки срываются графики выполнения работ и превышаются установленные бюджеты. Сложно оценить объём выполненной и оставшейся работы. Данная проблема возникает на этапе, когда проект, завершённый чуть более чем наполовину, продолжает разрабатываться после дополнительного финансирования без оценки степени завершённости проекта.
- *Недостаток трассировки.*
- *Недостаток мониторинга.* Невозможность наблюдать ход развития проекта не позволяет контролировать ход разработки в реальном времени. С помощью инструментальных средств менеджеры проектов принимают решения на основе данных, поступающих в реальном времени. Данная проблема возникает в условиях, когда стоимость обучения менеджмента владению инструментальными средствами сравнима со стоимостью разработки самой программы.
- *Неконтролируемые изменения.* У потребителей постоянно возникают новые идеи относительно разрабатываемого программного обеспечения. Влияние изменений может быть существенным для успеха проекта, поэтому важно оценивать предлагаемые изменения и реализовывать только одобренные, контролируя этот процесс с помощью программных средств. Данная проблема возникает вследствие нежелания конечного потребителя использовать те или иные программные среды. Например, когда при создании клиент-серверной системы потребитель предъявляет требования не только к операционной системе на компьютерах-клиентах, но и на компьютере-сервере.
- *Недостаточная надёжность.* Самый сложный поиск и исправление ошибок в программах на ЭВМ. Поскольку число ошибок в программах заранее неизвестно, то заранее неизвестна и продолжительность отладки программ и отсутствие гарантий отсутствия ошибок в программах. Следует отметить, что привлечение доказательного подхода к проектированию ПО поз-

воляет обнаружить ошибки в программе до её выполнения. Например, при попытке создать программу, требующую средств высокого уровня, с помощью средств низкого уровня. Например, при попытке создать средства автоматизации с СУБД на ассемблере. В результате исходный код программы получается слишком сложным и плохо поддающимся структурированию.

- *Неправильный выбор методологии разработки программного обеспечения.* Процесс выбора необходимой методологии может проблемно отразиться на всех показателях программного обеспечения - это его гибкость, стоимость и функциональность. Так называемые гибкие методологии разработки помогают решить основные проблемы, однако, стоит отметить, что и каскадная модель (waterfall)так же имеет свои преимущества. В некоторых случаях наиболее целесообразным будет применение гибридных методологий в связке Agile +каскадная модель + MSF + RUP и т.д.
- *Отсутствие гарантий качества и надежности программ из-за отсутствия гарантий отсутствия ошибок в программах вплоть до формальной сдачи программ заказчикам.* Данная проблема не является проблемой, относящейся исключительно к разработке ПО. Гарантия качества это проблема выбора поставщика товара(не продукта).

Жизненный цикл – ряд событий происходящих с системой в процессе её создания и дальнейшего использования.

Модель жизненного цикла – структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, использования и сопровождения ПО. Эти модели можно разделить на 3 группы:

- инженерный подход;
- с учетом спецификации задачи;
- современные технологии быстрой разработки.

**Инженерный подход**

1) Модель кодирования и исправления ошибок(создана в 60-70 гг.). Содержит такие этапы проектирования:
   - постановка задачи;
   - выполнение задачи до завершения, либо отмены;
   - проверка результата;
   - при необходимости переход к первому шагу.

2) Каскадная модель (водопад waterfall). Каскадная модель была разработана Винстоном Райсом 70 гг. и представлена на рис.1.1.

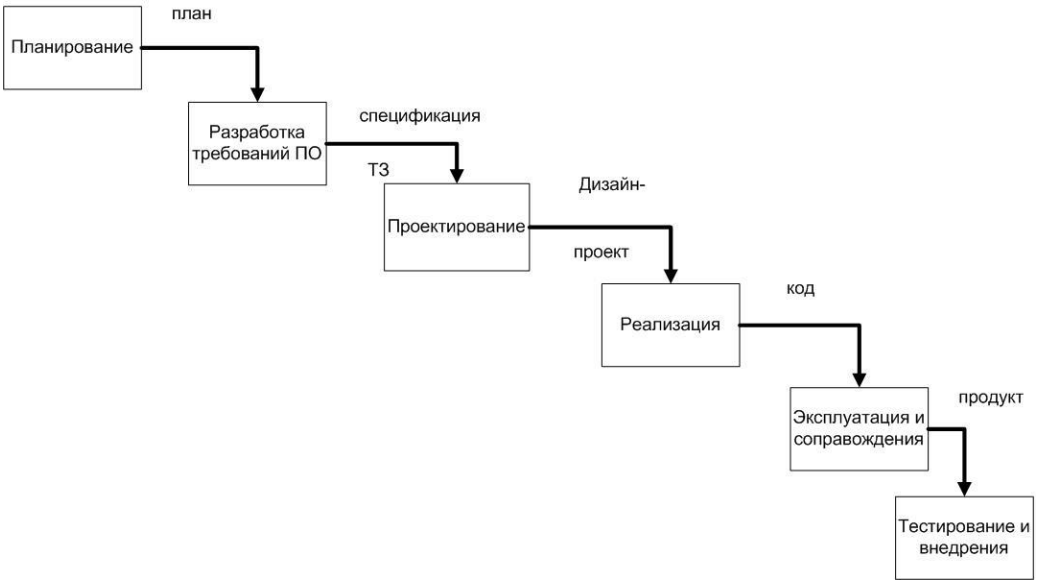



Рисунок 1.1 – Каскадная модель жизненного цикла ПО

Преимущества:

- последовательное выполнение этапов проекта в строго фиксированном порядке;
- позволяет оценивать качество на каждом этапе проекта.

Недостатки:

- отсутствие обратных связей между этапами;
- высокий уровень формализации;
- не соответствует реальным условиям разработки ПО.

3) Каскадная модель с промежуточным контролем (водоворот). Аналогична каскадной модели, но содержит обратные связи.

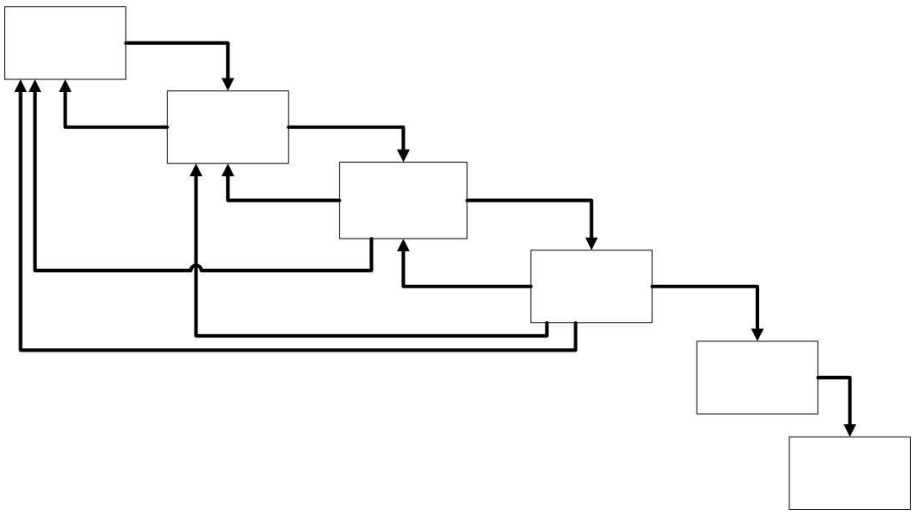


Рисунок 1.2 – Каскадная модель с промежуточным контролем жизненного цикла ПО

Преимущества:

− появления обратных связей.

Недостатки:

− десятикратное увеличения затрат на разработку.

4) V–образная модель (разрабатывается через тестирование).

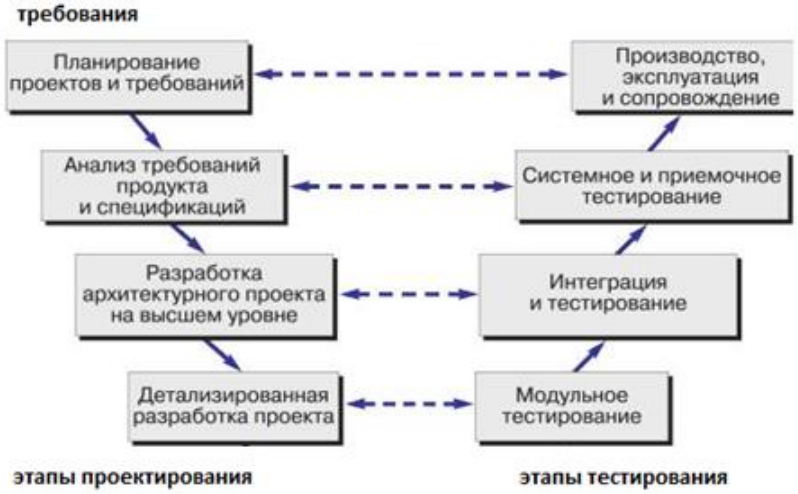



Рисунок 1.3 – V–образная модель жизненного цикла ПО

Преимущества:

- В модели особое значение придается планированию, направленному на верификацию и аттестацию разрабатываемого продукта на ранних стадиях его разработки. Фаза модульного тестирования подтверждает правильность детализированного проектирования. Фазы интеграции и тестирования реализуют архитектурное проектирование или проектирование на высшем уровне. Фаза тестирования системы подтверждает правильность выполнения этапа требований к продукту и его спецификации.
- В модели предусмотрены аттестация и верификация всех внешних и внутренних полученных данных, а не только самого программного продукта.
- В V-образной модели определение требований выполняется перед разработкой проекта системы, а проектирование ПО перед разработкой компонентов.
- Модель определяет продукты, которые должны быть получены в результате процесса разработки, причём каждые полученные данные должны подвергаться тестированию.
- Благодаря модели менеджеры проекта могут отслеживать ход процесса разработки, так как в данном случае вполне возможно воспользоваться временной шкалой, а завершение каждой фазы является контрольной точкой.

Недостатки:

- Модель не предусматривает работу с параллельными событиями.

- В модели не предусмотрено внесение требования динамических изменений на разных этапах жизненного цикла.
- Тестирование требований в жизненном цикле происходит слишком поздно, вследствие чего невозможно внести изменения, не повлияв при этом на график выполнения проекта.
- В модель не входят действия, направленные на анализ рисков.
- Некоторый результат можно посмотреть только при достижении низа буквы V.

**Учитывающие специфику ПО**

1. Модель на основе разработки прототипов.

Прототип – действующий компонент ИС, реализующий отдельные функции и внешние интерфейсы. Каждая итерация соответствует созданию фрагмента или версии ИС, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов, планируются работы следующей итерации.

На каждой итерации оцениваются:

− Риск превышения сроков и стоимости проекта;
− Необходимость выполнения еще одной итерации;
− Степень полноты и точности понимания требований к системе;
− Целесообразность прекращения проекта.

Прототипирование – использование прототипов на ранних стадиях жизненного цикла для того, чтобы:

− прояснить неясные требования (создается прототип пользовательского интерфейса);
− выбрать одно из ряда концептуальных решений (реализация сценариев);
− проанализировать осуществимость проекта.

Классификация прототипов:

1) Горизонтальные и вертикальные;
2) Одноразовые и эволюционные;
3) Бумажные, электронные, раскадровки и т.д.

Горизонтальные – моделирование интерфейса пользователя, не затрагивая логику обработки и базы данных.

Вертикальные – осуществляют вертикальный срез системы, анализ применимости , проверку архитектурных концепций (алгоритмы).

Одноразовые – для быстрой разработки (раскадровка, рисунок) зачастую без программных средств.

Эволюционные – первое приближение системы.

Таблица *1.1. – Классификация прототипов.*

| | Одноразовые | Эволюционные |
|---|---|---|
| Горизонтальные | • Уточняет примеры использования и функциональные требования<br>• Исследует возможные варианты пользовательского интерфейса | • Реализация базовых Usecases (вариантов использования)<br>• Адаптация системы к быстро меняющимся требованиям |
| Вертикальные | • Демонстрация технической осуществимости. | • Реализация и оптимизация основных алгоритмов<br>• Тестирование и настройка производительности системы. |

1) Спиральная модель

Спиральная модель была разработана в середине 1980-х годов Барри Боэмом. Она основана на классическом цикле ДемингаPDCA(plan-do-cgtck-act). При использовании этой модели ИС создается в несколько итераций методом прототипирования. Проект, в процессе своей реализации преодолевает все этапы.

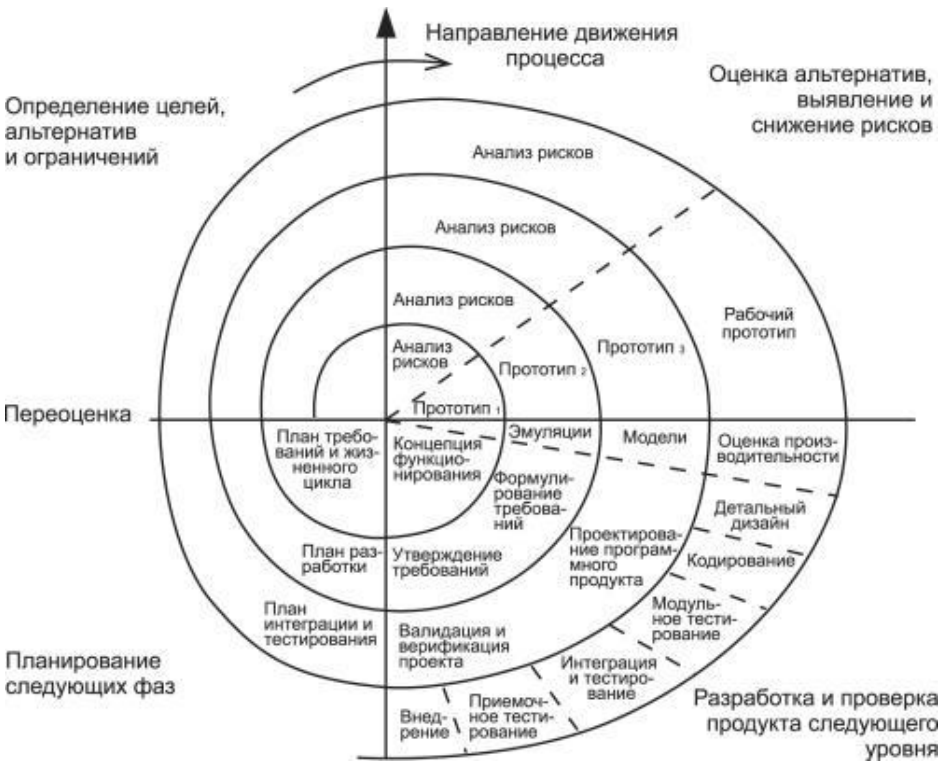



Рисунок 1.4. – Спиральная модель жизненного цикла ПО

Преимущества:

- быстрое получение результата;
- повышение конкурентоспособности;
- изменяющиеся требования не проблемы.

Недостатки:

- нет четкой разграниченности стадий (для больших проектов)

**Современные технологии**

Направлены на быстрое получения результатов, в основном используются в небольших проектах.

Современные технологии или гибкие :

- XP-экстремальные программы;
- SCRUM;
- RUP инкрементная модель.

Стандарты и международные организации, описывающие технологические процессы:

- IEEE – Institute of Electricul and Electronic Engineers – институт инженеров по электротехнике и электронике.
- ISO – International Standarts Organization – международная организация по стандартизации.
- EIA – Electronic Industry Association – ассоциация электронной промышленности.
- IEC – International Electrotehnical Commision – международная комиссия по электротехнике.

Национальные и региональные институты и организации:

- ANSI – American National Standart Institute- американский национальный институт стандартов.
- SEI – Software Engineering Institute – институт программной инженерии;
- ECMA – European Compute Manyfactures Association – Европейская ассоциация производителей компьютерного оборудования.

Стандарты жизненного цикла информационных систем:

- ГОСТ 34.601-90.
- ISO/ EIA 12207:1995(российский аналог – ГОСТ Р ИСО/МЭК 12207-99).
- Custom Development Method (методика Oracle).
- Rational Unified Process (RUP).
- Microsoft Solutions Framework (MSF). Включает 4 фразы: анализ, проектирование, разработка, стабилизация, предполагает использование объектно-ориентированного моделирования.
- Экстремальное программирование. В основе методологии командная работа, эффективная коммуникация между заказчиком и исполнителем в течение всего проекта по разработке ИС. Разработка ведется с использованием последовательно дорабатываемых прототипов.

# ТЕМА 3. МОДЕЛИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ (IDEF0, IDEF3,DFD ДИАГРАММЫ)

**Функциональное моделирование предметной области**

Для решения задач моделирование сложных систем существуют определенные методологии и стандарты. К таким стандартам относятся методологии семейства IDEF, позволяющие исследовать структуру, параметры и характеристики производственно-технических и организационно-экономических систем.

Общая методология IDEF включает частные методологии, основанные на графическом представлении систем: IDEF0, IDEF1, IDEF1.X, IDEF2,…, IDEF14 (idef.com)

Основу подхода, и как следствие, методологии составляет графический язык моделирования систем.

Свойства графического языка:

- полное и выразительное средство, способные наглядно представить широкий спектр деловых, производственных и др. процессов и предприятия на любом уровне детализации;

- обеспечивает точное и локальное описание моделируемых объектов, удобство использования интерпретации этого описания;

- облегчает взаимодействия и взаимопонимания персонала;

- язык может генерироваться рядом CASE средств –средств для реализации этапов жизненного цикла:

- Средства фирмы PLATINUM (BPWin, ERWin);
- продукты фирмы RATIONAL Software (Rational Rose, Rational United Process).

## Методология IDEF0

**IDEF0** – методология функционального моделирования и графическая нотация, предназначенная для формализации и описания бизнес-процессов. Особенностью является её акцент на соподчиненность объектов. Рассматриваются логические отношения между работами, а не их временная последовательность.

В IDEF0 система представляется как совокупность взаимодействующих работ или функций. Такая чисто функциональная ориентация является принципиальной - функции системы анализируются независимо от объектов, которыми они оперируют. Это позволяет более четко смоделировать логику и взаимодействие процессов организации.

Под моделью в IDEF0 понимают описание системы (текстовое и графическое), которое должно дать ответ на заранее определенные вопросы. Та-

ким образом, модель IDEF0, как и любая другая модель, является целевым образованием и предназначена, прежде всего, для исследования объекта.

Моделируемая система рассматривается как подмножество какого-либо более крупного объекта, т.е. система имеет границу, которая отделяет ее от окружающей среды. Одной из первых задач при моделировании является выделение системы (объекта моделирования) из окружающей среды.

Взаимодействие системы с окружающим миром описывается как вход (нечто, что перерабатывается системой), выход (результат деятельности системы), управление (стратегии и процедуры, под управлением которых производится работа) и механизм (ресурсы, необходимые для проведения работы). Находясь под управлением, система преобразует входы в выходы, используя механизмы.

Стандарт представляет организацию как набор модулей.

Цель моделирования (Purpose). Прежде чем строить модуль объекта исследователь должен четко определить цели моделирования, желаемый конечный продукт.

Точка зрения (Viewpoint) – взгляд человека, который видит систему в нужном для моделирования аспекте.

**Типы моделей:**

- модельAS-SI – отражает существующую организацию работы как есть;
- модельTO-BE – то как будет выглядеть система.

**Типы диаграмм:**

- контекстная диаграмма (одна в проекте);
- диаграмма декомпозиции;
- диаграмма дерева узлов;
- диаграмма только для экспозиции (FEO).

Контекстная диаграмма является вершиной древовидной структуры диаграмм и представляет собой самое общее описание системы и ее взаимодействия с внешней средой. После описания системы в целом производится разбиение ее на крупные фрагменты. Этот процесс называется функциональной декомпозицией, а диаграммы, которые описывают каждый фрагмент и взаимодействие фрагментов, называются диаграммами декомпозиции. После декомпозиции контекстной диаграммы проводится декомпозиция каждого большого фрагмента системы на более мелкие и так далее до достижения нужного уровня подробности описания.

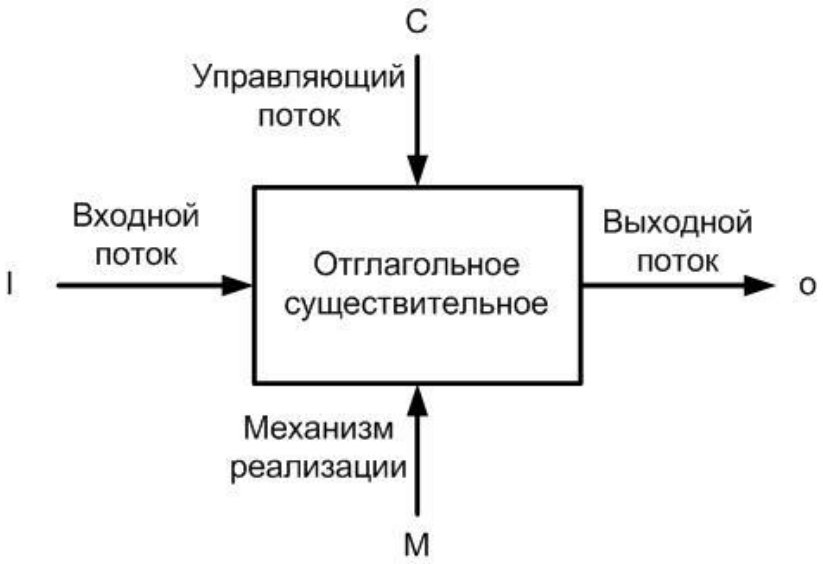



Рисунок 2.1 – Контекстная диаграмма (Название ICOM)

Основные правила:

− наиболее важная функция находиться в верхнем левом углу.

− стрелка входа приходит в левую кромку активности (вход материал или информация, которые используются или преобразуются работой получения результата. Допускается, что работа может не иметь ни одной стрелки входа)

− стрелка управления в верхнюю кромку активности (правила, стратегии, процедуры, стандарты, которыми руководствуется работа, должна хотя бы одна быть стрелка управления)

− стрелка механизма в нижнюю кромку активности (механизм – это ресурсы, которые выполняют работу. Могут не отображаться в модели)

− стрелка выхода – из правой кромки активности (выход – нечто производимое, должен быть хотя бы один выход).

В IDEF0 различают пять видов стрелок:

− Вход (**Input**) - материал или информация, которые используются или преобразуются работой для получения результата (выхода). Стрелка входа рисуется как входящая в левую грань работы.

− Управление (**Control**) - правила, стратегии, процедуры или стандарты, которыми руководствуется работа. Каждая работа должна иметь хотя бы одну стрелку управления. Стрелка управления рисуется как входящая в верхнюю грань работы.

− Выход (**Output**) - материал или информация, которые производятся работой. Каждая работа должна иметь хотя бы одну стрелку выхода. Стрелка выхода рисуется как исходящая из правой грани работы.

− Механизм (**Mechanism**) - ресурсы, которые выполняют работу, например персонал предприятия и т.д. Стрелка механизма рисуется как входящая в нижнюю грань работы.

– Вызов (**Call**) - специальная стрелка, указывающая на другую модель работы. Стрелка вызова рисуется как исходящая из нижней грани работы. В BPWin стрелки вызова используются в механизме слияния и разделения моделей.

## Методология IDEF3

**IDEF3** показывает причинно-следственные связи между ситуациями и событиями в понятной эксперту форме, используя структурный метод выражения значений о том, как функционирует система, процесс или предприятие.

Метод IDEF3 предназначен для разработки моделей, описывающих любую деятельность как упорядоченную последовательность событий и объекты, принимающие участие в этой деятельности.

Система описывается как упорядоченная последовательность событий с одновременным описанием объектов имеющих отношение к моделируемому процессу.

IDEF3 состоит из двух методов:

- ProcessFlowDescription – описание технологических процессов, с указанием того, что происходит на каждом этапе технологического процесса.
- ObjectStateTransitionDescription – описание переходов состояния объектов, с указанием того, какие существуют промежуточные состояния у объектов в моделируемой системе.

  Типы диаграмм:
  - Диаграмма описания последовательности этапов процесса.
  - Диаграмма сети трансформаций состояния объекта.

Единицаработы (unitofwork) – прямоугольник с прямыми углами и имеющий имя отглагольного существительного.

Модели IDEF3 можно применять для более полной детализации моделей IDEF0. Обычно они раскрывают сущность листьев модели IDEF0 (т.е. блоков, которые не имеют потомков). В этом случае еще на этапе разработки модели IDEF0 следует тщательно проработать взаимосвязь этого блока с остальной системой.

Связи показывают взаимоотношения работ.

**Виды связей:**

- Старшая (precedence) – сплошная линия показывает, что работа-источник должна закончится прежде, чем работа-цель начнется.
- Отношения (Relationline) – пунктирная линия, используется для отображения связей между единицами работ (объект порождается в одной работе и используется в другой).

- Потоки объектов (objectflow) – стрелка с двумя наконечниками, применяется для описания того, что объект используется в двух и более единицах работы.

**Перекрестки** (junction) – используется для отображения логики взаимодействия стрелок при слиянии и разветвлении или для отображения множества событий, которые могут или должны быть завершены перед началом следующей работы.


Рисунок 2.2 – Перекрестки

**Типы перекрестков:**

Таблица*2.1 – Типы перекрестков*

| Обозначения | Слияниестрелок(funinjunction) | Разветвление стрелок (funoutjunction) |
|---|---|---|
| AND (&) asynchronous | Все предшествующие процессы должны быть завершены | Все следующие процессы должны быть запущены |
| AND (&) synchronous | Все предшествующие процессы завершены одновременно | Все следующие процессы запускаются одновременно |
| OR (o) asynchronous | Один или несколько предшествующих процессов должны быть завершены | Один или несколько следующих процессов должны быть запущены |
| OR (o) synchronous | Один или несколько предшествующих процессов завершены одновременно | Один или несколько следующих процессы запускаются одновременно |
| OR (x) exclusive | Только один предшествующих процесс завершен | Только один следующий процесс запускается |

**Типы объектов ссылок:**

Таблица *2.2 – Типы объектов ссылок*

| Тип | Цель описания |
|---|---|
| Object | Описывает участие важного объекта в работе |
| Goto | Инструмент циклического перехода (повторяющейся последовательности работ). Если все работы цикла присутствуют на текущей диаграмме, цикл так же может изображаться стрелкой, возвращаться на запуск. Может ссылаться перекресток |
| UOB (Unitbehavior) | Множественное использование работы, но без цикла. Не используется для моделирования автоматически запускающихся работ. |
| NOTE | Применяется для документирования важной информации, относящейся к каким-либо графическим объектам на диаграмме. NOTE является альтернативой внесению текстового объекта в диаграмму. |
| ELAB (Elaboration) | Используется для усовершенствования графиков или их более детального описания. Употребляется для описания разветвления и слияния стрелок на перекрестках. |

## Методология DFD

**DFD** – Date Flow Diagrams– диаграммы потоков данных. Методология графического структурного анализа, описывающая внешние по отношению к системе источники и адресаты данных, логические функции, потоки данных и хранилища данных, к которым осуществляется доступ.

Информационная система принимает извне потоки данных. Для обозначения элементов среды функции системы используют понятие внешней сущности. Внутри системы существуют процессы преобразования информации, порождающие потоки данных. Потоки данных могут поступать на вход к другим процессам, помещаться (и извлекаться) в накоплении данных, передаваться к внешним сущностям.

Модель DFD, как большинство других структурных моделей – иерархическая модель. Каждый процесс, может быть, подвергнут декомпозиции, т.е. разбиению на структурные составляющие отношения между которыми в той же нотации могут быть показаны на отдельной диаграмме. Процесс нижнего уровня сопровождается миниспецификацией (текстовым описанием).

Внешняя сущность – математический предмет или физическое лицо, представляющее собой источник или приемник информации.

Процесс – преобразование входных потоков в выходные в соответствии с определенным алгоритмом.

DFD описывает:

- функции обработки информации(работы) изображается прямоугольником с округлыми краями , имеют входы и выходы, но не поддерживают управления и механизмы.
- документы (стрелки), описывают движение объектов из одной части системы в другую.
- внешние ссылки (externalreferences), обеспечивают интерфейс с внешними объектами, находящимися за границами моделируемой системы(прямоугольник с тенью)
- таблицы для хранения документов (datastore)

В таблице 2.3. приведено сравнение CASEсредств.

Таблица *2.3 – СравнениеCASE средств.*

| Программа | Тип диаграмм | Генерация исходного кода (поддержив. языки) | Обратный инжиниринг (поддержив. языки) | Поддержка баз данных | Интеграция со средствами разработки | Мультипользовательский режим | Среда функционирования | OpenSource | Дополнительные возможности |
|---|---|---|---|---|---|---|---|---|---|
| SybasePower Designer | UML 2.0 | C#, C++,Java Power Builder, Visual Basic | Java Power Builder, Visual Basic | MySQL, Oracle, Access | Java и сертификация под J2EE/EJB 2.0 | Возможно: хранение управление и создание версий | Windows | От 50$ до 300$ | Возможность создания новых шаблонов генерация кода генерирование XML и IOL поддержка XMI |
| Erwin | UM-LDFD IDF0 IDF3 | — | — | 20 различных СУБД и реализован обратный инженерный БД | Oracle, Designer, Rational Rose , Link Object Api | С помощью Model Mart | Windows | От 1000$ до 7000$ | Поддержка отчетов (HTML, RTF, TXT, PDF) перенос структуры БД из СУБД другого типа |
| Dia | NetworkUML 2.0 ER диаграммы (проектирование БД) | — | — | Нет (только проектирование БД) | — | — | IRix, Linux, Windows | Лицензия GPL + | Возможность создания и загрузки экспорт диаграмм сохранение XML |
| SmartDraw | UML 2.0 | — | — | — | Ms Office (Word Power Point, Excel) Visio Smart Draw | — | Windows | От 800$ до 4000$ | Поддержка растровой графики; отчеты,презентаци, документы, процедуры |
| EnterpriseArchitect | UML 2.0 | C++, C#, VB, PHP, Java, Visual Basic, Delphi, Piton | C++, C#, VB, PHP, Java, Visual Basic, Delphi, Piton | Прямое проектирование DDL, обратное - BC | Visual Studio. Net | + | Windows | От 100$ до 900$ | Документация MTML иRTF поддержка: паттернов проектирование макросов |
| VisualParadigm | UML 2.1 | C#, VB.Net, DDL Flash Action Script, Delphi, Feral, Objective.C | C#, VB.Net, DDL, Java, PHP, KML.Net dll, exe Piton, IDL | — | Eclipse, Web Logic, Borland Oracle | + | Linux, Mac OS, Windows | От 60$ до 2000$ | Экспорт JPG, PNG, SVG, PDF редактор форм, XMI поддержка операций с командной строки |

Продолжение таблицы *2.3.*

| Программа | Тип диаграмм | Генерация исходного кода (поддержив. языки) | Обратный инжиниринг (поддержив. языки) | Поддержка баз данных | Интеграция со средствами разработки | Мультипользовательский режим | Среда функционирования | OpenSource | Дополнительные возможности |
|---|---|---|---|---|---|---|---|---|---|
| Umbrello | Все типы UML | C++, C#, PHP, SQL, Java, Java Script, Action Script, Piton, IDL,XML, Perl, Ruby | — | — | — | — | Unix, Linux | + окружение RDF | Импорт (экспорт XM) документация DocBook, XHTML копирование .png |
| ArgoUML | UML 1.3 UML 1.4. | C++, C#, PHP, Java | Из исходного байт кода Java | — | — | — | — | Лицензия BSD + | Автоверификация моделей отчеты – сохранение gifsvg |
| Dia | UML 2.0 Network ER diagram | — | — | Нет (только проектирование БД ERdiagram) | — | — | IRix, Linux Windows | Лицензия GPL + | Воз создание и загрузка экспорт диаграмм сохранение XML |
| UModel | UML 2.1.1 | Java(1.4, 5.0, 6.0), C#(1.2, 2.0, 3.0), VB(7.1, 2.0, 9.0) | Java, C#,VB | — | Плагины Microsoft Visual Studio, Eclipse | + | Windows | От 100$ до 800$ | Документация HTMLWord, RTF; XML – схемы BPWN или (экспорт XVL) |
| RationalRose | UML различные типы диаграмм | C++, Power Builder, SQL, Windows Object Pr0, Ada | C++, Power Builder, SQL, Windows Object Pr0, Ada | Проект ирование | MS Visual Studio, SCC интерфейс | Интегрирование с PVCS | Windows, Unix, Solaris MP UX IBM RS | От 2000$ до 12000$ | Документирование полная поддержка Java, форматов CAB, ZIP |
| QReal | ML 2.1 | C#, .Net | В разработке | Схемы БД | — | + | Linux Mac OS Windows | Лицензия GPL + | Отчеты – авто поход к созданию новых графических редакторов |

# ТЕМА 4. УПРАВЛЕНИЕ ПРОЕКТАМИ ПРИ РАЗРАБОТКЕ ПО

**Управление проектами** – это обеспечение необходимого качества при ограничении стоимости, времени и содержания.

Существует множество типовых процессов производства ПО.

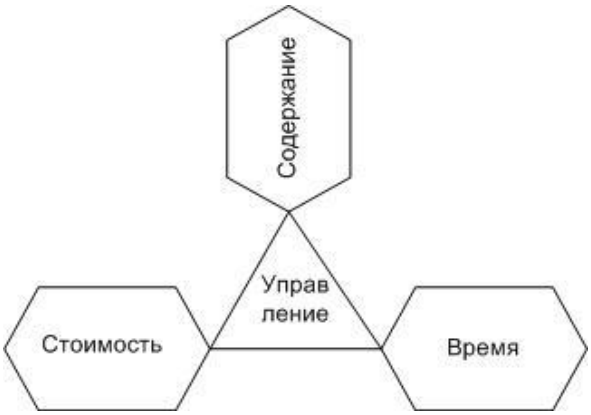



Рисунок 4.1 – Схема связи

**Методология** – это набор практик, методов, метрик, правил, используемых в процессе производства ПО.

**Цель методологии** – распределить ответственность между членами команды, обеспечит взаимозаменяемость, соблюдение сроков и контроля.

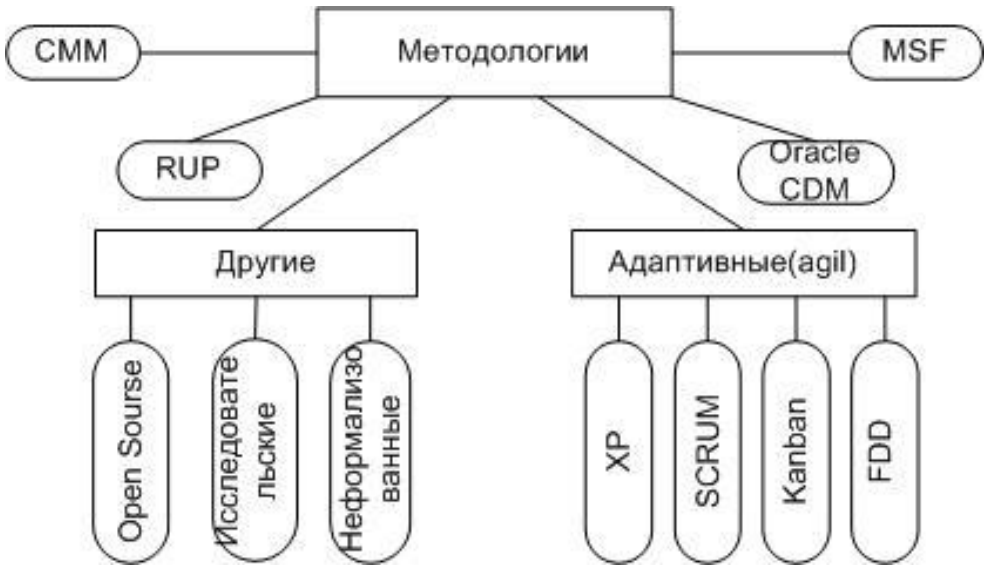



Рисунок 4.2. – Методологии.

Все методологии можно условно разбить на 3 категории:

1. Тяжелые – охватывают все спектры производства ПО, все спектры регламентируются от управления к требованиям до отношений с заказчиком.

Нетерпимы к изменению в процессе разработки, а люди рассматриваются как ресурсы.

2. Средние – основные – масштабированность (правильное планирование и итерационные цикл разработки).

3. Легкие – для небольших групп программ, главное восприимчивость к изменениям, взаимодействие людей в команде.

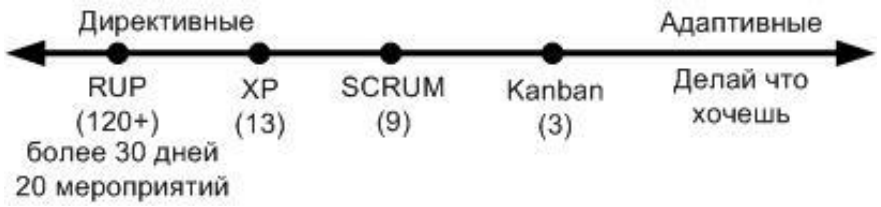



Рисунок 4.3 – Количество практик в методологиях

**Основные функциональные роли в коллективе разработчиков**

Мы придерживаемся ролевой структуры проекта, которая предложена Центром объектно-ориентированной технологии компании IBM. Представление роли разработчиков в организационном комитете, т.е. рассматривают не только разработчиков, но и тех кто не участвует в проекте в качестве исполнителя, но оказывает влияние на постановку задач проекта, выделение ресурсов и обеспечение осуществимости развития работ.

**Внешние роли:**

- *Заказчик* (customer) – реально существующий (в организации, которой подчинена команда, ли вне её) инициатор разработки или кто-либо другой уполномоченный принимать результаты (текущие и окончательные) разработки.
- *Планировщик ресурсов* (planner) – выдвигает и координирует требования к проектам в организации, осуществляющий данную разработку, а так же развивает и направляет план выполнения проекта с точки зрения организации.
- *Менеджер проекта* (projectmanager) – отвечает за развитие проекта в целом, гарантирует, что распределение заданий и ресурсов позволяет выполнить проект, что работы, что предъявления результатов идет по графику, что результаты соответствуют требованиям.
- *Руководитель команды* (teamleader) – производит техническое руководство команды в процессе выполнения проекта.
- *Архитектор* (architect) – отвечает за проектирование архитектуры (проекта) системы, согласовывает развитие работ, связанных с проектом.
- *Проектировщик подсистемы* (designer) – отвечает за проектирование подсистемы или категории классов, определяет реализацию и интерфейсы с другими подсистемами.

- *Эксперт предметной области* (domainexpert) – отвечает за изучение сферы приложения, поддерживает направленность проекта на решение задач данной области.
- *Разработчик* (developer) – реализует проектируемые компоненты, владеет и создает специфические классы и методы, осуществляет кодирование и автоматное тестирование, строит продукт.
- *Разработчик информационной поддержки* (informationdeveloper) – создает документацию, сопровождающую продукт, когда выпускается версия, включаемые в нее инструкционные материалы, ровно как ссылочные и учебные, а так же материалы помощи.
- *Специалист по пользовательскому интерфейсу* (humanfactorsengineer) – отвечает за удобство применения системы.
- *Тестировщик*(tester) – проверяет функциональность, качество и эффективность продукта, строит и исполняет тесты для каждой фазы развития проекта.
- *Библиотекарь* (librarian) – отвечает за создание и видение общей библиотеки проекта, которая содержит все проектные рабочие продукты, а также следит за соответствием рабочих продуктов стандартам.

Совместимыми ролями являются заказчик и тестировщик, а не совместимыми – разработчик и тестировщик.

## CMM (capability maturity model)

Система качества, разработанная SEI (Software Engineering Institute) позволяет точно оценить процесс разработки ПО. Это не технология, не стандарт, для нее нет никаких формальных описаний, не рекомендуется использовать CASE системы.

CMM предназначена для организации эффективного управления разработкой ПО. Она определяет ключевые действия, которые указывают, что нужно сделать для достижения требуемого качества.

**Цель:** гарантирование реализации проекта разработки ПО в заданный срок с заданным бюджетом и высоким качеством.

**Пять уровней CMM:**

I. Уровень Initial (низкий) – процессы разработки никак не организованы, потоки улучшения не принимаются. Успех зависит от индивидуальных способностей сотрудников. Проект представляет собой «Черный ящик».

II. Уровень Repeatable (повторяемый) – на основе анализа успешных проектов повторно использовать подходящие процессы разработки. Положительная практика документируется, организовывается учеба сотрудников, и определяются пути улучшения. Создаются внутрифирменные стандарты, внедряются процессы планирования и контроля за работой. Недостатки: все проблемы решаются по мере их возникновения; успех зависит от конкретных исполнителей.

III.Уровень Defined (определенный) детально стандартизируется и регламентируется процесс разработки ПО. Причины улучшения документируются, новые проекты реализуются на основе более зрелых процессов. Широко внедряются всевозможные учебные программы. Роль отдельных личностей перестает влиять на результат.

IV.Уровень Managed (регулированный) – процессы оценены по множеству критериев, максимально документированы и легко управляемые. На первом плане эффективное управление, благодаря чему повышается качество продуктов, и понижается требование к ресурсам.

V.Уровень Optimized (оптимизированный) – постоянное улучшение процесса разработки, повышение эффективности. Для каждого процесса определены сильные и слабые стороны и наиболее подходящие области применения.

## Rup (Rational Unified Process)

**RUP** – унифицированный проект разработки – процесс разработки от Rational Software. Создатели: Гради Буг; Ивар Якобсон; Джеймс Рамбо. Создание ПО происходит в несколько итераций в конце которых получается рабочая версия продукта, но не с полным функционалом.

RUP объединяет многие из лучших методов разработки современного программного обеспечения, в частности, RUP следует следующим советам разработки:

1. Разрабатывайте итеративно
2. Управляйте требованиями
3. Пользуйтесь модульными архитектурами
4. Используйте визуальное моделирование
5. Не забывайте о проверке качества
6. Следите за изменениями

**Структура RUP (процесс имеет 4 фазы):**

1) Исследование (Inception).
2) Уточнение плана (Elaboration).
3) Построение (Construction).
4) Развертывание (Transition).

**На каждой из фаз основное внимание уделяется разным процессам:**

1) Фаза – сбор и анализ требований.
2) Фаза – анализ требований и проектирование.
3) Фаза – разработка и кодирование.
4) Фаза – тестирование и распространение.

**Методология разработана на 9 основных потоках:**

1) Бизнес-анализ (анализ потребностей).

2) Сбор требований и управление требований (перевод в функциональные спецификации).
3) Анализ и моделирование (перевод требований в программную модель).
4) Кодирование.
5) Тестирование (проверка соответствия требованиям).
6) Управление конфигурацией и изменениями (отслеживание изменений).
7) Управление проектом.
8) Создание и поддержка среды разработки.
9) Развертывание (все что нужно для продажи или передачи продукта).

**Принципы RUP:**

- Ранняя идентификация и непрерывное (до окончания проекта) устранение основных рисков.
- Концентрирование на выполнении основных требований заказчиков к исполняющей программе (анализ и посторенние модели прецедентов).
- Ожидание изменений в требованиях проектных решениях и реализации в процессе разработки.
- Компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта.
- Постоянное обеспечение качества на всех этапах разработки проекта (продукта).

Соотношение между фазами и потоками:

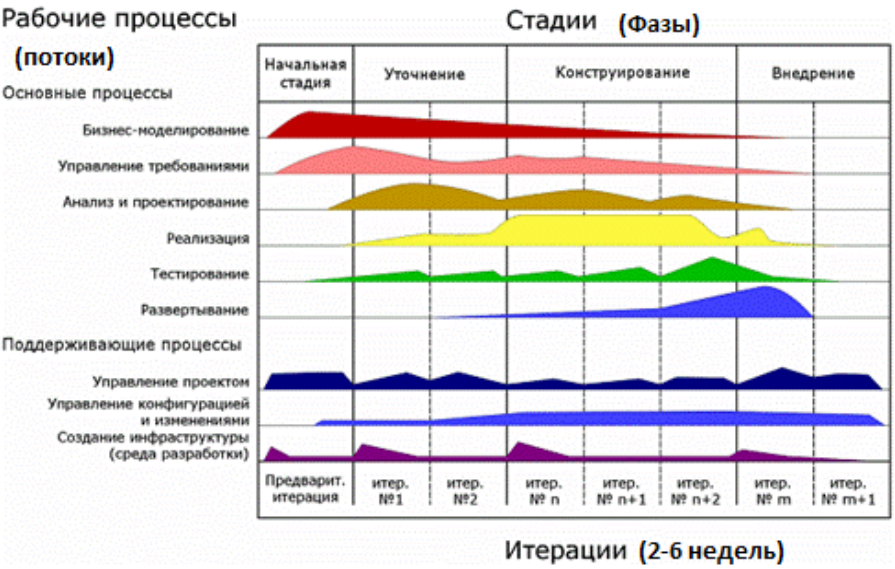



Рисунок 4.4. – Соотношение между фазами и потоками проекта.

Основным понятием процесса является исполнитель. Исполнитель определяет поведение и обязанности отдельных лиц или групп, работающих в одной команде. Поведение выражается через виды деятельности, производимые исполнителями, причем каждый исполнитель соотносится с рядом

связанных видов деятельности. Примеры исполнителей: *Системный аналитик*, *Разработчик*, *Разработчик тестов*.

*Вид деятельности* – это часть работы, выполнение которой может потребоваться от сотрудника и результат этой работы является значимым в контексте проекта. Вид деятельности имеет ясную цель, которая выражается, как правило, в создании или обновлении артефактов, таких как модель, план, класс. Примеры видов деятельности:

- *Планирование итерации*. Выполняет *Исполнитель: руководитель проекта*.
- *Рецензирование проекта*. Выполняет *Исполнитель: Рецензент проекта*.

Таким образом RUPинтерактивно отображается в двух элементах – динамическом (отображение времени выполнения и статическом (содержание). Результатом каждого из этапов является артефакт.

*Артефакт* – это "порция информации", порождаемая, модифицируемая или используемая в процессе. Артефакты – это вещественные продукты проекта: объекты, порождаемые или используемые проектом при работе над окончательным продуктом. Виды деятельности – это операции над активным объектом (исполнителем), артефакты – это параметры данных действий. Примеры артефактов: Проектная модель, План проекта, База данных требований к проекту.

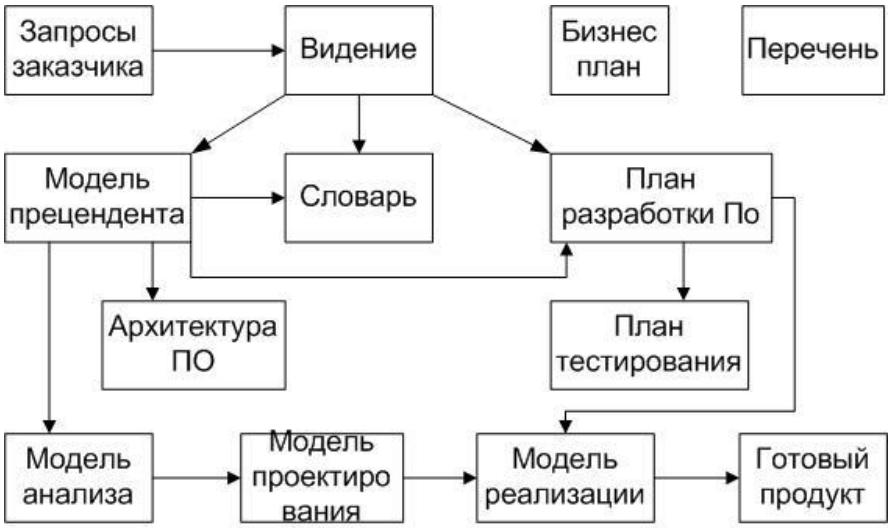



Рисунок 4.5 – Артефакты Rup

## MSF (MicrosoftSolutionFramework)

В методологии определены 6 ролевых кластеров, которые структурируют проектные функции разработчиков.

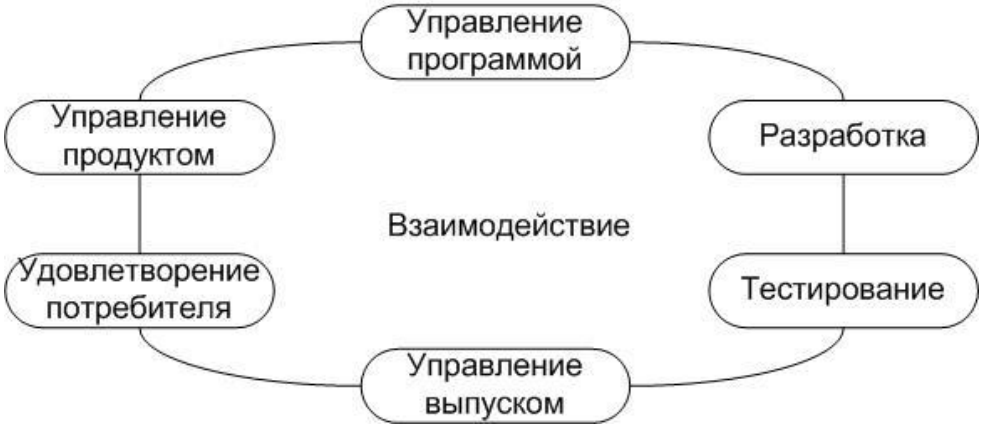



Рисунок 4.6 – Кластеры вMSF

- Управление продуктом – обеспечение интересов заказчика, содержит области компетенции:
  - планирование продукта;
  - планирование доходов;
  - представление интересов заказчика;
  - маркетинг.
- Управление программой – обеспечения реализации решений в рамках ограничений проекта (удовлетворение требований к бюджету и результату), области компетенции
  - управление проектом;
  - выработка архитектуры;
  - контроль производительного процесса;
  - административные службы.
- Разработка – построение решения в соответствии со спецификацией, области компетенции:
  - технологическое консультирование;
  - проектирование и осуществление реализации;
  - разработка приложений;
  - разработка инфраструктуры.
- Тестирование- одобрение выпуска продукта только после того, как все дефекты выявлены и устранены. Области компетенции:
  - разработка тестов;
  - отчетность о тестах;
  - планирование тестов;

- Удовлетворение потребителя – повышение эффективности использования продукта. Области компетенции:
  - интернацианализация (иноязычность);
  - обеспечение технической поддержки;
  - обучение пользователей;
  - удобство эксплуатации (эргономика);
  - графический дизайн.
- Управление выпуском – безпрепятственное внедрение и сопровождение продукта. Компетенции:
  - инфраструктура;
  - сопровождение;
  - бизнес-процессы;
  - управление выпуском готового продукта.

## Oracle CDM (Custom Development Method)

Основы CASE – технологии и инструментальной среды фирмы Oracle составляют:

1) методология нисходящего структурно проектирования – разработка прикладной системы в виде последовательности определенных этапов;
2) поддержка всех этапов жизненного цикла информационной системы;
3) ориентация на реализацию приложений в архитектуре «клиент-сервер» с использованием всех особенностей современных серверов (декларативные ограничения целостности, хранимые процедуры, триггеры баз данных и поддержку клиентской части всех стандартов и требований к графическому интерфейсу конечного пользователя);
4) наличие центральной базы данных (репозитория) для хранения спецификаций проекта прикладной системы на всех этапах её разработки (база специальной структуры, работающей под управлением СУБД Oracle);
5) возможность многопользовательского режима работы для согласованности работы и не допускает возможности, иногда каждый программист работает со своей версией и модифицирует её независимо от других;
6) автоматизация последовательного перехода от одного этапа к другому, что обеспечивается специальными утилитами, при помощи которых по спецификации концептуального уровня (модели предметного уровня) автоматически получают первоначальный вариант спецификации уровня проектирования ( описание баз данных и состава программных модулей). Этот процесс после уточнений автоматически генерирует следующий этап, вплоть до готовых программных модулей;
7) автоматизация стандартных действий по проектированию и реализации приложений, документированию, подготовки спецификацией.

**Методика выделяет следующие процессы:**

- определения производственных требований;
- исследование существующих систем;
- определения технической архитектуры;
- проектирование и построение баз данных;
- проектирование и реализация модулей;
- конвертирование данных;
- тестирование;
- обучение;
- переход к новой системе;
- подтверждение и сопровождение.

Особенности методологии:

- Три модели жизненного цикла:
  - классическая (предусматривает все этапы);
  - быстрая разработка (ориентация на использование инструментов моделирования и программирования Oracle);
  - облегченный поход (малые проекты и возможности протипировать приложения);
- не предусматривает включения дополнительных задач и привязку и к остальным;
- все модели каскадные;
- система программно-техническая (невозможность перейти к новой информационной системе);
- связано с инструментарием Oracle и плохо приспособлена к проектам где используются другие инструментальные средства.

## Open Source

Открытая ПО – программное обеспечение с открытым исходным кодом. Исходный код доступен для просмотра, изучения и изменения, возможности доработки самой программы, для создания нового ПО и исправления ошибок. Это происходит благодаря заимствованию исходного кода (если позволяет совместимость лицензий), через изучение использованных алгоритмов, структур данных, технологий, интерфейсов и методик.

**Некоторые лицензии:**

- Apache Software License (Веб-сервер)
- BSD License (Беркли, Unix подобные системы)
- Common Public License – CPL (IBM)
- Eclipse Public License

- GNU General Public License-GPL(автор передает ПО в общедоступную собственность, но сохраняет авторство)
- MIT License
- Mozilla Public License – MPL
- Nethack (компьютерная ролевая игра, благодаря лицензий портируема на все операционные системы)
- Nokia General Public License
- Open Software License (OSL) проект Debian лицензия не ограничивающая права копирования
- PHP License
- Python License
- Qt Public License – QPL
- Sun Public License (SPL)
- WTFPL (для свободного ПО, произведений культуры и научных работ)

## XP (ExternalProgramming)

XP – одна из гибких методологий применяется для создания программ коллективом от 10 до 32 человек при ранее заданных сроках сдачи.

**12 практик XP:**

- Короткий цикл обратной связи

1. Заказчик всегда рядом
2. Разработка через тестирование
3. Планирование
4. Парное программирование
   - Непрерывный, а не пакетный процесс
5. Непрерывная интеграция – автоматизированный процесс постоянные тесты
6. Рефакторинг – изменение исходного кода без изменения внешнего поведения для улучшения логичности и прозрачности кода.
   Методы рефакторинга:
   - Выделение метода.
   - Инкапсуляция поля, если у класса имеется открытое поле, то его делают закрытым и делают функции доступа get и set.
   - Перемещение метода.
7. Частые небольшие релизы
   - Понимание, разделяемое всеми
8. Метафора системы – краткое словесное описание
9. Коллективное владение кодом
10. Стандарт кодирования – набор правил и соглашений, используемых при написании исходного кода. Принимаются и используются для единообразного оформления.

- Способ выбора идентификатора.
- Использование регистра букв.
- Стиль отступов
- Способ расстановки скобок, ограничивающих логические блоки
- Использование пробелов для оформления логических и арифметических выражений
- Стиль комментариев

11. Простота при проектировании
    - Использование user stories
12. Социальная защищенность программы

## SCRUM

Scrum – это набор принципов, на которых строиться процесс разработки, позволяющий в жестко фиксированные и небольшие по времени итерации (называемые спринтами) представлять конечному пользователю работающее ПО с новыми возможностями, для которых определен наибольший приоритет.

**Три основные роли:**

1) ProjectOwner – отвечает за видение проекта и приоритеты реализации.
2) Team – (команда) отвечает за реализацию и качество продукта.
3) ScrumMaster – устраняет препятствия в работе и руководит процессом.

**Девять практик Scrum:**

- ScrumMaster – член команды, проводит ежедневные митинги.
- ProductOwner – единая точка окончательных решений в проекте, формирует требования пользователя, отвечает за приему результатов на каждой итерации и дает задачи всей команде, а не ее членам.
- Команда – самоорганизующийся коллектив $7\pm2$ (чило Миллера – объем кратковременной памяти человека).

  Основные принципы в команде:

- Кросс-функциональность (возможность членов команды делать некоторую часть работы вне своей основной зоны компетентности).
- Сидят в одном помещении

- Итерация (спринт sprint) – 1-4 недели состоит из этапов:

- Сбор требований.
- Проектирование.
- Разработка.
- Тестирование.

- План итераций (taskboard) – на доске приоретизированный список функций системы, каждая задача имеет одного ответственного исполнителя.
- В течении одного sprint на доску ничего добавлять нельзя.

– Доска

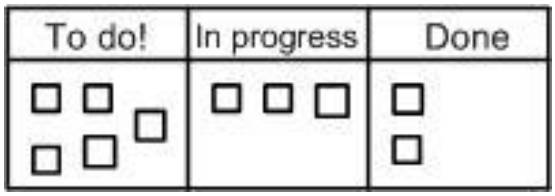



Рисунок 4.7 – Доска зщадач


Рисунок 4.8 – Диаграмма сгорания

– Ежедневный митинг – предназначен для организационной работы (не более 15 минут)

– Что сделано вчера.

– Что будет сделано сегодня.

– Какие проблемы.

– Демо. В конце каждой итерации создаются revue и показывают, что сделано.

## Kanban

Kanban – система проектирования, организации производства и снабжения, позволяющие реализовать принципы «точно в срок».

**3 практики:**

1. Визуализируйте производство:
   - Разделить работу на задачи (нанесенные на карточки)
   - Используйте названные столбцы, чтобы показать положение задачи в производстве.
2. Ограничивайте WIP (workingprogress – работу, выполняемую одновременно) на каждом этапе производства.
3. Измеряйте время цикла (среднее время выполнения задачи) и оптимизируйте постоянно процесс, чтобы уменьшить время.

Таблица *4.1 –Доска Kanban*

| Цели проекта | Очередь задач | Проработка дизайна | Разработка | Тестирование | Деплоймент | Закончено |
|---|---|---|---|---|---|---|
| (Необязательно) Высокоуровневые цели проекта, чтобы команда видела например: «увеличить скорость работы» | Хранятся задачи, готовые к выполнению. Для выполнения берется верхняя карточка. | Задачи, для которых дизайн кода или интерфейса еще не ясен и обсуждается. | Задача, находится до конца ее разработки. Можно вернуть в предыдущий столбец, если есть вопросы. | Пока задача тестируется. Если найдены ошибки, возвращается в разработку. | Т.е. направлен выполнить новую версию. | Когда все работы по задаче закончены полностью. |

## FDD (Featuredrivendevelopment)

Интерактивная методология разработки ПО, принимающая за основу важную для заказчика функциональность (свойства) разрабатываемого ПО. Основой целью является разработка реального, работающего ПО систематически, в поставленные сроки.

**Включает 5 базовых видов деятельности:**

Начало проекта

1) Разработка общей модели
2) Составления списка необходимых функций системы

Для каждой функции:

3) Планирование работы над каждой функции
4) Проектирование функций
5) Реализация функций

Разработки делятся на «хозяев классов» и «главных программистов». Главные привлекают хозяев для работы над очередным свойством. Работа над проектом предполагает частые сборки и делится на итерации, каждая из которых предполагает реализацию определенного набора функций.

**Набор практик:**

- Объектное моделирование области – исследование и выяснение рамок предметной области решаемой задачи. Результатом является общий каркас, который можно в дальнейшем дополнить функциями.
- Разработка по функциям – любая сложная функция разбивается на меньшие подфункции до тех пор пока каждая разбитая подзадача не

может быть названа свойством (т.е. реализована за 2 недели). Это облегчает создание корректно-работающих функций расширяющих и модифицирующих систему.

- Индивидуальное владение классом (кодом) – каждый блок закреплен за конкретным владельцем-разработчиком. Владелец ответственный за согласованность, производительность и концептуальную целостность своих классов.
- Команда по разработке функций (свойств) – маленькая динамическая формируемая команда разработчиков, занимающаяся подзадачей. Позволяет нескольким разработчикам участвовать в дизайне свойства, а также оценивать дизайнерские решения перед выбором наилучшего.
- Проверка кода – обеспечение качества кода путем выявления ошибок.
- Конфигурационное управление – помогает с идентификацией исходного кода для всех функций (свойств), разработка которых завершена на данный момент, и с протоколированием изменений, сделанных разработчиками классов.
- Регулярная сборка – гарантирует, что есть весь продукт (система), разработка которых завершена на текущий момент, и с протоколированных изменений, сделанных разработками классами.
- Обозримость хода работ и результатов – частые точные отчеты о ходе выполнения работы на всех уровнях внутри и за пределами проекта о выполненной работе помогает менеджерам правильно руководить проектом.

# ТЕМА 5. UML (UNIFIEDMODELLINGLANGUAGE)

UML – язык для определения, представления и проектирования и документирования программных систем, организационно –экономических систем и других систем различной природы.

Создан в конце 1994 г. Гради Буг и Джеймс Рамбо как объединение их методов Booch, OMT (Object Modelling Technique)

**Структура языка:**

- Сущности: структурные, поведенческие, группирированные, аннотационные.
- Отношения: зависимые, ассоциации, реализации, обобщения.
- Диаграммы: прецедентов, классов, объектов, последовательнстей, состояний, деятельности, кооперацийй, компонентов, размещения.

**Точки зрения диаграмм языка UML**

- Use case view (видение прецедентов) – взгляд на систему, независящий от ее реализации.
- Logical view (логическое представление) – поведение системы, независимо от ее реализации.
- Component view (представление компонентов) – с точки зрения иерархии библиотеки кодов, исполняемых файлов и других компонентов системы.
- Deployment view (представление размещения) – с точки зрения физического размещения модулей программы на этапах вычислительной техники.

**Инструменты:**

- Rational Rose (IBM)
- Visual Studio  начиная с 2010 версии
- Enterprise Architect
- MS Visio
- Umbulla
- Poseidon
- Visual Paradigm
- Интернет ресурсы, использующие принципы UML

**Назначение UML:**

1. Визуализация
2. Конструирование
3. Специфицирование
4. Документирование

Стандарт UML предлагает следующий набор диаграмм (версия стандарта 1.1. принята в 1997г.)

- Диаграммы вариантов использования (usecasediagrams) – для моделирования бизнес-процессов организации (требования к системе)
- Диаграммы классов (classdiagrams) – для моделирования статической структуры классов системы и связей между ними.
- Диаграммы поведения системы (behaviordiagrams):
    - Диаграммы взаимодействия (interactiondiagrams)
        - Диаграммы последовательности (sequence diagrams)
        - Диаграммы кооперации (collaborationdiagrams) для моделирования процесса обмена сообщениями между объектами.
    - Диаграммы состояний (statechartdiagrams) – для моделирования поведения объектов системы при переходе из одного состояния в другое
    - Диаграммы деятельностей (activitydiagrams) - для моделирования поведения объектов системы в рамках различных вариантов использования, или моделирования деятельностей
- Диаграммы реализаций (implementationdiagrams)
    - Диаграммы компонентов (componentdiagrams) – для моделирования иерархии компонентов (подсистем) системы.
    - Диаграммы размещения (deploymentdiagrams) –для моделирования физической архитектуры системы.

## Диаграммы вариантов использования (диаграммы прецедентов)

Отображают требования к разрабатываемой системе

**Прецедент (вариант использования)** – последовательность действий (транзакций), выполняемых системой в ответ на события, инициируемое некоторым внешним объектом(действующим лицом).Описывает взаимодействие между пользователем и системой.

Варианты использования - это не зависящее от реализации высокоуровневое представление о том, что пользователь ожидает от системы. Рассмотрим каждый фрагмент этого определения по отдельности.

Прежде всего, варианты использования не зависят от реализации. Представьте себе, что вы пишете руководство по работе с системой. Надо, чтобы ваши варианты использования можно было реализовать на языках Java, C++, Visual Basic или на бумаге. Варианты Использования заостряют внимание на том, что должна делать система, а не на том, как она должна будет делать это.

Далее, варианты использования - это высокоуровневое представление системы. Если, например, в модели содержится 3000 вариантов использования, вы потеряете преимущество простоты. Создаваемый набор вариантов использования должен дать пользователям возможность легко увидеть всю систему целиком на самом высоком уровне. Поэтому вариантов использова-

ния не должно быть слишком много, чтобы клиенту не пришлось долго блуждать по страницам документации, пытаясь понять, что будет делать система. В то же время, вариантов использования должно быть достаточно для того, чтобы полностью описать действия системы. Модель типичной системы обычно содержит от 20 до 50 вариантов использования. Для того, чтобы при необходимости разбить варианты использования на части, можно использовать связи различных типов, так называемые связи использования и расширения. Для лучшей организации системы можно также формировать группы вариантов использования, объединяя их в пакеты.

Наконец, варианты использования должны заострять внимание на том, что пользователи должны получить от системы. Каждый вариант использования должен представлять собой завершенную транзакцию между пользователем и системой, представляющую для первого некоторую ценность. Названия вариантов использования должны быть деловыми, а не техническими терминами, имеющими значение для заказчика. В рассматриваемом' нами примере АТМ нельзя назвать вариант использования "Интерфейс с банковской системой, осуществляющий перевод денег с кредитной карточки и наоборот". Вместо этого лучше назвать вариант использования "Оплатить по карточке" — так будет понятнее для заказчика. Варианты использования обычно называют глаголами или глагольными фразами, описывая при этом, что пользователь видит как конечный результат процесса. Его не интересует, сколько других систем задействованы в варианты использования, какие конкретные шаги надо предпринять и сколько строчек кода надо написать, чтобы заплатить по счету карточкой Visa. Для него важно только, чтобы оплата была сделана. Еще раз повторим, что нужно заострить внимание на результате, который потребитель ожидает от системы, а не на действиях, которые надо предпринять для достижения этого результата.

**Действующее лицо (actor)** – это роль, которую пользователь играет по отношению к системе. Представляет собой роли, а не конкретных людей или наименования работ. Делятся на типы:

I тип – это физические личности, или пользователи системы.

II тип – внешняя система. Сделав систему действующем лицом, или предполагаем, что она не будет изменяться вообще. Действующие лица находятся вне сферы действия того, что мы разрабатываем и, таким образом, не подлежит контролю с нашей стороны.

III тип – время. Время становится действующим лицом, если от него запуск какие-либо событий в системе.

**Отношение** – взаимодействия между действующим лицом и прецедентом. Существует несколько типов связей: связь коммуникации (communication), использования(uses), расширения (extends) и обобщения действующего лица(actorgeneralization)


Рисунок 5.1 – Пример диаграмма вариантов

**Связь коммуникации** – это связь между вариантом использования и действующим лицом. Направление стрелки позволяет понять кто инициирует коммуникацию.

**Связь использования** – позволяет одному варианту использования использовать функциональность другого (моделирование многократно используемой функции) связь полагает, что, что один вариант использования всегда использует функциональные возможности другого.

**Связь расширения** позволяет только при необходимости использовать функциональные возможности, предоставляемые другим вариантом использования.

**Связь обобщения** – показывает, что у нескольких действующих лиц имеются общие черты.

Поток событий включает:

- Краткое описание – описание что вариант использования должен делать
- Предусловия – условия, которые должны быть выполнены, прежде чем вариант использования начнет выполнятся сам.
- Основной поток событий
- Альтернативный поток событий (поэтапное описание, что должно происходить)
- Постусловия – всегда должны быть выполнены после завершения вариантов использования.

Основной и альтернативный потоки событий включают следующие описания:

- Каким образом запускается вариант использования.
- Различные пути выполнения варианта использования.
- Нормальный или основной поток событий.
- Отключения от основного потока событий (так называемые альтернативные потоки).
- Поток ошибок.

- Каким образом завершается вариант использования.

*Основной поток*

1. Вариант использования начинается, когда клиент вставляет свою карточку в банкомат
2. Банкомат выводит приветствие и предлагает клиенту ввести свой персональный идентификационный номер
3. Клиент вводит номер
4. Банкомат подтверждает введенный номер. Если номер не подтверждается, выполняется альтернативный поток событий А1.
5. Банкомат выводит список доступных действий
    - положить деньги на счет
    - снять деньги со счета
    - перевести деньги
6. Клиент выбирает пункт «Снять деньги»
7. Банкомат запрашивает сумму
8. Клиент вводит требуемую сумму
9. Банкомат определяет, имеется ли на счету достаточно денег. Если денег недостаточно, выполняется альтернативный поток событий А2. Если во время подтверждения суммы возникли ошибки, выполняется поток ошибок Е1.
10. Банкомат вычитает требуемую сумму из счета клиента.
11. Банкомат выдает клиенту требуемую сумму наличными.
12. Банкомат возвращает клиенту его карточку.
13. Банкомат печатает чек для клиента
14. Вариант использования завершается.

*Альтернативный поток А1. Ввод неправильного идентификационного номера:*

1. Банкомат информирует клиента, что идентификационный номер введен не правильно.
2. Банкомат возвращает клиенту его карточку.
3. Вариант использования завершается.

*Альтернативный вариант использования А2. Недостаточно денег на счету:*

1. Банкомат информирует клиента, что денег на счету недостаточно.
2. Банкомат возвращает клиенту его карточку.
3. Вариант использования завершается.

*Поток ошибок Е1. Ошибка в подтверждении запрашиваемой суммы:*

1. Банкомат сообщает пользователю, что при подтверждении запрашиваемой суммы произошла ошибка и дает номер телефона службы поддержки.
2. Банкомат заносит сведения об ошибке в журнал ошибок.
3. Банкомат возвращает клиенту его карточку.
4. Вариант использования завершается.

Основные правила для создания диаграмм вариантов использования:

1. Количество вариантов использования должно стремится к числу Миллера (7±2)
2. Варианты использования должны быть использованы с точки зрения действующих лиц, а не с точки зрения системы
3. Границы системы должны быть четно определены
4. Избегать многочисленных запутанных связей между элементам.
5. Пользователь должен четко понимать варианты использования.

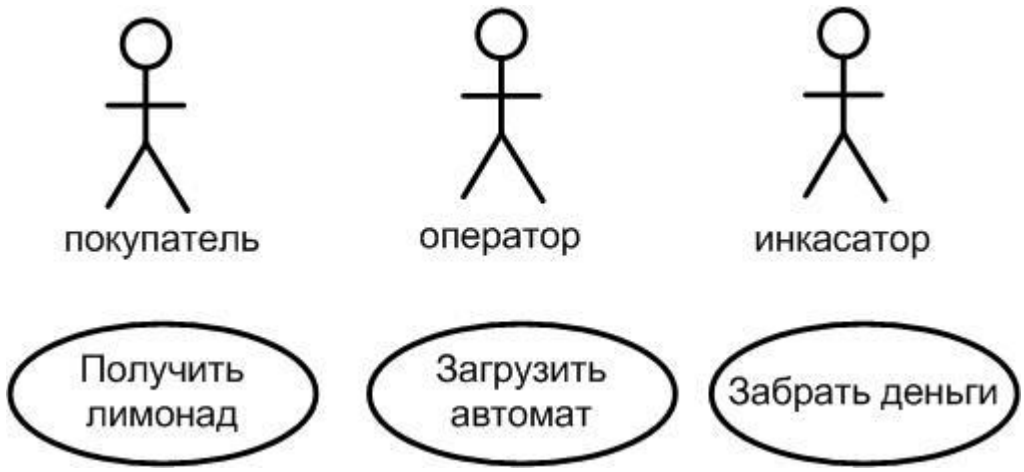



Рисунок 5.2 – Пример автомат для покупки лимонада

Потоки событий

1. Покупатель может купить лимонад
2. Наличие денег, жажды, лимонада
3. Последовательность действий
   1) Бросить деньги
   2) Выбрать сорт лимонада
   3) Автомат выдает стакан
   4) Выдача сдачи
4. А1: отсутствие лимонада. Переход к пункту 2 основного потока, иногда выдать деньги.
   А2: отсутствие сдачи. Если сдачи нет переход к пункту 2. Если снова нет сдачи – выдать деньги
5. Наличие стакана в руках у покупателя

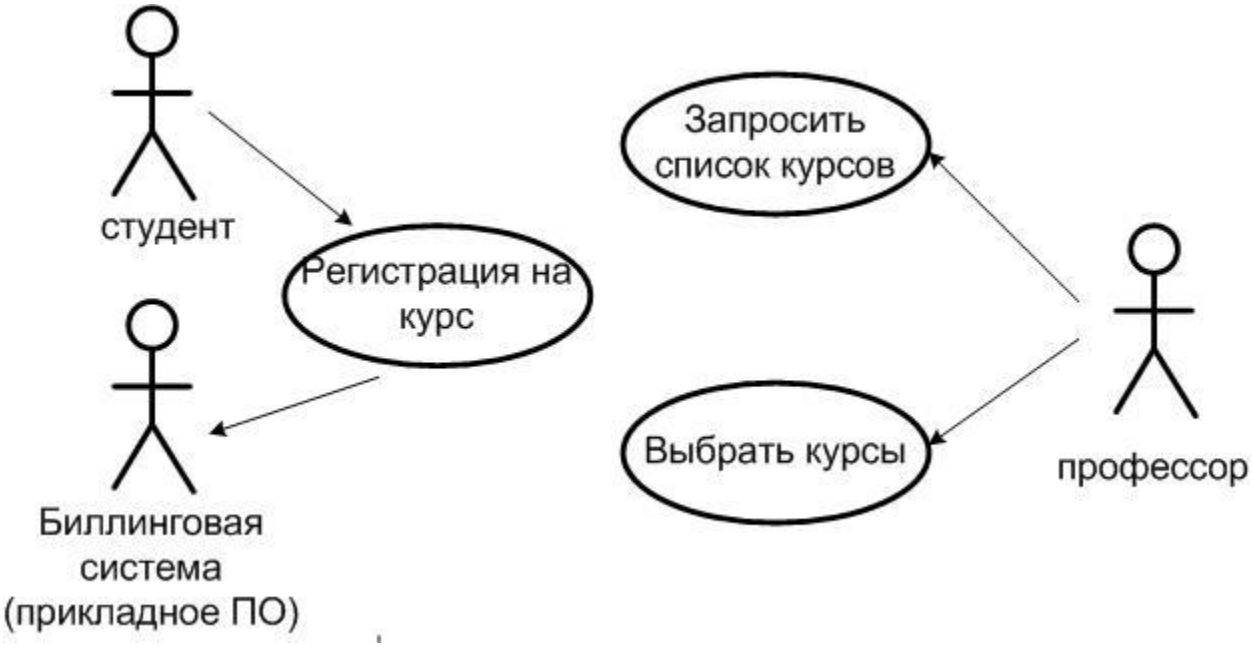



Рисунок 5.3 – Пример регистрация студентов на курсы

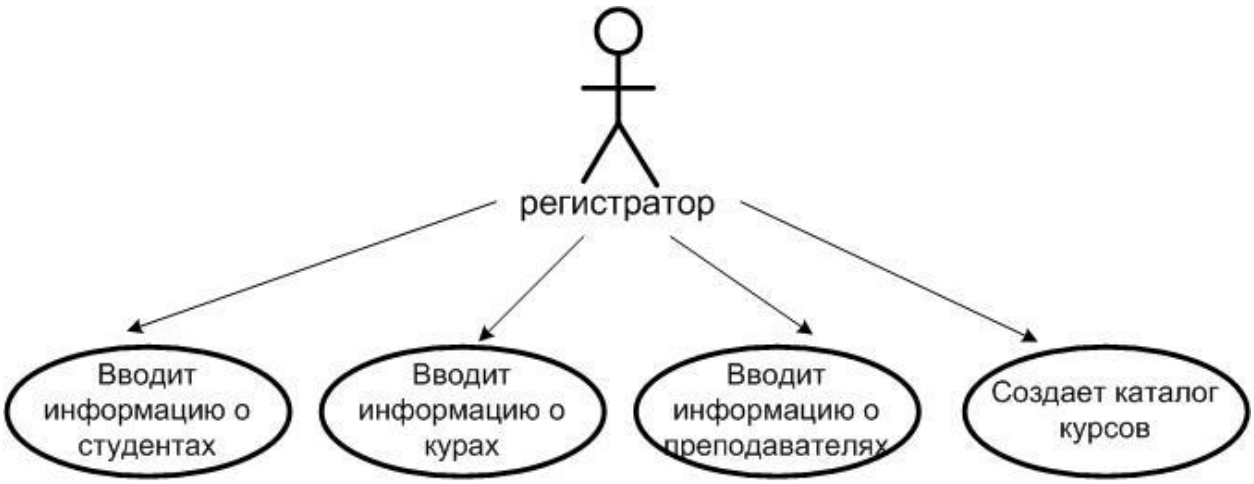



Рисунок 5.4 – Пример регистрация студентов на курсы

*Опишем поток «выбрать курс»*

1. Профессор выбирает курсы для преподавателей
2. Перед началом выполнения должны выполнить поток – ввод информации о курсах (из usecases)
3. Основной поток
   1) Вариант использования начинает выполняется, когда профессор входит в систему и вводит пароль, система проверяет пароль (E1) и предлагает выбрать текущий семестр(E2). После этого система предлагает профессору выбрать действия из ADD, DELETE, REVIEW, PRINT, QUIT

ADD – S-1: добавить предложение по курсу
DELETE –S-2: удалить
….
QUIT – вариант использования завершается

S-1: добавить предложение по курсу. Система выводит экран курса, содержащие поля для ввода номера и названия курса. Профессор вводит номер и название курса (E-3) система одобряет предложение по курсу (E-4). Профессор выбирает(E-5) некоторое предложение – система устанавливает связь между профессором и выбранным предположением (E-6). Выполнение варианта: переход в начало.

4. E-1 Профессор ввел неправильный идентификационный пароль, выдается сообщение об ошибке. Можно завершить работу или повторить ввести пароль.
   E-2, E-3 – введен неверный семестр – повторить или завершить.
   E-4 – предложение по курсу не может быть отображено – профессора информируют о невозможности выполнить. Переход в начало.
   E-6 – связь между профессором и предложением по курсу не может быть установлена – информация сохраняется, чтобы связь была создана позднее.

**Диаграмма взаимодействия**

Диаграммы взаимодействия описывают поведение взаимодействующих групп объектов. Они охватывают взаимодействия в рамках одного варианта использования.

Диаграммы взаимодействия

| Диаграммы последовательности (sequence) направлены на временное отображение взаимодействия. | Диаграммы кооперации (collaboration)направлены на объектное отображение взаимодействия. |
|---|---|

1) **Сообщение** (message) основной элемент, средство с помощью которого объект отправитель запрашивает у объекта получателя выполнение одной из его операций.

Виды сообщений:

- Информационные сообщения(message) – снабжает объект-получатель некоторой информацией для обновления его состояния.
- Сообщение запрос (informative) – запрашивает у объекта получателя выполнение некоторого действия.
- Императивное сообщение (imperative) – запрашивает у объекта получателя выполнение некоторого действия.

*(маркер итерации) – сообщение повторяется n раз. {…} – ограничения.

**С точки зрения UML сообщения делятся:**


Требуют ответа от объекта получателя.


Вызывая объект, может продолжить свою работу. Не требуют ответа.


Сообщение, которое объект посылает сам себе.

**Фокус управления** – это начало и конец активности объекта во время взаимодействия.

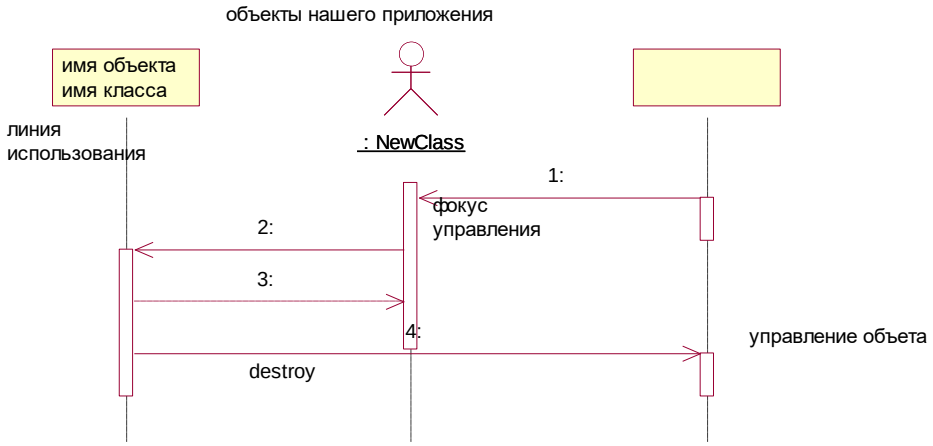



Рисунок 5.5 – Пример диаграммы последовательности действий

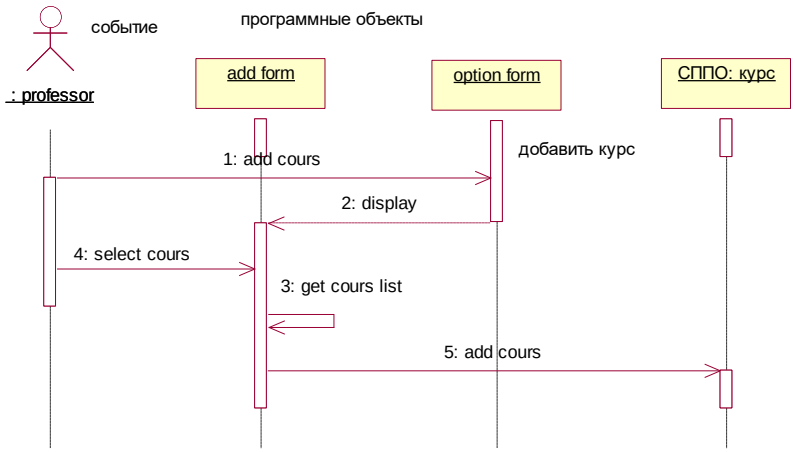



Рисунок 5.6 – Пример запись преподавателей на курсы

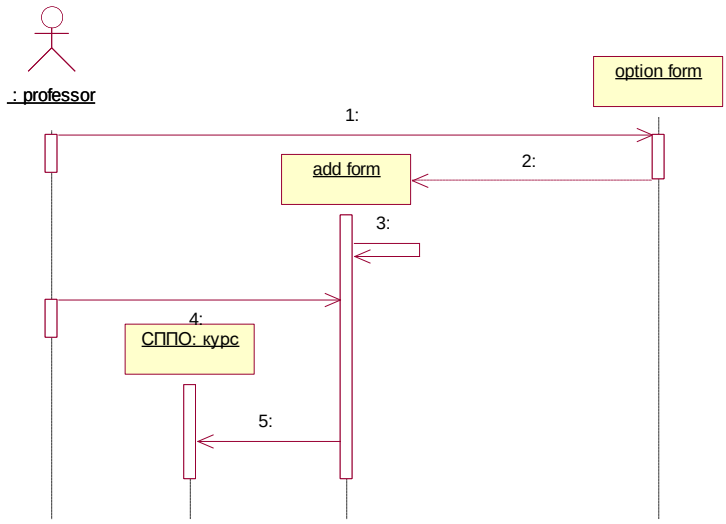



Рисунок 5.7 – Пример Запись преподавателей на курсы

**Типы объектов:**

- Граничные объекты (формы).
- Объекты бизнес-логики приложения (БД).
- Управляющие объекты.

2)**Диаграммы кооперации** – это граф вершинами которого являются объекты.

**Преимущества диаграмм взаимодействия**: позволяет определить классы, которые нужно создать, связи между ними, операции и ответственности каждого класса.

**Методы объектно-ориентированного анализа:**

1) Анализ неформального описания задачи
   Существительное – претенденты объектов и классов.
   Глагол – претенденты для методов класса
2) CRCкарточки (Class Responsibility Collaboration)


(класс – ответственный – взаимодействие)

3) Анализ поведения: в классы объединены объекты, которые имеют сходное поведение.

**Недостатки диаграмм взаимодействия:**

- Не отображают поведение объекта во многих вариантах использования (для этого используются диаграммы состояний).
- Не позволяет точно, алгоритмически описать поведение объектов (для этого предназначены диаграммы деятельности).

**Двухэтапный подход к разработке диаграмм взаимодействия:**

1. Этап: изображается информация высокого уровня и сообщение не соотноситься с методами.

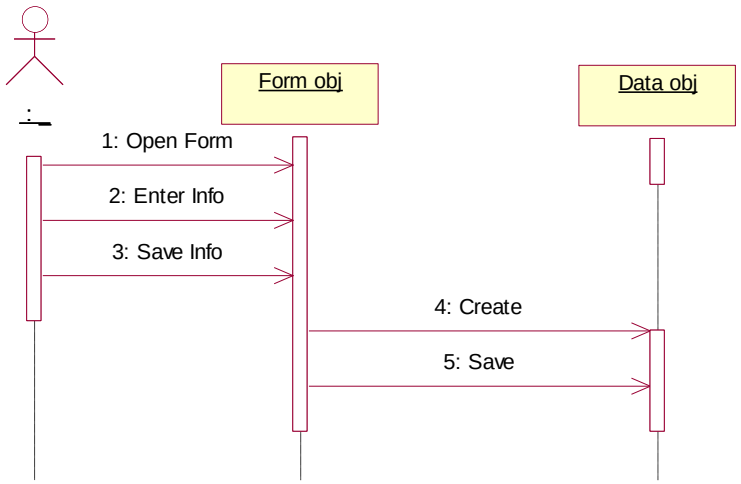



Рисунок 5.8 – Первый этап

2. Этап: происходит оптимизация диаграмм, и добавляются управляющие объекты.

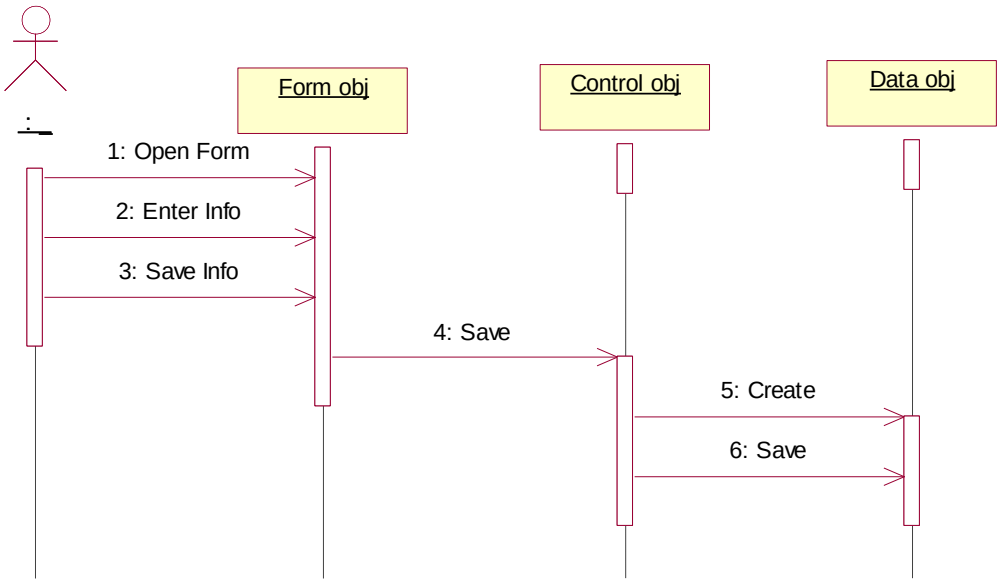



Рисунок 5.9 – Второй этап

Управляющие объекты отвечают за координацию других объектов и делегирование ответственностями. Основное преимущество – отделение бизнес-логики приложения от логики определяющей последовательность событий.

**Сравнение диаграмм последовательности и кооперативных диаграмм**

У разных разработчиков имеются различные предпочтения по поводу выбора вида диаграммы взаимодействия. В диаграмме последовательности делается акцент именно на последовательность сообщений: легче наблюдать порядок, в котором происходят различные события. На кооперативной диаграмме можно использовать пространственное расположение объектов для того, чтобы показать их статическое взаимодействие.

Одним из принципиальных свойств любой формы диаграммы взаимодействия является их простота. Посмотрев на диаграмму, можно легко увидеть все сообщения. Однако, если попытаться изобразить нечто более сложное, чем единственный последовательный процесс без особых условных переходов или циклов, то такой подход не сработает.

Диаграммы взаимодействия наиболее хороши, когда они отображают простое поведение; при более сложном поведении они быстро теряют свою ясность и наглядность. Если нужно показать сложное поведение системы на одной диаграмме, то следует использовать диаграмму деятельностей.

Диаграммы взаимодействия следует использовать, когда нужно описать поведение нескольких объектов в рамках одного варианта использования. Они хороши для отображения взаимодействия между объектами и вовсе не так хороши для точного описания их поведения.

Если нужно описать поведение единственного объекта во многих вариантах использования, то следует применить диаграмму состояний. Если же описывается поведение во многих вариантах использования или многих параллельных процессах, следует рассмотреть диаграмму деятельностей.

## Диаграммы классов

**Диаграммы классов** определяет типы классов и статические связи, которые существуют между ними.

**Классы** – это типы объектов, которые содержат данные и поведение, влияющее на эти данные.

| «стереотип»<br>имя_класса |
|---|
| атрибуты |
| методы |

Отображения классов в UML

**Диаграммы классов могут рассматриваться с 3-х точек зрения:**

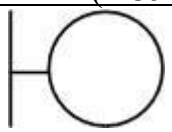
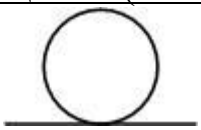
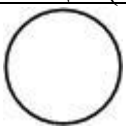
1. Концептуальная – рассматривается понятия предметной области и не зависит от средств реализации программного продукта.

2. Спецификация – рассматриваются интерфейсы, а не внутренней реализации классов
3. Реализация – определяет реализацию класса вашего программного продукта

**Механизм стереотипов**

Стереотип – это механизм, позволяющий разделять классы на категории, это упрощает навигацию по проекту и управления проектом.

**Стереотипы UML**

| Граничные (Boundary) | Сущности(Entity) | Управляющие(Control) |
|---|---|---|
|  |  |  |
| Классы, которые находятся на границе системы и окружающей среды (формы, интерфейсы, отчеты) | Содержит информацию сохраняемую в системе постоянно (хранят бизнес-логику) | Отвечают за координацию действий других классов (много сообщений посылается, мало получают) |

**Механизм пакетов**

Пакет – способ группировки классов в компоненты более высокого уровня. Позволяет уменьшить количество зависимостей между компонентами системы.

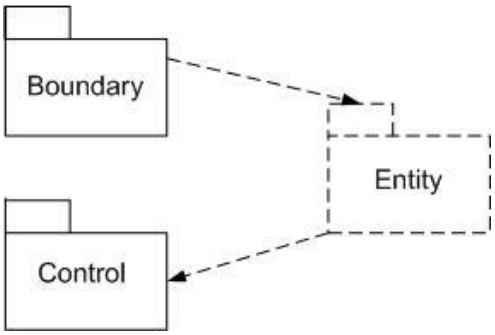



Рисунок 5.10 – Пакет

**Механизм видимости атрибутов**

Атрибут – это элемент информации, связанный с классом.

+ public (общий, открытый)
- privat (закрытый, секретный)
# protected (защищенный)
PackageorImplementation (пакетный) – общий, но в рамках этого пакета.

**Операции** (или методы) реализуют связанной с классом поведение Имя (аргумент 1: тип;…): тип возвращаемого значения.

В языке UML операции имеют следующую нотацию:

***Имя Операции (аргумент! : тип данных аргумента!, аргумент! : тип данных аргумента!,...) : тип возвращаемого значения***

**Бывают четырех типов:**

1) Операции реализации – реализация бизнес функций (каждое сообщение системы можно соотнести с операцией реализации)
2) Операции управления: управляют создание и уничтожением объектов
3) Операции доступа – обеспечивают доступ к закрытым и защищенным атрибутам (get и set)
4) Вспомогательные операции – закрытые и защищенные методы классов, необходимых, необходимых для выполнения его ответственностей.

| Работник |
|---|
| + ФИО<br>+ должность<br>- зарплата<br># нагрузка |
| |

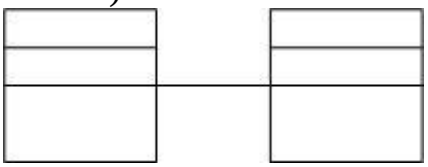
| Телевизор |
|---|
| +язык меню<br>- частота сигнала<br>+ порядок каналов |
| + вкл; + выкл<br>- декодированный сигнал<br>- самодиагностика |

**Связи между классами**

Отношения – это семантическая связь между классами. Дает возможность классу узнавать об атрибутах, операциях и связях другого класса. Чтобы класс мог послать сообщение на диаграмме классов между ними должна существовать связь.

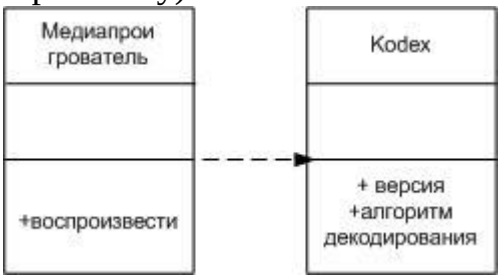
**Типы сообщений (отношений):**

1. Связь ассоциации (association)



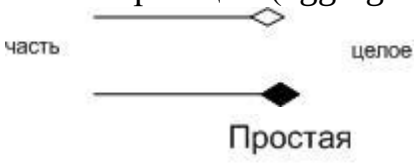
Если классы связанны, то они могут передавать сообщения
→ однонаправленные
– двунаправленные

2. Связь зависимости (dependency)

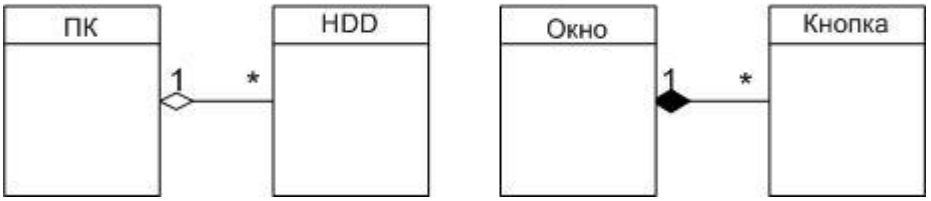


Показывает, что один класс зависит от определений, сделанных в другом классе

3. Связь агрегации (aggregations)

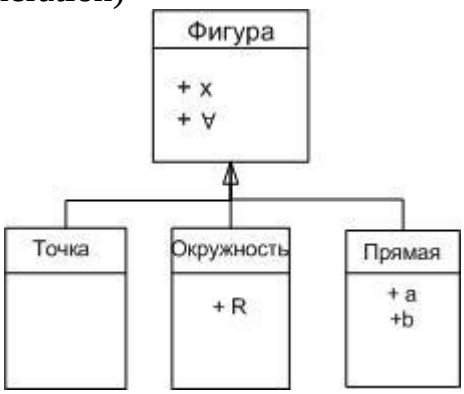



Это связь между целым и его частью



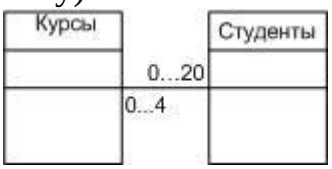

Композиция агрегирования – отображает ситуацию, когда целое владеет частью и их время жизни совпадает.

4. Связь обобщения (generation)




Показывает связи наследования между классами

5. Множественность (multiplicity)




Показывает столько экземпляров одного класса с экземпляром другого класса в данный момент.

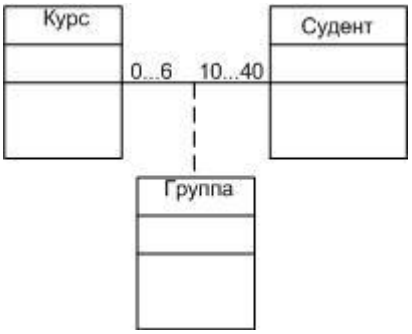
**Дополнительные атрибуты связей**

1) Имена связей


Связи можно уточнять с помощью имен связей или ролевых имен (необязательно)

2) Роли


3) Классы ассоциаций




Создаются в случае громоздкости связи декомпозиции (задается для хранения дополнительной информации о классе, для того, чтобы не загромождать)

Пример. Сказка о Курочке Рябе.

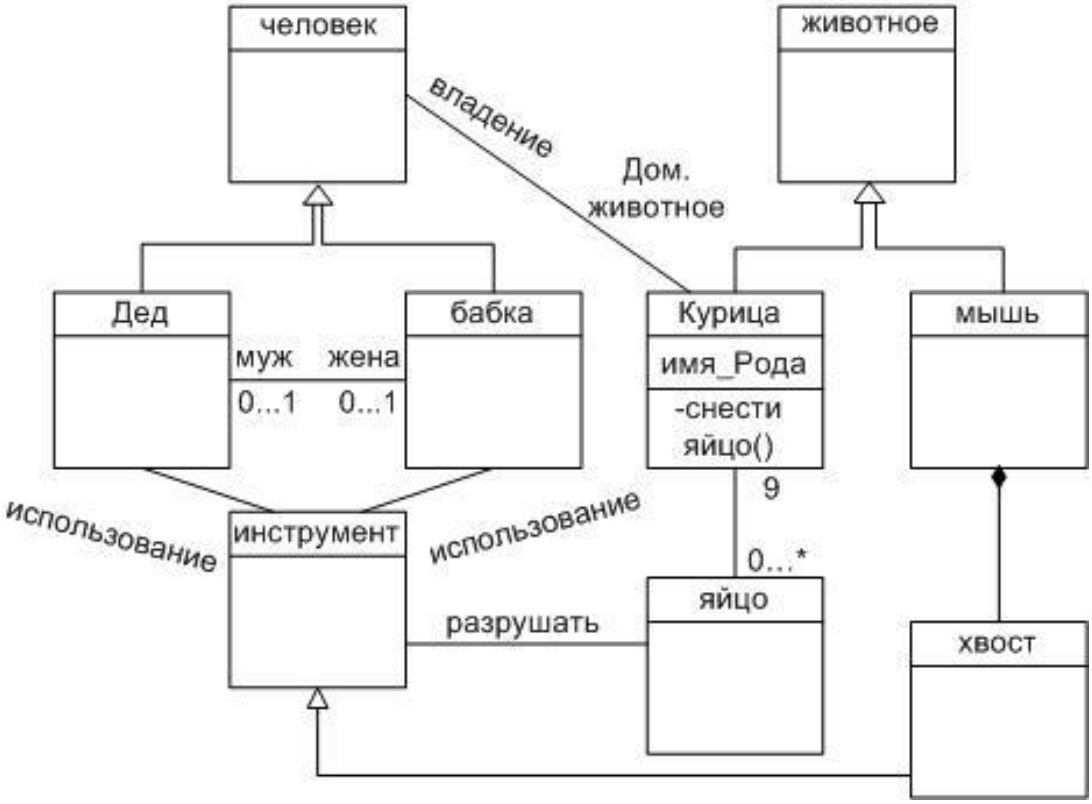



Рисунок 5.11 – Пример диаграммы классов

## Диаграммы состояний

Диаграмма состояний представляет собой конечный автомат и определяет все возможные состояния, в которых может находиться конкретный объект, а так же процесс смены состояний объекта в результате наступления некоторых событий.

Строиться для одного класса и отражают динамику поведения единого объекта во всех use case.

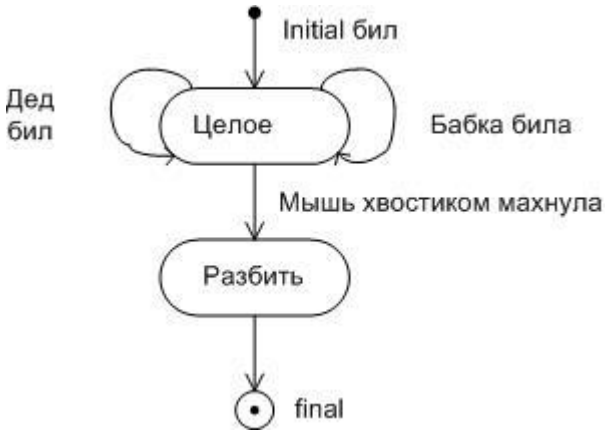


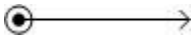

Рисунок 5.12 – диаграмма состояний

Диаграммы состояний применяется для динамического моделирования сложных объектов, необходимых для систем реального времени имитационного моделирования и показывает поток управления объектом в течение его жизненного цикла.

**Специальные состояния:**

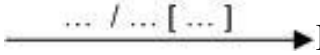
1) Начальное состояние (обязательное и должно быть одно)

2) Конечное состояние (количество от 0…п)



**Состояние** – это ситуация в жизненном цикле объекта, во время которого он удовлетворяет некоторому условию, выполняет какое-то действие или ожидает какого-то события.

1) Entry – это поведение, которое выполняется, когда объект переходит в новое состояние. Рассматриваетсякак часть перехода и направленное действие.
2) Do – просто действие, поведение, реализованное объектом, пока он находится в данном состоянии.
3) Exit – выходное действие, составная часть перехода, является непрерывным.



На строке перехода могут быть события / действие [guardcondition] ограждающие условия.

Событиемназывают переход из одного состояния в другое.

Действие – непрерываемое событие, осуществляемое объектом, как часть перехода.

Ограждающее условие(guardcondition) – указывает, когда переход может быть осуществлен.

**Типы переходов:**

1. Тригерный переход – определяется событием тригером, внешним по отношению к объекту.

Пример. Сигнализация.


Рисунок 5.13 – Пример сигнализация

2. Нетригерный переход наступает по завершению do – деятельности к объетам.

Пример.

3. Аргументный – если в течении выполнения действия появится сообщение для другого объекта. Это записывается следующим образом.

^ Цель. Событие (Аргументы)

^ - включена отправка сообщений другому объекту.

Цель – объект, получающий событие.

Событие – посылаемое сообщение.

Аргументы – параметры посылаемого сообщения.

Пример.

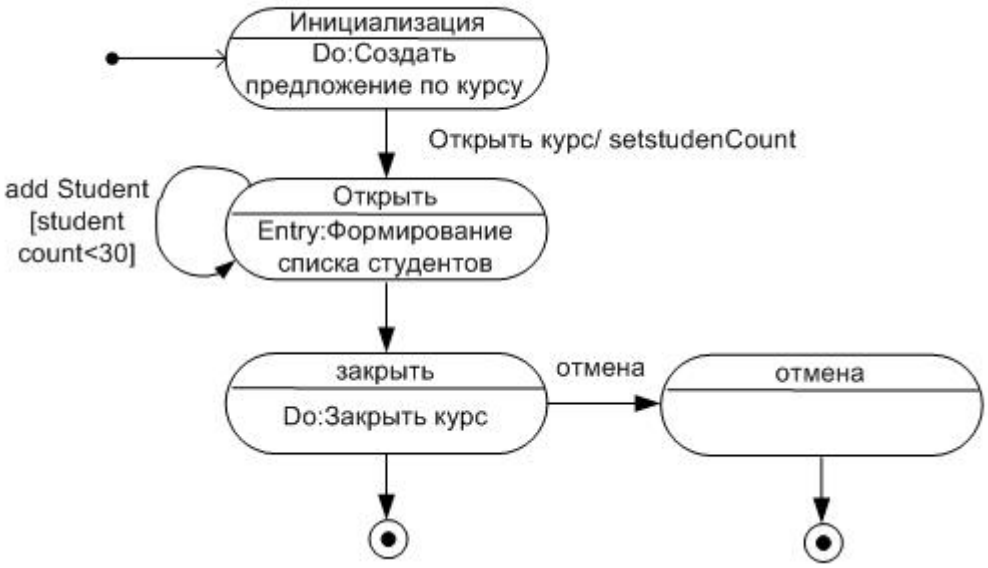



Рисунок 5.14 – Пример аргументного перехода

**Параллельное поведение объектов** в системе моделируется с помощью вложенных состояний.

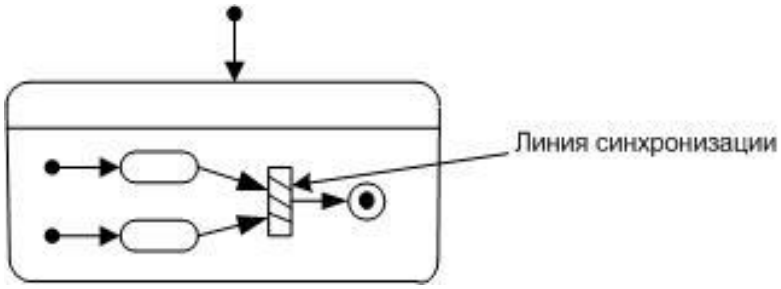



Рисунок 5.15 – пример описания параллельного поведения объекта

## Диаграммы деятельности

Диаграммы деятельности используются для отображения логики внутри методов, а так же для отображения деятельности во всех usecase.

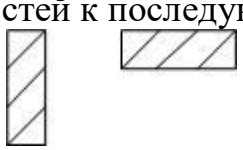
**Основные элементы:**

Входные состояния

Выходные состояния

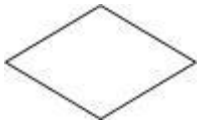
Деятельность

Линии синхронизации (аккумулируют переход от нескольких деятельностей к последующим).



Множественная активация цикла


Ветвление



Ограждающее условие


Пример

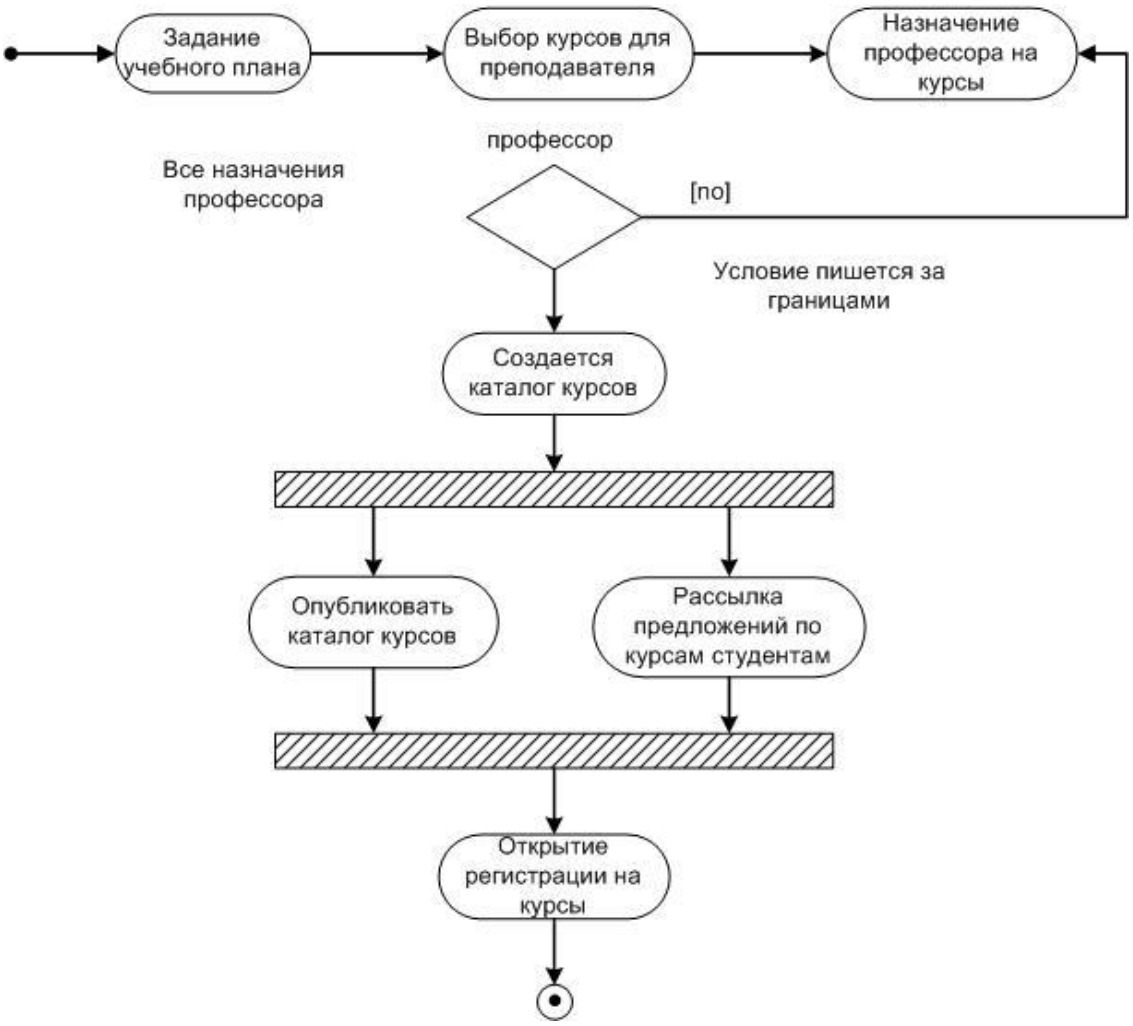



Рисунок 5.16 – Диаграммы деятельности

**Диаграмма компонентов**

Для создания конкретной физической системы необходимо реализовать все концептуальные логические аспекты всех рассмотренных моделей в конкретных физических сущностях (файлах, библиотеках и т.д.)

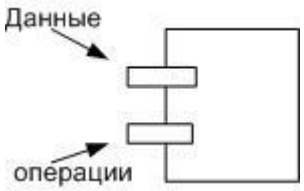
**Физическая система** – реально существующий прототип моделей системы.

**Диаграмма компонентов** – описание особенностей физического представления системы, определяет её архитектуру и устанавливает зависимость между программными компонентами.

**Компонент** – это физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а так же функциональное поведение моделируемой программной системы.

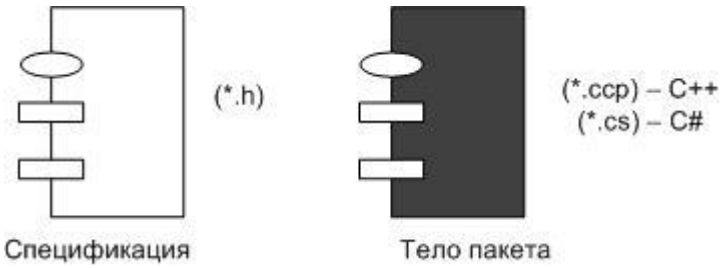
**Типы компонентов:**

1) Программный модуль

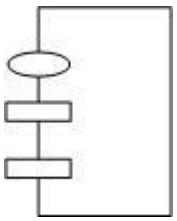



2) Подпрограммы

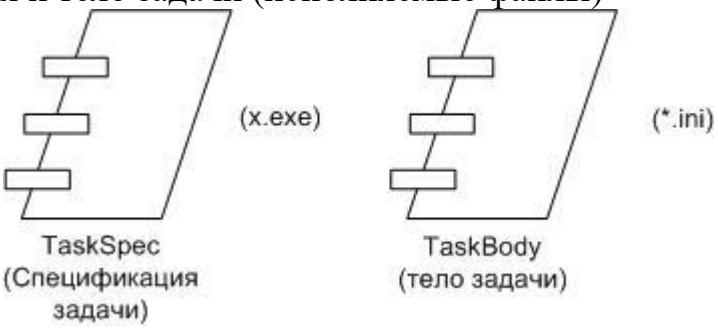

3) Спецификация и тело пакета




4) DDL – файлы



5) Спецификация и тело задачи (исполняемые файлы)




6) Связь – изображается только в виде зависимости

⟶ Направлены от зависимого к независимому

Пример:

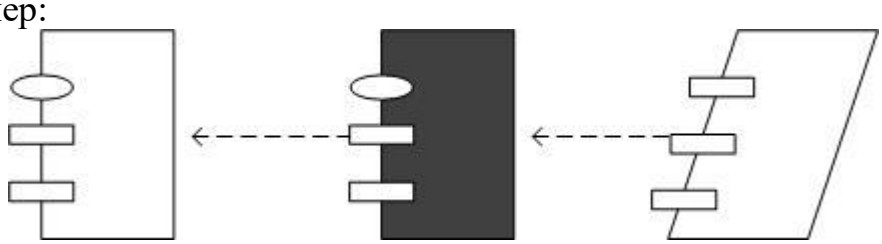


Рисунок 5.17 – диаграмма компонентов

**Зависимость позволяет:**

- Выделить компоненты, пригодные для повторного использования.
- Увидеть связи компиляции и генерации объектного кода.

**Прямое проектирование** – преобразование модели в программный код на каком-либо языке программирования.

**Обратное программирование** – (reverse engineering) преобразование программного кода в модель.

В Rational Rose есть средства для обратного проектирования, которые позволяют получать диаграммы классов, компонентов.

В Visual Studio начиная с версии 2010 тоже есть средства, позволяющие генерировать диаграммы классов.

**Генерация кода:**

1) Проверка модели (Tools→Check→model)
2) Отображение классов на компоненты
3) Выбор компонента и генерация кода
   Tools→Code→generation
   Найти сгенерированные файлы.
   C:\\Program Files\Ration Rouse\C++\Sourse\имя_проекта
   Пример: клиент ATM

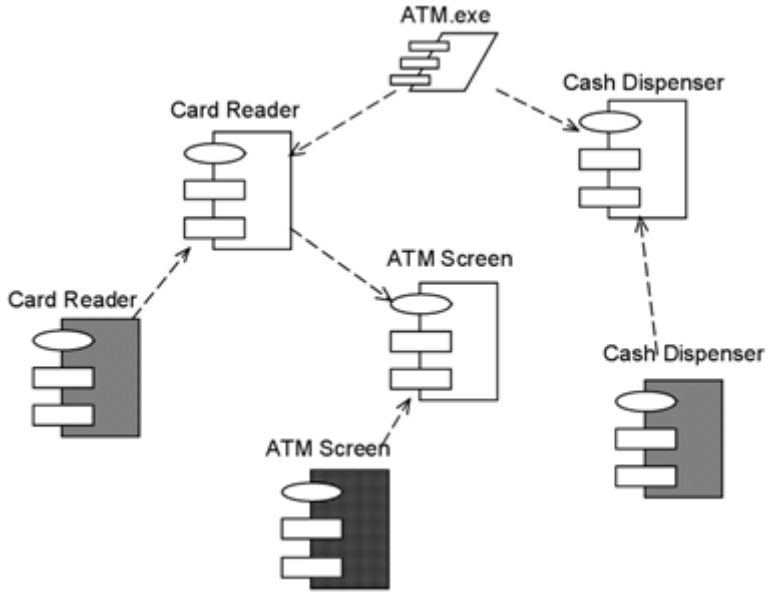



Рисунок 5.18 – Диаграмма компонентов

Класс ATM Screen преобразуется в компонент ATM Screen диаграммы. Он так же преобразуется и во второй компонент ATM Screen. Вместе два эти компонента представляют тело и заголовок класса ATM Screen. Выделенный темный компонент – спецификация пакета и соответствующий файлу тела ATM Screen на C++ (*.cpp). Невыделенный компонент тоже спецификация , но соответствующий заголовочному файлу класса C++(*.h). Компонент

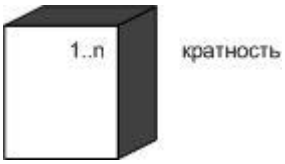
ATM.exe является спецификацией задачи и представляет поток обработки информации (исполняемая программа).

**Диаграммы размещения (диаграммы развертывания)**

Диаграммы размещения относятся к физическому проявлению системы и отображают конфигурацию узлов сети и размещенные на них компоненты.

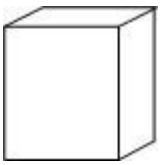
**Компоненты**

1) Процессоры




Это любое устройство, обладающее вычислительной мощностью, можно указать имя, стереотипы (КПК, ПК, сервер и т.д.) и кратность.

2) Узел



Это аппараты не обладающее вычислительной мощностью (принтер, терминал и т.д.)

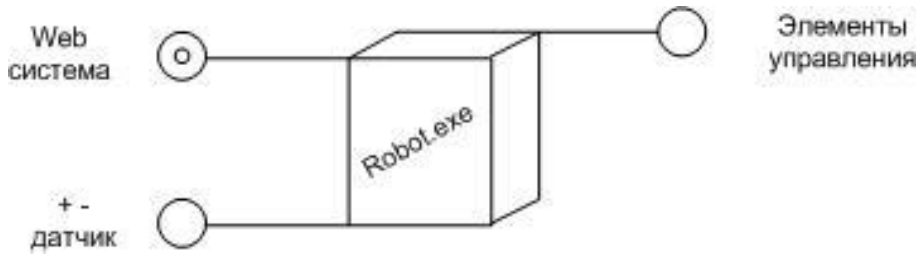
3) Связи



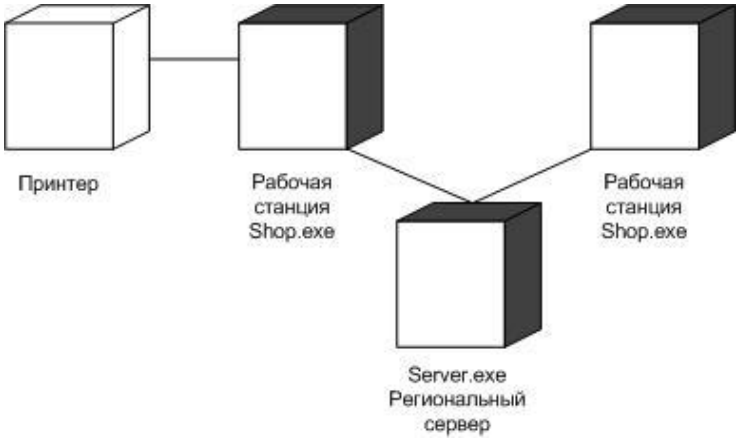
Показывают физические связи между двумя процессорами, устройствами. Наличие физической связи по обмену информацией между устройствами.

**Задачи диаграмм размещения:**

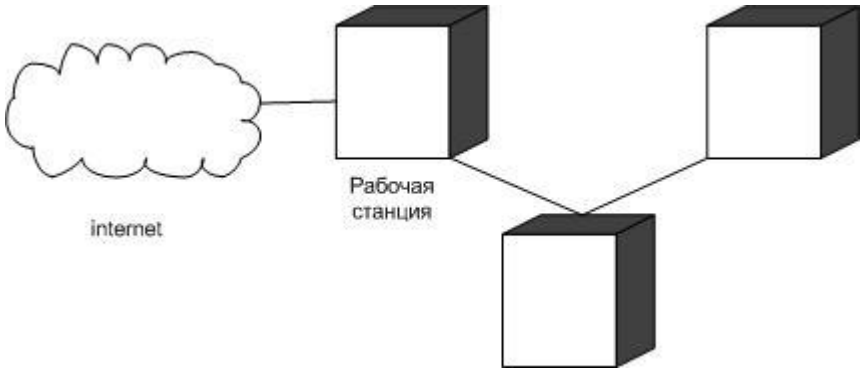
1) Отображение встраиваемых систем – когда программный продукт получает информацию с датчиков и отправляет информацию устройствам




2) Клиент серверные системы




3) Распределенные системы.



# ТЕМА 6. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

## Основы тестирования Программного Обеспечения

**Функции тестирования:**

1) Верификация – (от лат. Versus -истинный) – подтверждения соответствия конечного продукта определенным требованиям.
2) Валидация – (от лат. Validus - здравый) – гарантия, что продукт удовлетворяет требованиям заказчика.
3)

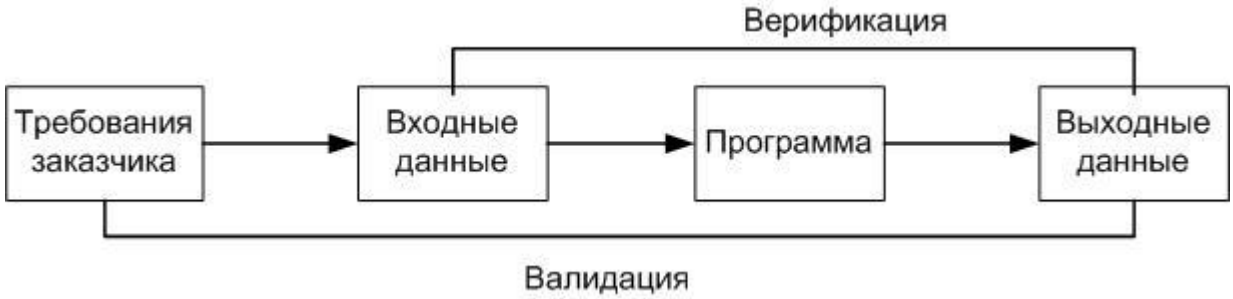



Рисунок 6.1 – Функции тестирования

**Цель тестирования** – снизить неопределенность представления о качестве продукта, распознать дефекты в объекте тестирования и увеличит вероятность, что при любых обстоятельствах продукт будет работать в установленных требованиях.

**Дефект** (error, bug, fail) – это отклонение фактического результата ото ожидаемого в процессе работы программы, а так же нежелательные побочные реакции на те или иные действия пользователя.

**QA** (quality assurance) – система обеспечения и управление качеством процессов при производстве любого продукта.

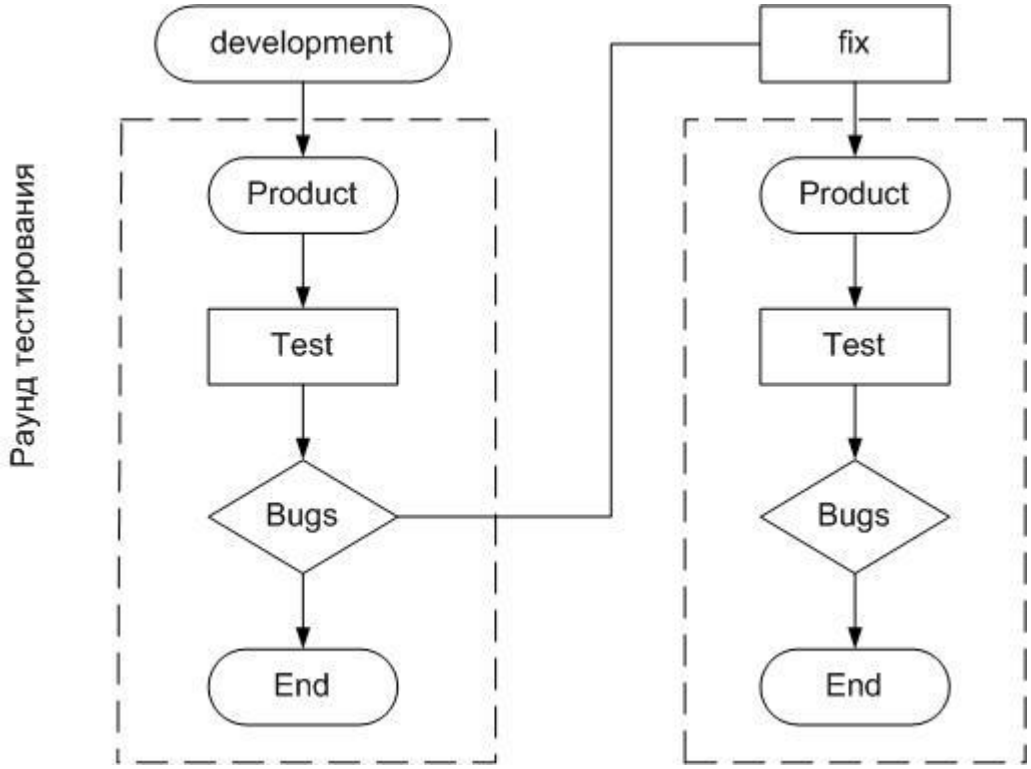


Рисунок 6.2 – цикл тестирования

**Fix** – это исправления, вносимые после раунда тестирования. Существует четыре типа доработок:

1) Fix1 – изъян в коде программы, правится только код, выявляется при помощи отладки и модульного тестирования.
2) Fix2 – изъян в спецификации, нужно править и спецификацию и код, выявляется на этапе итерационного тестирования.
3) Fix3 – изъян в архитектуре, необходимо менять архитектуру, спецификацию и программный код.
4) Fix4 – ошибки в требованиях.

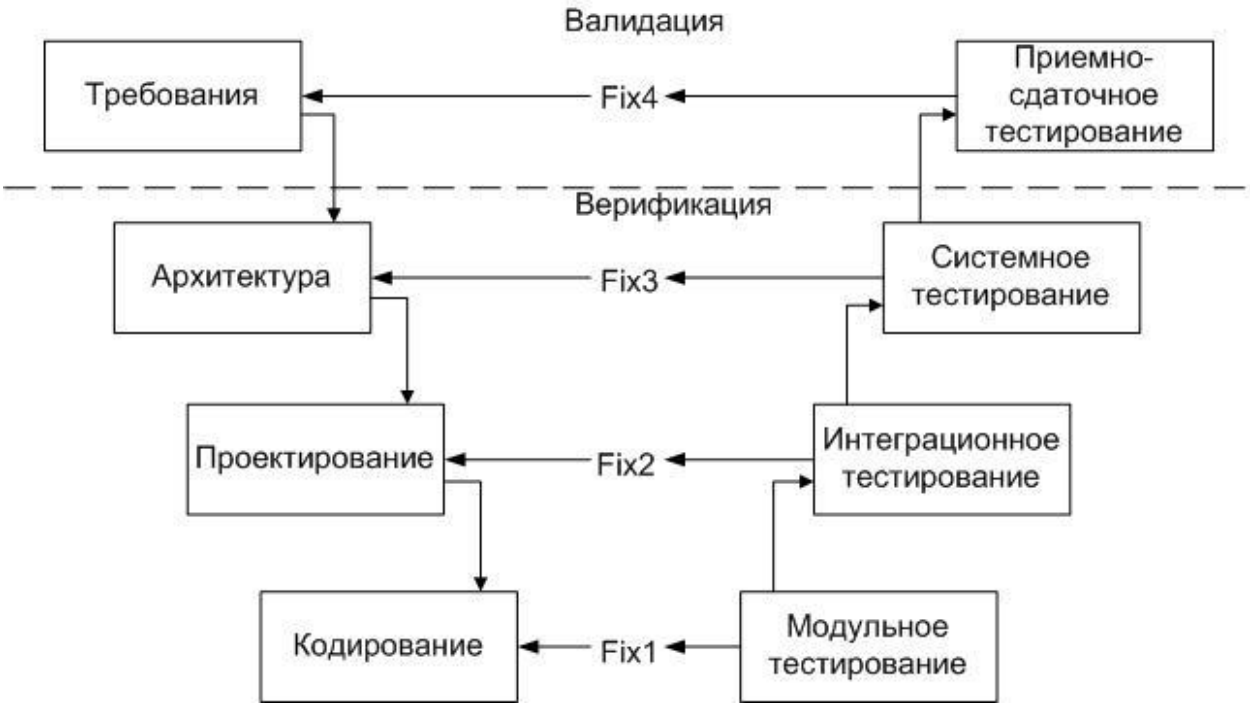


Рисунок 6.3 – V-модель (Fix)

**Классификация методов тестирования:**

1) Исполняется либо нет программного кода
   - статическое тестирование;
   - динамическоетестирование;
2) Известны или нет способы реализации
   - тестирование белого ящика;
   - тестированиечерного ящика;
3) По уровню объекта тестирования
   - модульное (отдельные модули);
   - интеграционное (несколько модулей);
   - подсистем;
   - системное (тестируется весь продукт);
   - β-тестироввние (тестирование пользователя);
   - приемно−сдаточное тестирование;
4) По свойствам объекта тестирования
   - функциональное;
   - тестирование производительности (нагрузочное);
   - совместимость (на различных платформах);
   - тестирование удобства (usability);
   - тестирование надежности, отказоустойчивости;
5) По природе объекта тестирования
   - системы реального времени real-time;
   - web приложения;
   - e-business;

- научные.

**Unit –тестирование (модульное).** Выполняется программно, заключается в изолированной проверке каждого отдельного элемента, путем запуска тестов в искусственной среде.

Тест – драйвер подменят ту часть системы, или внешней среды, которая вызывает объект тестирования. Основные задачи:

- обеспечить передачу данных в объект тестирования;
- отобразить результаты вызова и проверить их правильность;

Заглушка подменяет ту часть системы, которую вызывает объект тестирования. Свойства:

- имеет тот же интерфейс;
- возвращает результат того же типа;
- заменяет недостающие компоненты и стимулирует их действия;
- должны имитировать аварийные ситуации;


Рисунок 6.4 – Unit-тестирование

**Функциональное тестирование.** Проверка соответствия ПО функциональным требованиям пользователя.

Необходимо использовать каждый use-case, используя как верные значения, так и заведомо ошибочные для подтверждения:

- Продукт адекватно реагирует на все вводимые данные и выводятся ожидаемые результаты.
- При неправильных данных должны выводится сообщения об соответствующих ошибках.

**Нагрузочное тестирование.** Целью является оценка способности системы правильно функционировать при некотором превышении планируемых нагрузок.

Проверяют нагружая систему на 75 % от максимальной нагрузки в течении суток. Необходимо для системы реального времени (real-time) с круглосуточной беспрерывной работой и для языков с прямым обращением памяти (C, C++).

**Регрессионное тестирование.** Направленно на обнаружение ошибок в уже протестированном коде.

Регрессионная ошибка – возникает после внесения правок в результате раунда тестирования.

## Документирование тестирования

**BugTrackingSystem (BTS)** – это система регистрирования и отслеживания жизненного цикла дефектов (ошибок).

BTS:

- bugzilla (web);
- bugTracker.Net;
- red mine;
- Rational Clear Quest.

## План тестирования

**План тестирования** – это документ описывающий роль тестирования, методы, критерии начала и окончания, а так же описание тест – кейсов.Содержит:

1. Введение – описание, сроки выполнения тестирования, список необходимых документов.
2. Необходимые ресурсы – аппаратные и программные ресурсы, список персонала с описанием ответственности.
3. Особенности и подходы к тестированию – типы, стратегии тестирования (т.е. как будет осуществляться тестирование).
4. Критерии начала тестирования – готовность тестового стенда (продукта) законченность разработки функциональных требований, наличие документов.
5. Критерии окончания тестов – требования к срокам, требования к количеству ошибок, выдержка определенного периода без изменений программного кода CodeFreeze (CF), выдержка определенного времени без новых ошибок ZeroBugBounce (ZBB).
6. Описание тест - кейсов.

**Тест –кейс** (тестовый случай) – это совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции.

3 этапа: Action(действие) >ExpectedResult(ожидание результатов) >TestResult(тест результат).

Таблица *6.1. – Описание тест-кейса.*

| № тест - кейса | № функциональных требований | Описание тест –кейса | Сценарий | Ожидаемый результат |
|---|---|---|---|---|
| | | | | |

**Сценарии:**

- PreCondition – предусловие, как довести систему для выполнения теста.
- TestCase – список действий, приводящий систему из одного состояния в другое.

- PostCondition – (необязательно) список действий приводящий систему в исходное состояние до тестов (актуально автоматических тестов – препятствует засорению БД).

## BugReport

**BugReport** – отчет о дефектах в программе – документ описывающий последовательность действий, приводящих к некорректной работе объекта тестирования с указанием причин и ожидаемых результатов.

Таблица *6.2 – BugReport*

| № вид | № тест - кейса | Описание названия | Воспроизводимость | Приоритет ошибки | Класс | Описание | Способ воспроизведения |
|---|---|---|---|---|---|---|---|
| | | | Всегда<br>Иногда<br>Не удалось воспроизвести | необязательно | | Что править в программном коде | Каким образом повторить |

**Класс ошибки** (severity)

- Критическая (criticalcrush)
- Серьезная (serious) – нарушение функционала.
- Неважная (minor) – не несет ущерб функционалу.
- Предложения (suggestion (Enhancement)) – по изменению.

**Алгоритм тестирования:**

I. Этап. Проверка в нормальных условиях. Предполагает реальные условия функционирования программы.

II. Этап. Проверка в экстремальных условиях. Тестовые данные включают: граничные условия. Область входных переменных, их границы, данные которые должны воспроизводиться программой как правильные данные (очень маленькие и большие числа), отсутствие данных.

III. Этап. Проверка на граничные количества элементов массивов (1 или большое количество).

IV. Этап. Проверка на исключительные ситуации, если данные лежат за пределами допустимых значений.

**Аксиомы тестирования:**

1) Невозможно тестировать свою собственную программу.
2) Тестирование производится для обнаружения ошибок (т.е. ошибки есть).
3) Тесты должны покрывать как конкретные функции, так и исключительные ситуации.
4) Необходимо избегать невоспроизводимых тестов.

## Автоматическое тестирование

**Автоматическое тестирование** предназначено для программ, для которых работоспособность и безопасность при любых входных данных являются наиважнейшими приоритетами: веб-сервер, клиент / сервер SSH, sandboxing, сетевые протоколы.

**Fuzztesting (фаззинг)**. Фаззинг – методика тестирования, при которой на вход программы подаются невадные, непредусмотренные или случайные данные. Суть в том, чтобы изменять ожидаемые программой входные данные в произвольных местах.

Пример: zznf (linux), minifuzz (windows), filefuzz (windows).

Фаззерыпротоколов: PROTOS (WAP, HTTP-reply, LDAP, SNMP, SID) (Java, SPIKE (linux))

Фреймворки файзеров: Snlley (фреймворк для создания сложных структур данных).

SATSolvers - решают задачи выполнимости булевых формул. Отвечает на вопрос: всегда ли выполнима заданная формула, и если не всегда, то выдается набор значений, на котором она ложна.

Пример: STP, MiniSAT.

**Avalanche** – инструмент обнаружения программных дефектов при помощи динамического анализа. Результатом такого анализа становится либо набор входных данных, на которых в программе возникает ошибка, либо набор новых тестовых данных, позволяющий обойти ранее не выполнявшееся и, соответственно, еще не проверенные фрагменты программы.

Инструмент состоит из 4-х основных компонентов: двух модулей расширения (плагинов) Valgrind-Tracegrind и Covgrind, инструмента проверки выполнимости ограничений STP и управляющего модуля.

**KLEE**не ищет подозрительные места, а пытается покрыть как можно больше кода и провести исчерпывающий анализ путей в программе. KLEEпредставляет собой символическую виртуальную машину, где параллельно выполняются символические процессы, где каждый процесс один из путей в исследуемой программе. Для каждого уникального пройденного пути сохраняется набор входных данных, необходимых для прохождения по этому пути.

Ограничения:

- Нет поддержки многопоточных приложений, символических данных с плавающей точкой (ограничения STP) ассемблерные вставки.
- Тестируемое приложение должно быть собрано в  - байт коде (так же как и его библиотеки!)
- Для эффективного анализа нужно править код.

**Cucumber**- это инфраструктура тестирования, позволяющая писать тесты на простом языке управляемой поведением разработки (BDD) в стиле:

- Given – представляет контекст выполнения сценария тестирования, направленной точки вызова сценария в приложении любые необходимые данные.
- When определяет набор операций, инициирующих тестирование, таких как действия пользователей или подсистем.
- Then описывает ожидаемый результат тестирования.

Т.е. в стиле условие, операция, результат, который понятен пользователю. Затем контрольные тесты записываются в файлы функций, охватывающих один или несколько сценариев тестирования. Cucumber интерпретирует тесты на указанном языке программы и использует Selenium для управления тестами в браузере.

**Управление браузером с помощью Selenium.**

Имея написанные тесты необходимо обеспечить их выполнение на Webбраузере, для этого и используется Selenium, инфраструктуру автоматизации Web-браузера хорошо сочетающуюся с Cucumber. Есть два способа управления браузером:

- Selenium – RSиспользует JavaScript для управления web-страницей и работает внутри песочницы JavaScript
- WebDriver использует встроенный механизм автоматизации, который работает быстрее и не так подвержен ошибкам, но поддерживает меньше браузеров.

# ТЕМА 7. ПРОЕКТИРОВАНИЕ И СТАНДАРТЫ

## Метод вариантных сетей для выбора среды разработки

Метод вариантных сетей используется для выбора одной из нескольких альтернатив. Основан на получении обобщенных показателей агрегирующий множество частных характеристик с учетом веса и значимости.

Состоит из следующих шагов

1. Определить частные показатели, характеризующие конкурентные средства вариантов.
2. Определить значение показателей по каждому из вариантов (в натуральных единицах измерения или баллах).
3. Экспертным методом установить значимость показателей так называемые весовые коэффициенты k.
4. Установить эталон для сравнения.
5. Сопоставить частные показатели с эталонными и определить индексы по каждому варианту.

$$I = \frac{k_i}{\max k_i}$$

6. Вычислить совокупный показатель конкурентоспособности

$$I_k = \frac{\sum_{i=1}^{n} k_i I}{\sum_{i=1}^{n} k_i}$$

Пример. Выбор языка программирования для разработки проекта

Таблица *7.1. – Сравнительный анализ средств разработки*

| Критерий | | | Коэфф. Вес-ти | Абсолютное значение показателей | | | Относительное значение показателей | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| № | Название | Ед. изм | | Вар. A | Вар. B | Вар. C | A | | B | | C | |
| | | | | | | | $k_{ij}$ | $b_i K$ | K | $b_i K$ | K | $b_i K$ |
| 1 | Требова-ния к ресурсам | Мб | 0.15 | 100 | 50 | 100 | 1 | 0.15 | 0.5 | 0.075 | 1 | 0.15 |
| 2 | Дружественность Пользова-телю | балл | 0.20 | 5 | 4 | 3 | 1 | 0.20 | 0.8 | 0.16 | 0.6 | 0.12 |
| 3 | цена | | 0.20 | 4 | 5 | 4 | 0.8 | 0.16 | 1 | 0.20 | 0.8 | 0.16 |
| 4 | Затраты наадапта-цию | балл | 0.15 | 5 | 3 | 3 | 1 | 0.15 | 0.6 | 0.09 | 0.6 | 0.09 |
| 5 | БД | балл | 0.15 | 5 | 3 | 3 | 1 | 0.15 | 0.6 | 0.09 | 0.6 | 0.09 |
| 6 | Соотв. совр. техники | балл | 0.15 | 5 | 4 | 5 | 1 | 0.15 | 0.8 | 0.12 | 1 | 0.15 |
| Итого: | | | | | | | | | | | | |
| | | | | | | | 0.96 | | 0.735 | | 0.79 | |

Выбор производился по трем вариантам:

Вариант A – C#

Вариант B – ObjectPascal

Вариант C – C++

На основании проведеннонго анализа можем сделать вывод, что наиболее приемлемый для нас вариант –это вариант A (C#).

## Стандарты оформления программного кода

Соглашение об именовании переменной, учитывая стиль именования, используя регистр и дополнительные символы.

Венгерская нотация – имена идентификаторов предваряются некоторыми префиксами.

Таблица*7.2 – Венгерская нотация*

| Префикс | Описание | Пример |
|---|---|---|
| s | Строка | sClientName |
| sz | Строка, которая не может быть NULL | szClientName |
| n, i | Целочисленная переменная | nSize, iDim |
| l | long (длинное, целое) | lText |
| b | Булевская переменная | bIsEmpty |
| t, dt, d | Дата, время | tDepartnu |
| a | Массив (array) | aMass |
| p | Указатель (pointer) | pBox |
| g | Глобальная переменная | g_NDim |
| C | Класс | CBus |
| T | Тип (type) | TObject |
| I | Интерфейсный класс | IClient |

**Применение малого и большого регистра**

1. Pascal Casing – начальная буква большой регистр
2. Camel_Casing – первая строчная

accesTime, privatm_accesTime, _ accesTime

Стандарт кодирования – набор правил и соглашений, используемых некотором языке программирования, включает:

- Способы выбора названий идентификаторов и используемый регистр символов.
- Стиль отступов при оформлении логических блоков.
- Способы расстановки скобок, ограничивающие логические блоки.
- Использование пробелов при оформлении выражений.
- Стиль комментариев.

**Стиль отступов**

Таблица*7.3 – Стиль отступов*

| K&R | СтильОлмана |
|---|---|
| if (<cound>)<br>{…<br>tab 8<body><br>} | if (<cound>)<br>{<br>tab 4-8 <body><br>} |
| **СтильУойте** | **GN4** |
| if (<cound>)<br>tab 4-8 { { <body> } | if (<cound>)<br>tab 4 {<br>tab 4-8 <body><br>tab 4 } |

## Парадигмы программирования

Парадигмы программирования – это системы идей и понятий, определяющих стиль написания компьютерной программы. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняющей компьютером.

Парадигмы:

Императивное программирование – описывает процесс вычислений в виде инструкций, изменяющих состояние программы. Особенностью является присваивание, что увеличивает сложность моделей вычислений и делает программы подверженными специфическим ошибкам.

Декларативное программирование

Если программа описывает какого нечто, а не как его создать (HTMLстраницы)

Программа декларативна, если она написана на исключительно функционально логическом языке программирования с ограничениями.

Программы легко поддаются методикам метапрограммирования – когда программа может генерироваться по ее описанию.

Структурное программирование – в основе лежит представление программы в виде иерархической структуры блоков.

Любая программа представляет собой структуру, построенную из трех типов базовых конструкций: последовательные выполнения, ветвления, цикл.

Повторяющиеся или нет блоки, могут быть оформлены в виде подпрограмм (функции, процедуры).

Разработка пошагово, методом «сверху вниз».

Функциональное программирование – процесс вычисления (программирование) трактуется как вычисления значений функций в математическом понимании последних. Функциональное программирование предполагает обходиться вычисление результатов функций от исходных данных и резуль-

татов других функций, и не полагает явного хранения состояния программы. Соответственно, не предлагает оно и изменяемость этого состояния (нет понятия переменная).

Логическое программирование – парадигма, основанная на автоматическом доказательстве теорем (принципы логического вывода информации на основе заданных правил и правил вывода).

Объектно-ориентированное программирование – парадигма, в которой основными концепциями являются понятия объектов и классов.

Инкапсуляция – это свойство системы, позволяющее описать новый класс на основе уже существующего, с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом; новый класс - потомком, наследником или производным классом.

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта (одинаковая спецификация – разная реализация).

Класс – описываемый на языке терминологии (пространство имен) исходного кода моделью еще не существующей сущности (объекта). Фактически он описывает устройство объекта, являясь своего рода чертежом.

Объект – сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса или копировании прототипа.

Прототип – это объект образец, по образу и подобию которого создаются другие объект.

## Шаблоны проектирования

При реализации проектов по разработке программных систем и моделировании бизнес-процессов встречаются ситуации, когда решение проблемы в различных проектах имеют сходные структурные черты.

### Классификация паттернов по категориям их применения:

Архитектурные паттерны – множество предварительно определенных подсистем со спецификацией их ответственностей, правил и базовых принципов установления отношений между ними.

Паттерны проектирования – специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними.

Паттерны анализа – специальные схемы для представления общей организации процесса моделирования.

Паттерны тестирования – специальные схемы для представления общей организации процесса тестирования программных средств.

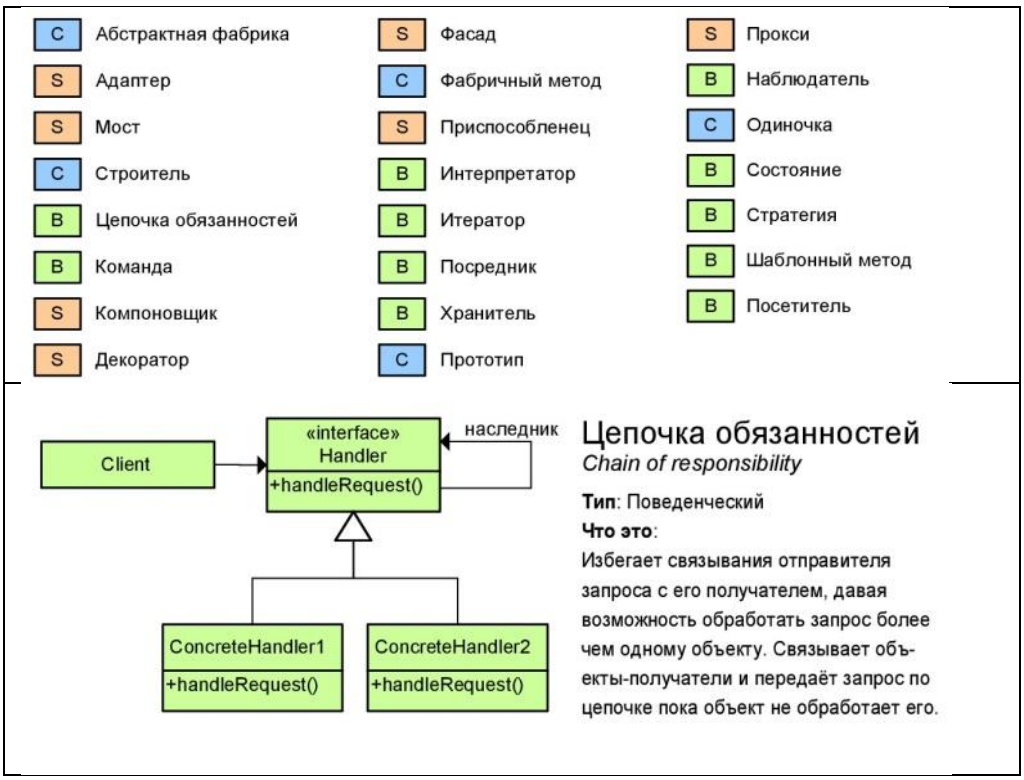
Паттерны реализации – совокупность компонентов и других элементов реализации, используемых в структуре модели при написании программного кода.

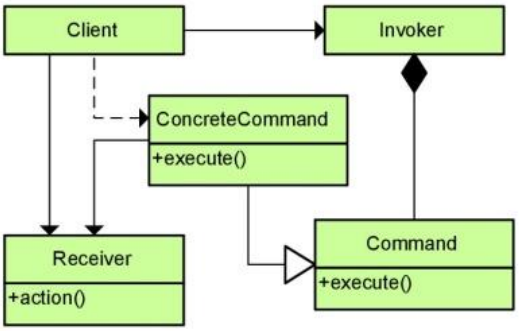
## Паттерны проектирования

Описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте. Наиболее известными в этой категории являются паттерныGoF (GrandofFour), названые 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. Эти паттерны не зависят от языка реализации, но их реализация зависит от области приложения.

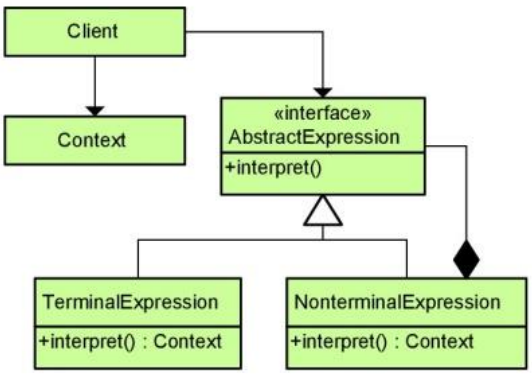
Паттерны проектирования в контексте UML представляют собой параметризованную кооперацию вместе с описание основных базовых принципов использования.





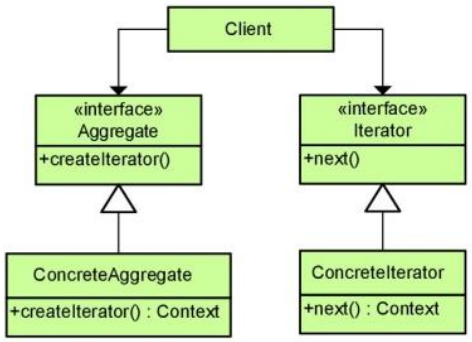

## Команда
*Command*

**Тип**: Поведенческий
**Что это**:
Инкапсулирует запрос в виде объекта, позволяя передавать их клиентам в качестве параметров, ставить в очередь, логировать а также поддерживает отмену операций.



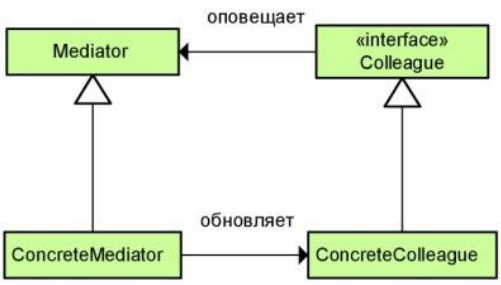

## Интерпретатор
*Interpreter*

**Тип**: Поведенческий
**Что это**:
Получая формальный язык, определяет представление его грамматики и интерпретатор, использующий это представление для обработки выражений языка.



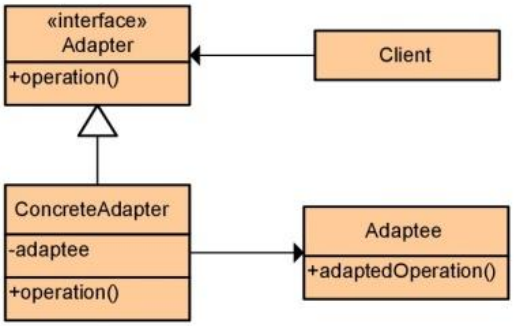

## Итератор
*Iterator*

**Тип**: Поведенческий
**Что это**:
Предоставляет способ последовательного доступа к элементам множества, независимо от его внутреннего устройства.



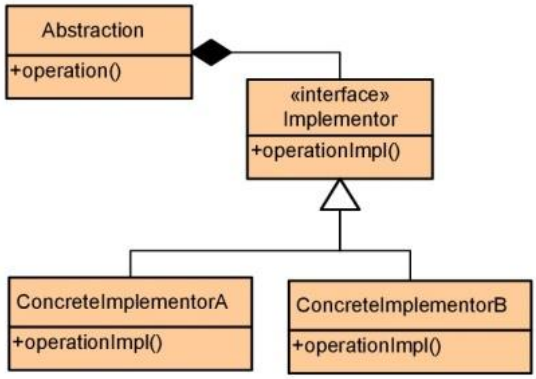

## Посредник
*Mediator*

**Тип**: Поведенческий
**Что это**:
Определяет объект, инкапсулирующий способ взаимодействия объектов. Обеспечивает слабую связь, избавляя объекты от необходимости прямо ссылать друг на друга и даёт возможность независимо изменять их взаимодействие.



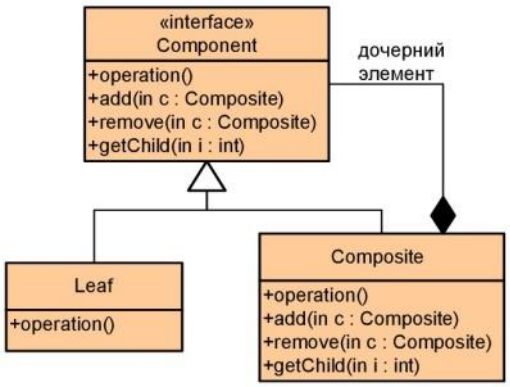

## Адаптер
*Adapter*

**Тип**: Структурный
**Что это**:
Конвертирует интерфейс класса в другой интерфейс, ожидаемый клиентом. Позволяет классам с разными интерфейсами работать вместе.



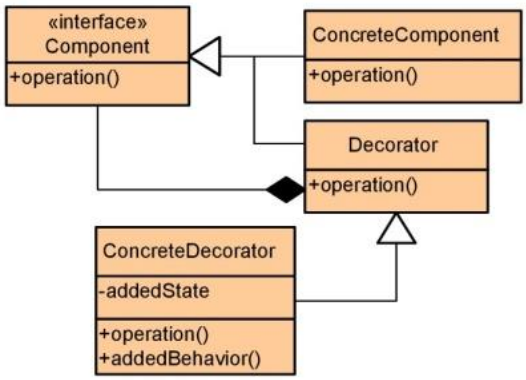

## Мост
*Bridge*

**Тип**: Структурный
**Что это**:
Разделяет абстракцию и реализацию так, чтобы они могли изменяться независимо.



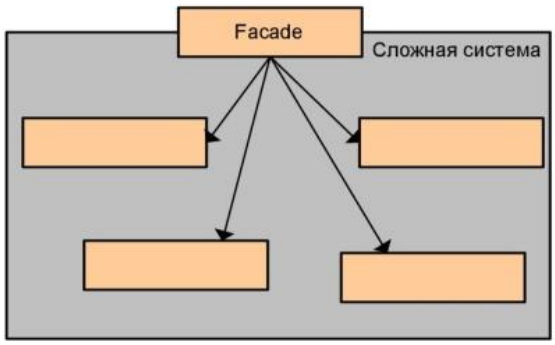

## Компоновщик
*Composite*

**Тип**: Структурный
**Что это**:
Компонует объекты в древовидную структуру, представляя их в виде иерархии. Позволяет клиенту одинаково обращаться как к отдельному объекту, так и к целому поддереву.



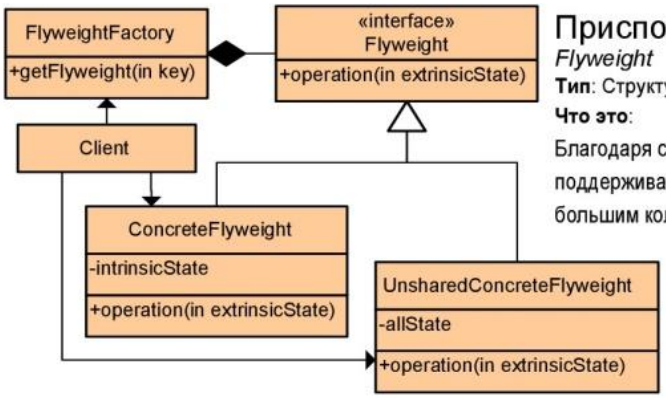

## Декоратор
*Decorator*

**Тип**: Структурный
**Что это**:
Динамически предоставляет объекту дополнительные возможности. Представляет собой гибкую альтернативу наследованию для расширения функциональности.




## Фасад
*Facade*

**Тип**: Структурный
**Что это**:
Предоставляет единый интерфейс к группе интерфейсов подсистемы. Определяет высокоуровневый интерфейс, делая подсистему проще для использования.

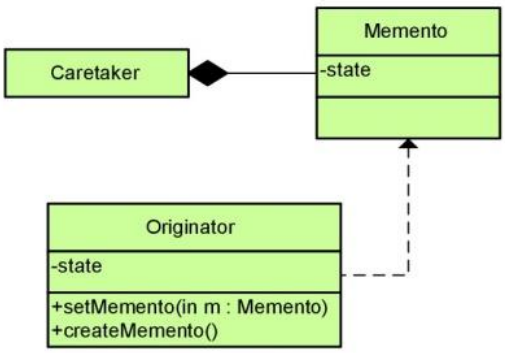



## Приспособленец
*Flyweight*

**Тип**: Структурный
**Что это**:
Благодаря совместному использованию, поддерживает эффективную работу с большим количеством объектов.

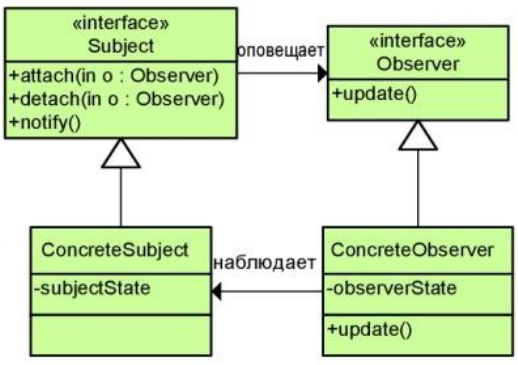
## Хранитель
*Memento*

**Тип**: Поведенческий
**Что это**:
Не нарушая инкапсуляцию, определяет и сохраняет внутреннее состояние объекта и позволяет позже восстановить объект в этом состоянии.



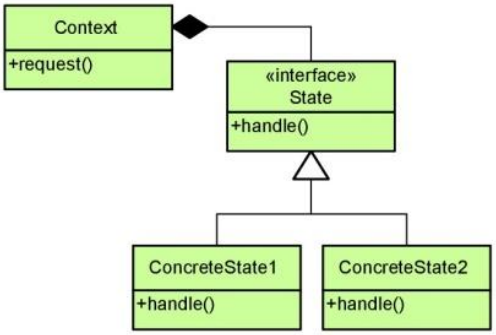

## Наблюдатель
*Observer*

**Тип**: Поведенческий
**Что это**:
Определяет зависимость “один ко многим” между объектами так, что когда один объект меняет своё состояние, все зависимые объекты оповещаются и обновляются автоматически.



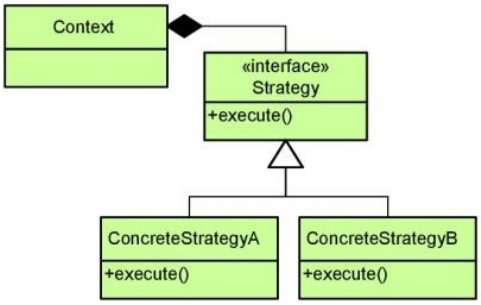
## Состояние
*State*

**Тип**: Поведенческий
**Что это**:
Позволяет объекту изменять своё поведение в зависимости от внутреннего состояния.



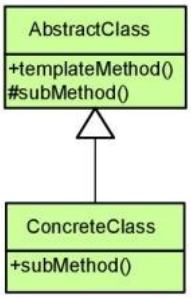

## Стратегия
*Strategy*

**Тип**: Поведенческий
**Что это**:
Определяет группу алгоритмов, инкапсулирует их и делает взаимозаменяемыми. Позволяет изменять алгоритм независимо от клиентов, его использующих.



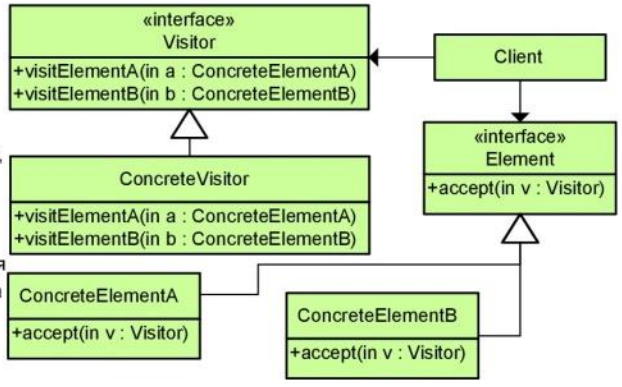

## Шаблонный метод
*Template method*

**Тип**: Поведенческий
**Что это**:
Определяет алгоритм, некоторые этапы которого делегируются подклассам. Позволяет подклассам переопределить эти этапы, не меняя структуру алгоритма.



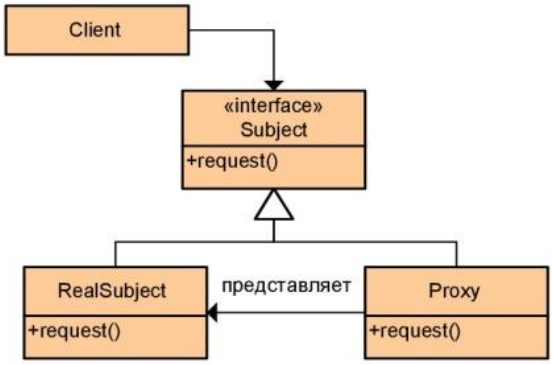

## Посетитель
*Visitor*

**Тип**: Поведенческий
**Что это**:
Представляет собой операцию, которая будет выполнена над объектами группы классов. Даёт возможность определить новую операцию без изменения кода классов, над которыми эта операция проводится.



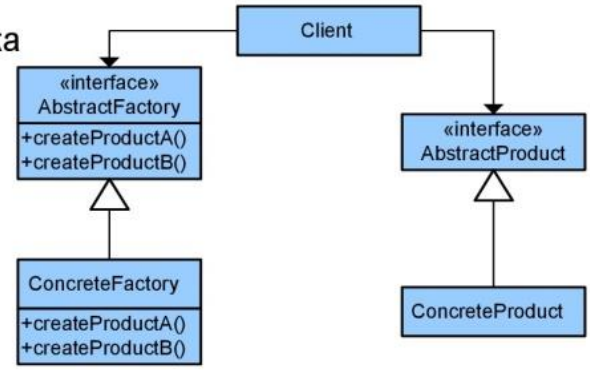
## Прокси
*Proxy*

**Тип**: Структурный
**Что это**:
Предоставляет замену другого объекта для контроля доступа к нему.



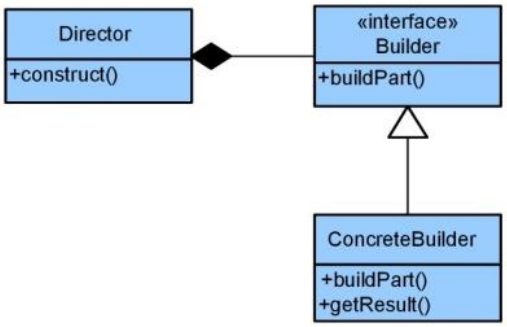

## Абстрактная фабрика
*Abstract factory*

**Тип**: Порождающий
**Что это**:
Предоставляет интерфейс для создания групп связанных или зависимых объектов, не указывая их конкретный класс.



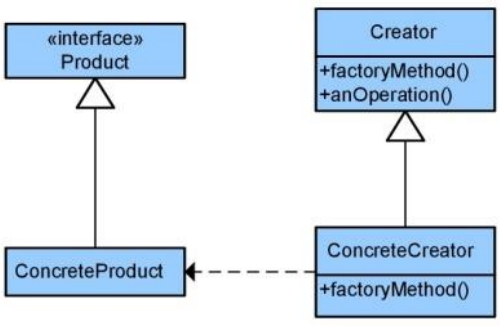

## Строитель
*Builder*

**Тип**: Порождающий
**Что это**:
Разделяет создание сложного объекта и инициализацию его состояния так, что одинаковый процесс построения может создать объекты с разным состоянием.




## Фабричный метод
*Factory method*

**Тип**: Порождающий
**Что это**:
Определяет интерфейс для создания объекта, но позволяет подклассам решать, какой класс инстанцировать. Позволяет делегировать создание объекта подклассам.

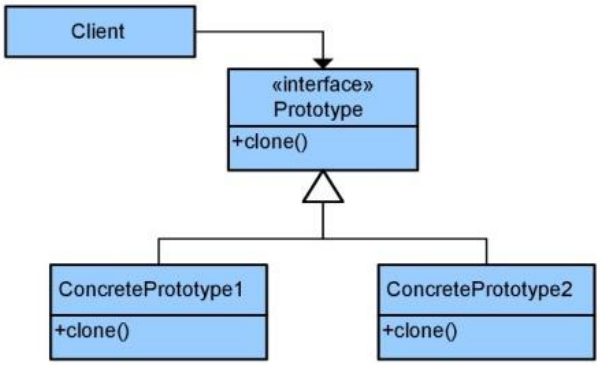


## Прототип
*Prototype*

**Тип**: Порождающий

**Что это**:

Определяет несколько видов объектов, чтобы при создании использовать объект-прототип и создаёт новые объекты, копируя прототип.




## Одиночка
*Singleton*

**Тип**: Порождающий

**Что это**:

Гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к нему.