1. **Проектирование программного обеспечения** — процесс создания проекта программного обеспечения (ПО), а также дисциплина, изучающая [методы проектирования](http://ru.wikipedia.org/w/index.php?title=%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F&action=edit&redlink=1). Проектирование ПО является частным случаем [Проектирования](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) продуктов и процессов.

**Целью** проектирования является определение внутренних свойств системы и детализации её внешних (видимых) свойств на основе выданных заказчиком [требований к ПО](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%BC%D1%83_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8E) (исходные условия задачи). Эти требования подвергаются анализу.

1. Удовлетворение функциональным требованиям.

2. Надежность

3. Удовлетворение явными – неявными требованиями по ресурсопотреблению

4. Дизайн (явн. – неявн.)

5. Цена

6. Требования к процессу разработки.

**Основные проблемы**: высокая стоимость и низкая надежность

(Диаграмма стоимости, рисунок)

Сопровождение(67%), тестирование(15%), программирование(7%), проектирование(11%).

Говорят, что ПО содержит ошибку, если его поведение не соответствует спецификации. В ПО ошибка,

если оно не выполняет того, что пользователю разумно от него ожидать. Отказ ПО – проявление в нем

ошибки. Любая программа и спецификация содержит всегда ошибки. Надежность ПО – вероятность

работы без отказа в течение определенного времени, рассчитана с учетом стоимости для пользователя

каждого отказа.

Технология проектирования – система принципов для создания надежного, экономичного, эффективного

ПО, работающего в разных сложных системах.

Различают методы, средства и процедуры для технологии проектирования ПО.

**Методы обеспечивают**:

1. Планирование и оценка проекта

2. Анализ системных и программных требований

3. Проектирование архитектуры и структуры программы

4. Решение задач кодирования, тестирования и сопровождения.

**Средства**:

1. Автоматизированная/автономная поддержка методов.

2. CASE – системы (Computer – Aided Software Engin)

**Процедуры** определяют порядок применения методов и утилит, формирование отчетов по определению

требований, контроль качества + координация изменений, оценка прогресса проекта. Процедуры

обеспечивают непрерывную технологическую цепочку разработки.

**Критерии.** Качество кода может определяться различными критериями. Некоторые из них имеют значение только с точки зрения человека. Например, то, как отформатирован текст программы, совершенно не важно для компьютера, но может иметь серьёзное значение для последующего сопровождения. Многие из имеющихся [стандартов оформления кода](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82_%D0%BE%D1%84%D0%BE%D1%80%D0%BC%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%BA%D0%BE%D0%B4%D0%B0), определяющих специфичные для используемого языка соглашения и задающие ряд правил, улучшающих читаемость кода, имеют своей целью облегчить будущее сопровождение ПО, включающее отладку и обновление. Существуют и другие критерии, определяющие, «хорошо» ли написан код, например, такие, как структурированность — степень логического разбиения кода на ряд управляемых блоков.

* Читаемость кода
* Лёгкость поддержки, [тестирования](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), отладки, исправления ошибок, изменения и [портируемости](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F" \o "Портирование программного обеспечения)
* Низкая сложность кода
* Низкое использование ресурсов: памяти и процессорного времени
* Корректная обработка [исключительных ситуаций](http://ru.wikipedia.org/w/index.php?title=%D0%98%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D1%8F&action=edit&redlink=1)
* Малое число предупреждений при [компиляции](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) и [линковке](http://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%BA%D0%BE%D0%B2%D1%89%D0%B8%D0%BA)

1. **Жизненный цикл программного обеспечения** (ПО) — период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации[[1]](http://ru.wikipedia.org/wiki/%D0%96%D0%B8%D0%B7%D0%BD%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%86%D0%B8%D0%BA%D0%BB_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F#cite_note-1). Этот цикл — процесс построения и развития ПО.

**Модель жизненного цикла ПО** — структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении жизненного цикла. Модель жизненного цикла зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует. Классическая модель - это спиральная модель.

**Водопадная** модель жизненного цикла ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *waterfall model*) была предложена в 1970 г. [Уинстоном Ройсом](http://ru.wikipedia.org/w/index.php?title=%D0%A3%D0%B8%D0%BD%D1%81%D1%82%D0%BE%D0%BD_%D0%A0%D0%BE%D0%B9%D1%81&action=edit&redlink=1). Она предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

[**Спиральная модель**](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C) ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *spiral model*) была разработана в середине 1980-х годов [Барри Боэмом](http://ru.wikipedia.org/w/index.php?title=%D0%91%D0%B0%D1%80%D1%80%D0%B8_%D0%91%D0%BE%D1%8D%D0%BC&action=edit&redlink=1). Она основана на классическом [цикле Деминга](http://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB_%D0%94%D0%B5%D0%BC%D0%B8%D0%BD%D0%B3%D0%B0) PDCA (plan-do-check-act). При использовании этой модели ПО создается в несколько [итераций](http://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F) (витков спирали) [методом прототипирования](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F).

Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации.

На каждой итерации оцениваются:

* риск превышения сроков и стоимости проекта;
* необходимость выполнения ещё одной итерации;
* степень полноты и точности понимания требований к системе;
* целесообразность прекращения проекта.

**Последовательность шагов:**

1. Анализ

2. Проектирование

3. Кодирование

4. Тестирование

5. Сопровождение

1)Анализ требований и целей.

2)Описывается структура, архитектура ПО, структура модулей и алгоритмов.

3) и 4) Это выполнение программы для выявления ошибок.

5) Внесение изменений в эксплуатированное ПО.

Достоинства: документация.

Недостатки: туго отклонения от последовательности шагов, т.е. сильно зависит от точной

формулировки задачи (цели).

* 1. **Основные понятия надежности систем.**По определению, установленному в ГОСТ 13377-75, ***надежность***— свойство объекта выполнять заданные функции, сохраняя во времени значения установленных эксплуатационных показателей в заданных пределах, соответствующих заданным режимам и условиям использования, технического обслуживания, ремонта, хранения и транспортирования. Таким образом, надежность является внутренним свойством системы, заложенным при ее создании и проявляющимся ***во времени***при функционировании и эксплуатации.

**Низкая надежность** – во многом следствие высокой сложности ПО. Сложность определяется

количеством внутренних компонент и количеством связей между ними. Связи могут быть

разнотипными. Сложность архитектуры определяется количеством связей.

Сложность структуры программы – функция модулей и связей. Компоненты сложности:

1) Сложность самой проблемы

2) Сложность управления процессом разработки

3) Сложность обеспечения гибкости конечного программного продукта

4) Описание поведения отдельных подсистем.

Продолжать

Большие системы имеют тенденцию к эволюции в процессе их использования. Сложность

настолько высока, что один разработчик не может учесть всех тонкостей. Измеряется в

человекагодах. Например, для Windows NT – 2000 чел-т. Возникают трудности с

координацией работы. Главная проблема руководства – поддержание целостности основной

идей в ходе работы. ПО нужно разрабатывать таким образом, чтобы продукт можно было

легко модифицировать и развивать. Проектировать желательно так, чтобы взаимовлияние

частей было меньше.

**Ошибки сопряжений**. группа этих ошибок вызывает неверное взаимодействие ПО с другими программами или подпрограммами, с системными программами, устройствами ЭВМ или входными данными.

Искажение входной информации. Указанная причина вызывает нарушение функционирования ПО, когда входные данные не попадают в допустимую область значения переменных. В этом случае возникает несоответствие между исходной информацией и возможностями программы.

Неверные действия пользователя связаны с неправильной интерпретацией сообщений, с неправильными действиями пользователя при работе в диалоговом режиме. Часто эти ошибки являются следствием некачественной программной документацией.

Неисправность аппаратных средств ИС. Эти неисправности оказывают определенное влияние на характеристики надежности ПО. Появление отказов или сбои в работе аппаратуры приводят к нарушению хода обработки информации и, как следствие, могут искажать как исходные данные, так и саму программу.

Следствием появления ошибок в программе является ее отказ. Последствия отказов ПО можно разделить на:

- полное прекращение выполнения функций программы;

- кратковременное нарушение хода обработки информации в ИС.

Степень серьезности последствий отказов ПО оценивается соотношением между временем восстановления программы после отказа и динамическими характеристиками объектов, использующих результаты работы этой программы.

Аварийное завершение работы прикладного ПО легко идентифицируется, так как операционная система выдает сообщения, содержащие аварийный код. Характерными причинами появления аварийного завершения являются ошибки при выполнении макрокоманды, неверное использование методов доступа, нарушение защиты памяти, нехватка ресурсов памяти, неверное использование макрокоманды, возникновение программных прерываний, для которых не указан обработчик, и другие причины.

* 1. **Уменьшение сложности программных систем** достигается за счет использования интегрированных программных средств.

Для решения проблемы сложности использовуются концепции декомпозиции, абстракции и

иерархии.

**Декомпозиция** – такое разбиение системы, чтобы высокочастотная динамика ее была

заключена в данных компонентах, а межкомпонентное взаимодействие предоставляло лишь

низкочастотную динамику системы (Г.Буч). Т.е.задача эта сродни разрезания графа, чтобы

было минимум ребер. Сущ-ет декомпозиция алгоритмическая и объектно-ориентированная.

Алгоритмическая – обычное разделение алгоритмов, где каждый модуль системы выполняет

один из важных этапов процесса (рисунок: основная функция – f1,f2,f2,….от них далее и т.д.)

Объектно – ориентированная. Критерий разделения – принадлежность объекта различным

абстракциям данной предметной области.

(тут нарисована диаграмма классов, но в конспекте ее не было)

Особенности: основана на объектах, а не на алгоритмах.

**Абстракция** – это такие сущ-ные характеристики объекта, которые отличают его от др. видов

объектов и определяют его особенности с точки зрения рассмотрения и анализа.

Абстрагирование позволяет получить некоторую модель предметной области, которая

содержит объекты, отражающие реальные объекты. В процессе сущ.отсечение

несущественных деталей.

**Иерархия** позволяет разделить систему по уровням понимания. Бывает структурная и по типу.

Структурная: отношение «целое – часть». То есть отношение ассоциации и агрегации.

По типу: отношение «тип – подтип».

Концептуальные особенности структурного и ООП:

1)схема водопада 1)спиральная модель

2)абстракция (структуры) 2)абстракция (классов)

3)декомпозиция (алгоритм.) 3)декомпозиция (об. –ориент.)

4)иерархия (структурная) 4)иерархия (структ.- типовая)

* 1. **Структурное проектирование** основано на модульной структуре программного продукта и типовых управляющих структурах алгоритмов обработки данных различных программных модулей.

В любой типовой структуре блок, кроме условного, имеет только один вход и выход, безусловный переход на блок с нарушением иерархии запрещен (оператор типа GoTo в структурном программировании не используется). Цель процесса структурного проектирования заключается в разработке технического решения, которое удовлетворяет требованиям к программной системе.

**Проектирование программной системы:**

1. Построение объектной модели.

1.1. Определение классов.

1.2. Подготовка словаря.

1.3. Определение зависимостей.

1.4. Уточнение атрибутов.

1.5. Дальнейшее усовершенствование модели.

Требования: пользователь формирует свои потребности к будущему проекту. Проекты по

такому принцип можно разбить на:

1.Управляемые пользователем

2.Контролируемые пол-ем

3.Не зависимые от пол-ля

**Схема структурного проектирования**;

Требования заказчиком полностью, ответственность за опр. требования – на разработчика.

Требования к крупной системе осущ. спец. группой людей: сост. из представителей заказчиков и

разработчика. Один от заказчика имеет право для принятия решения (босс), но не явл.

пользователем, другой опытный пользователь. Процесс заключается:

- анализ имеющихся систем

- беседа с потенциальными пользователями

- проведение оценки осуществленности и достоинств проекта. Рассмотреть задачи, относящиеся к

проекту

- проверка правильности требований

Требования

Цели

Предварительное внешнее проектирование

Детальное внешнее проектирование

Стук-ра программы

Внеш.проек.модуля

Проект.логики модуля

Проект БД

Арх.системы

П(проект)

А(анализ)

**Пользователь ответственен за проверку требований на полноту и точность, а разработчик – за**

**проверку осуществимости и понятности.** Для каждого требования - приоритет, чтобы принимать

компромиссные решения.

Цели: конкр. ориентиры для проектирования программного продукта. Бывают цели продукта и

цели проекта. ЦПродукта с точки зрения пользователя, ЦПроекта – для разработчика.

C точки зрения пользователя:

1. Резюме

2. Определение роли пользователя, перечисление ф - ий

3. Документация (цели для документации, типы документации и круг читателей для каждого

типа)

4. Эффективность: характеристики, пропускная способность и т.п.

5. Конфигурация: указывает, от чего программа будет зависеть и работать

6. Безопасность обеспечения цели

7. Обслуживание: намеченная стоимость и время исправления ошибок

8. Установка – методы и средства настройки системы под конкретные условия

9. Надежность

Цели относятся к разработке и в явном виде не проявляются, в конечном результате:

- ориентировочная стоимость каждого этапа

- календарный план проекта

- цели для каждого процесса тестирования

- степень адаптированности ПО, которая дожна быть достигнута

- вопросы сопровождения системы, которые необходимо учитывать при разработке

- уровни надежности для каждого этапа разработки для достижения заданной надежности

системы

- внутренняя документация проекта

- критерий для оценки готовности продукта к использованию.

* 1. **Этап анализа. Взаимодействие с заказчиком для уточнения функциональности.**

**Модели взаимодействия с заказчиком в области разработки программного обеспечения:**

**Модель проектного аутсорсинга**. В рамках модели предполагается выделение индивидуального менеджера проекта взаимодействия с заказчиком (сбора функциональных требований, стоимостной оценки реализации проекта, уточнения требований и демонстрации результатов).

Как правило, модель применяется для ограниченной по времени разработки одного конкретного программного продукта (модуля продукта).

**Модель выделения группы разработки**. Модель предполагает выделение под задачи заказчика отдельной группы специалистов F-Technology (менеджера, разработчиков, специалистов по тестированию, аналитиков) на длительный период. Конкретный состав группы определяется совместно с заказчиком, в зависимости от специфики и объема работ. Выделенная группа обладает всеми компетенциями, необходимыми заказчику, работает в тесном сотрудничестве с представителями заказчика.

Такой формат взаимодействия направлен на длительное сотрудничество и гарантирует глубокое погружение группы в специфику деятельности заказчика, что приводит к реализации наиболее эффективного с точки зрения функциональности программного обеспечения.

В рамках данной модели возможна совместная разработка программного обеспечения с привлечением группы разработки заказчика.

**Этап планирования и анализа требований**

1. Установочная встреча с заказчиком и инструктором.

2. Отчет о формировании плана проекта.

3. Отчет об основных требованиях пользователя и моделях пред-метной области. Уточнение плана проекта.

4. Отчет о ходе анализа требований. Черновые варианты моделей требований и высокоуровневой архитектуры. Уточнение плана проекта.

5. Отчет о создании спецификации требований. Модели требова-ний и высокоуровневой архитектуры. Критерии аттестации. Документ спецификации требований (техническое задание). Согласование с заказ-чиком. Уточнение плана проекта.

* 1. **Макетирование** - это метод конструирования физических объектов, при помощи которого создают пространственные трехмерные модели.

В программировании словом "прототип" чаще называется промежуточный упрощённый вариант программной системы, позволяющий представить заказчику, как будет выглядеть и работать законченная система (другое слово для обозначения того же самого -- макет). Иногда под прототипом понимается изделие, используемое в качестве образца при разработке другого изделия. Здесь же прототипом называется тщательно разработанная и отлаженная заготовка программной системы, предназначенная для использования в качестве основы для создания различных программных систем, архитектура которых относится к одному типу. Прототип программной системы - это, можно сказать, её скелет. В устройстве прототипа фиксируется определённая культура программирования.

**Макетирование (прототипирование),** тоже, что и спиральная модель, но без тщательного

проектирования. Подходит для не согласованных заказчиков и непонятливых разработчиков, т.к.обычно в худшем случае приходиться переписывать 80 – 90%.

Создается макет (прототип) проекта для снятия неопр- ти требования заказчика. Макет может

быть на бумаге в виде спецификаций, может быть упрощенный работающий макет, тут иметься

два вида: 1) «на выброс», 2) «эволюционное макетирование» - это когда макет создается на основе

уже сущ-го макета, путем внесения ряда изменений:

Сбор и уточнение требований

Быстрое проектирование

Построение макета

Оценка макета заказчиком

Уточнение требований

Недостатки:

1) Макет может быть принят заказчиком за продукт

2) Макет может быть принят разработчиком за продукт

3) У заказчика может сложиться неверное представление о сроках разработки программы.

4) Разработчик может идти на компромисс, используя не подходящие структуры данных и

компоненты, забывая, почему они не подходят.

* 1. **Требования к программному обеспечению** — совокупность утверждений относительно атрибутов, свойств или качеств программной системы, подлежащей реализации. Создаются в процессе разработки требований к программному обеспечению, в результате анализа требований.

Требования могут выражаться в виде текстовых утверждений и графических моделей.

В классическом техническом подходе совокупность требований используется на стадии проектирования ПО. Требования также используются в процессе проверки ПО, так как тесты основываются на определённых требованиях.

Этапу разработки требований, возможно, предшествовало технико-экономическое обоснование, или концептуальная фаза анализа проекта. Фаза разработки требований может быть разбита на выявление требований (сбор, понимание, рассмотрение и выяснение потребностей заинтересованных лиц), анализ (проверка целостности и законченности), спецификация (документирование требований) и проверка правильности.

**Виды требований по уровням**

Бизнес-требования — определяют назначение ПО, описываются в документе о видении (vision) и границах проекта (scope).

Пользовательские требования — определяют набор пользовательских задач, которые должна решать программа, а также способы (сценарии) их решения в системе. Пользовательские требования могут выражаться в виде фраз утверждений, в виде способов применения (use case), пользовательских историй (user story), сценариев взаимодействия (scenario).

Функциональные требования — охватывают предполагаемое поведение системы, определяя действия, которые система способна выполнять[источник не указан 815 дней]. Описывается в системной спецификации (англ. system requirement specification, SRS).

**Виды требований по характеру**

Функциональный характер — требования к поведению системы

Бизнес-требования

Пользовательские требования

Функциональные требования

Нефункциональный характер — требования к характеру поведения системы

Бизнес-правила — определяют ограничения, проистекающие из предметной области и свойств автоматизируемого объекта (предприятия)

Системные требования и ограничения — определения элементарных операций, которые должна иметь система, а также различных условий, которым она может удовлетворять. К системным требованиям и ограничениям относятся:

Ограничения на программные интерфейсы, в том числе к внешним системам

Требования к атрибутам качества

Требования к применяемому оборудованию и ПО

Требования к документированию

Требования к дизайну и юзабилити

Требования к безопасности и надёжности

Требования к показателям назначения (производительность, устойчивость к сбоям и т. п.)

Требования к эксплуатации и персоналу

Прочие требования и ограничения (внешние воздействия, мобильность, автономность и т. п.).

* 1. **Определение целей на этапе анализа: общие цели, цели продукта и цели проекта.**

**Проект** — деятельность, направленная на создание уникального продукта (или результата) за ограниченный период времени силами специально созданной для этого организационной структуры, которая расформировывается после завершения проекта.

**Цель проекта** — то, чем должен завершиться проект. Проект может иметь несколько:

**Бизнес-цель проекта** — создание добавочной стоимости, увеличение капитализации, сокращение расходов и т.д. Бизнес-цель тесно связана с понятием Результат проекта.

**Техническая цель проекта** — эквивалентна понятию Продукт проекта, т.е. это то, что должно быть создано в результате проекта.

**Финансовые цели**

Они все связаны с деньгами. Например, проект должен дать 15% прибыли на инвестированные деньги.

**Качественные цели**

Эти цели указывают на степень «хорошести» результата. Например, если в вашем проекте вы предполагаете повысить уровень удовлетворенности клиентов, то вы можете выразить это в повышении количества положительных отзывов.

**ВременнЫе цели**

Например, ваш проект может быть привязан к определенной дате: дате выпуска продукции вашей компании, дате проведения выставки, дате сдачи здания в эксплуатацию.

Для того чтобы проект был успешным, у него должна быть четко определенная и реальная цель. Цель проекта - желаемый результат деятельности, достигаемый в пределах установленного интервала времени.

Сформулированные цели должны соответствовать принципу SMART, согласно которому они должны быть:

* ясными и точными (S - Specific);
* измеримыми (M - Measurable),
* достижимыми (A - Achievable);
* непротиворечивыми как между собой так и со стратегическими целями организации (R - Related);
* определены по срокам их достижения (T - Times-bound).

К формулированию целей нужно отнестись внимательно по ряду причин:

* разное понимание целей участниками проекта приведет к ненужной трате ресурсов и усилий рабочей группы, цели достигнуты не будут;
* незначительные сдвиги границ целей вызывают значительные изменения сроков и бюджета проекта;
* то что в целях не прописано (забыли прописать или неправильно поняли), то это обязательно выпадет из рассмотрения и выполнено не будет)
  1. **Внешнее проектирование системы**— это про­цесс описания внешних функций проекта и ожидаемого пове­дения разрабатываемого продукта с точки зрения внешнего по отношению к нему наблюдателя-пользователя. Цель – конструирование и описания внешних взаимодействий без конкретизации внутреннего устройства. Результаты выражаются в форме

внешних спецификаций. Важно соблюдать концептуальную целостность продукта, который

предпочитает максимально соответствовать между внешними функциями системы.

Ответственные за внешние п. – системные аналитики и специалисты по GUI.

Внешнее проектирование мало связано с программированием. Больше касается проблем и

нужд пользователя и психологии общения человека с машиной.

**Существует 2 этапа:**

предварительный и детальное внеш.проектирование.

**Предварительный**: определяет все функции пользователя без определения синтаксиса и

семантики.

**Детальное**: описывает входные/выходные данные, преобразование системы; характеристики

надежности, эффективность и замечания по программированию.

**Входные данные**: точное описание синтаксиса и семантики: формат, допуст. значения, обл.

изменения.

**Выходные данные**: точное описание всех результатов ф –ии: реакция сообщение об ошибке и

т.д.

Должна описываться функц. связь между вх/вых данными. Должны быть описаны результаты

всех не правильных вх. данных. Изменение состояния системы – многие ф –ии могут сие

осуществлять. Это должно описано с точки зрения пользователя.

Характеристики надежности – описание воздействия всевозможных отказов ф-ии на систему и

ее файлы.

Эффективность ф-ии – затрачиваемое время и память.

Замечания по прогр. – описание идей от-но устр-ва ф-ии. Проверка правильности имеет

важное значение, т.к. в.п. – поворотный этап проекта.

* 1. **Разработка архитектуры системы**

Во время анализа требований к системе основное внимание уделялось выяснению того, что должно быть сделано, вне зависимости от того, как это сделать. На этапе разработки системы решается вопрос, как реализовать решения, принятые на этапе анализа.

Сначала разрабатывается общая структура (архитектура) системы. Архитектура системы определяет ее разбиение на модули, задает контекст, в рамках которого принимаются проектные решения на следующих этапах разработки. Приняв решения о структуре системы в целом, разработчик системы производит ее разбиение на относительно независимые в реализации части (модули), разделяя разработку между разработчиками выделенных модулей, что дает возможность расширить фронт работ, подключить к разработке системы новых исполнителей.

На этапе конструирования системы ее разработчик должен принять следующие решения:

* определить разбиение системы на модули;
* выявить асинхронный параллелизм в системе;
* определить состав вычислительного комплекса, на котором будет работать система;
* распределить компоненты системы по процессорам вычислительного комплекса и независимым задачам;
* организовать управление хранилищами данных;
* организовать управление глобальными ресурсами;
* выбрать принципы реализации управления программным обеспечением;
* организовать управление пограничными ситуациями.
  1. **Проектирование структуры программы**. Сегодня процесс создания сложных программных приложений невозможно представить без разделения на этапы жизненного цикла. Под жизненным циклом программы будем понимать совокупность этапов:

Анализ предметной области и создание ТЗ (взаимодействия с заказчиком)

Проектирование структуры программы

Кодирование (набор программного кода согласно проектной документации)

Тестирование и отладка

Внедрение программы

Сопровождение программы

Утилизация

* 1. **Внешнее проектирование модулей** – процесс описания планируемого поведения ПИ с точки зрения потенциальных пользователей.

В процессе внешнего проектирования модулей разрабатываются внешние взаимосвязи модулей, которые представляют собой внешние спецификации каждого модуля (предназначенные для различных групп специалистов, как пользователей, так и разработчиков) и необходимы другим модулям, осуществляющим обращение к данному модулю. Внешняя спецификация модуля не должна содержать никакой информации о внутреннем устройстве модуля, об особенностях реализованного в нем алгоритма. Кроме того, недопустимо, чтобы спецификация содержала какие-либо ссылки на вызывающие модули или на контексты, в которых этот модуль используется.

* 1. **Проектирование логики модуля**. Внутреннее проектирование — один из последних этапов в длинной це­пи процесса проектирования программного обес­печения. Оно представляет собой подробное внутреннее конст­руирование программного продукта, разработку внутренней логики каждого модуля системы, которая затем выражается текстом конкретной программы.
  2. **Пошаговая детализация** представляет собой простой процесс, предполагающий первоначальное выражение логики модуля в терминах гипотетического языка очень высокого уровня с последующей детализацией каждого предложения в терминах языка более низкого уровня, до тех пор пока наконец не будет достигнут уровень используемого языка программирования. На протяжении всего процесса логика выражается основными конструкциями структурного программирования.
  3. **Объе́ктно-ориенти́рованное проектирование** (ООП) — это часть [объектно-ориентированной](http://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F&action=edit&redlink=1) [методологии](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F), которая предоставляет возможность [программистам](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) оперировать [понятием](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BD%D1%8F%D1%82%D0%B8%D0%B5) «[*объект*](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))», нежели понятием «[*процедура*](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)» при разработке своего [кода](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4). Объекты содержат [инкапсулированные](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))  [данные](http://ru.wikipedia.org/wiki/%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5)  и  [процедуры](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0), [сгруппированные](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) вместе, отображая т.о. сущность объекта. «[Интерфейс объекта](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))», описывает взаимодействие с объектом, то, как он определен. [Программа](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), полученная при реализации объектно-ориентированного исходного кода, описывает взаимодействие этих объектов.
  4. –
  5. **Автома́тное программи́рование** — это [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), при использовании которой программа или её фрагмент осмысливается как модель какого-либо формального [автомата](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%BE%D0%B2).

В зависимости от конкретной задачи в автоматном программировании могут использоваться как [конечные автоматы](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82), так и автоматы более сложной структуры.

Определяющими для автоматного программирования являются следующие особенности:

* временной период выполнения программы разбивается на *шаги автомата*, каждый из которых представляет собой выполнение определённой (одной и той же для каждого шага) секции кода с единственной точкой входа; такая секция может быть оформлена, например, в виде отдельной функции и может быть разделена на подсекции, соответствующие отдельным состояниям или категориям состояний
* передача информации между шагами автомата осуществляется только через явно обозначенное множество переменных, называемых *состоянием автомата*; между шагами автомата программа (или её часть, оформленная в автоматном стиле) не может содержать неявных элементов состояния, таких как значения локальных переменных в стеке, адреса возврата из функций, значение текущего счётчика команд и т. п.; иначе говоря, состояние программы на любые два момента входа в *шаг автомата* могут различаться между собой только значениями переменных, составляющих *состояние автомата* (причём такие переменные должны быть явно обозначены в качестве таковых).

Полностью выполнение кода в автоматном стиле представляет собой цикл (возможно, неявный) шагов автомата.

Название автоматное программирование оправдывается ещё и тем, что стиль мышления (восприятия процесса исполнения) при программировании в этой технике практически точно воспроизводит стиль мышления при составлении формальных автоматов (таких как машина Тьюринга, автомат Маркова и др.)

**20. Домен** -- четко определенные предметные области. Домен – реальный гипотетически или

абстрактный мир, населенный отчетливым набором объектов, которые ведут себя в соответствии с

правилами поведения в этом домене. Домен расссм., как единица для цели анализа.

Схема доменов:

Это деление по принципу уровней абстракцией Дейкстры. Уровни: 1 – прикладной, 2 – сервисный

подсистемы, 3 – архитектурный, 4 – реализация.

1 – поддержка общих механизмов и сервисных ф – ий для прикладного.

3 –обеспечивает общ. механизмы управления данными и системой как единым целым. В нем

содержаться интерфейсы.

**21. Информационная модель**- это совокупность диаграмм типа "**сущность - связь**". Модели "**сущность - связь**" определяют структуру и взаимные связи используемой в системе информации, в полной мере отображающей её работу.

Цель информационной модели заключается в выработке непротиворечивой интерпретации данных и взаимосвязей между ними, что необходимо для интеграции, совместного использования и управления целостностью данных.

**Модель процесса проектирования** описывает трехфазный, ориентированный на конечного пользователя, непрерывный процесс разработки

В соответствии с первым принципом разработка системы выполняется в виде нескольких краткосрочных мини-проектов фиксированной деятельности от 2 до 6 недель, соответственно называемых итерациями. Каждая итерация включает свои собственные этапы анализа требований, проектирования, реализации, тестирования, интеграции и завершается созданием рабочей системы. Итерационный цикл основывается на постоянном расширении и дополнении системы с периодической обратной связью и адаптации дополняемых модулей к ядру системы. Согласно RUP жизненный цикл ПО разбивается на отдельные циклы, в каждом из которых создаётся новое поколение продуктов. Каждый цикл в свою очередь разбивается на 4 стадии:

начальная стадия – inception;

стадия разработки – elaboration;

конструирования – construction;

ввод в действие – transition.

**22. Методы тестирования программного обеспечения.**

Существует несколько методов тестирования:

Тестирование программ методом "чёрного ящика" (Black box testing)

Тестирование софта методом "белого ящика" (White box)

Тестирование ПО методом "серого ящика" (Grey box)

Тестирование не функциональных аспектов программы.

**Тестирование программы как "белого ящика" и "чёрного ящика"**

В терминологии профессионалов тестирования (программного и некоторого аппаратного обеспечения) фразы "тестирование белого ящика" и "тестирование черного ящика" относятся к тому, имеет ли разработчик тестов и тестирощик доступ к исходному коду тестируемого ПО, или же тестирование выполняется через пользовательский интерфейс либо прикладной программный интерфейс, предоставленный тестируемым модулем.

При тестировании белого ящика (англ. white-box testing, также говорят — прозрачного ящика), разработчик теста имеет доступ к исходному коду и может писать код, который связан с библиотеками тестируемого ПО. Это типично для юнит-тестирования (англ. unit testing), при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции — работоспособны и устойчивы, до определенной степени.

При тестировании чёрного ящика (англ. black-box testing), тестировщик имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов, с уверенностью в том, все ли идет правильно, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши. Как правило, тестирование чёрного ящика ведётся с использованием спецификаций или иных документов, описывающих требования к системе.

Если «альфа-» и «бета-тестирование» относятся к стадиям до выпуска продукта (а также, неявно, к объёму тестирующего сообщества и ограничениям на методы тестирования), тестирование «белого ящика» и «черного ящика» имеет отношение к способам, которыми тестировщик достигает цели.

Бета-тестирование в целом ограничено техникой чёрного ящика (хотя постоянная часть тестировщиков обычно продолжает тестирование белого ящика параллельно бета-тестированию). Таким образом, термин «бета-тестирование» может указывать на состояние программы (ближе к выпуску чем «альфа»), или может указывать на некоторую группу тестировщиков и процесс, выполняемый этой группой. Итак, тестировщик может продолжать работу по тестированию белого ящика, хотя ПО уже «в бете» (стадия), но в этом случае он не является частью «бета-тестирования» (группы/процесса).

### Задачи и цели интеграционного тестирования

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, должно быть заключение о том, что эти модули являются внутренне непротиворечивыми и соответствуют требованиям. Однако отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей - тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое. Такое тестирование называют *интеграционным*. Его цель - удостовериться в корректности совместной работы *компонент* системы.

*Интеграционное тестирование* называют еще *тестированием архитектуры системы*. С одной стороны, это название обусловлено тем, что интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы - таким образом, интеграционные тесты проверяют *полноту*взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов - один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. Т.е., с этой точки зрения, интеграционные тесты проверяют *корректность* взаимодействия*компонент* системы.

 На этом шаге мы рассмотрим ***технологию восходящего и нисходящего тестирования***.

    При плановом подходе программа проверяется последовательно блок за блоком, причем если программа состоит из центрального блока, который проводит обращения к периферийным блокам, мало связанным друг с другом, то возможны следующие два основных подхода к контролю такой программы, два основных направления тестирования:

* от периферии к центру (***восходящее тестирование***) или, наоборот,
* от центра к периферии (***нисходящее тестирование***).

    При первом, ***восходящем тестировании***, применяемом обычно для небольших программ, сначала тестируют отдельные периферийные блоки, а затем переходят к тестированию центральной части, которая, разумеется, взаимодействует только с отлаженными уже блоками.

    При ***нисходящем тестировании***, используемом для достаточно больших программ, параллельно с контролем периферийных блоков (или даже до начала их контроля) производится и контроль центрального блока, выполняемого на компьютере совместно с имитаторами периферийных блоков, называемых ***заглушками***. В задачу имитаторов входит моделирование работы соответствующих блоков с целью поддержать функционирование центрального блока. Обычно заглушки выдают простейший результат, например константу и сообщение о факте своего участия в работе. Вместо постоянной величины на наиболее поздней стадии тестирования может выдаваться и случайная величина в требуемом диапазоне.

**23. Регрессио́нное тести́рование** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *regression testing*, от [лат.](http://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D1%82%D0%B8%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *regressio* — движение назад) — собирательное название для всех видов [тестирования программного обеспечения](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), направленных на обнаружение ошибок в уже протестированных участках [исходного кода](http://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4). Такие ошибки — когда после внесения изменений в программу перестает работать то, что должно было продолжать работать, — называют *регрессионными ошибками* ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *regression bugs*).

Регрессионное тестирование (по некоторым источникам) включает new bug-fix - проверка исправления вновь найденного дефекта, old bug-fix - проверка, что исправленный ранее и верифицированный дефект не воспроизводится в системе снова, а также side-effect - проверка того, что не нарушилась работоспособность работающей ранее функциональности, если ее код мог быть затронут при исправлении некоторых дефектов в другой функциональности. Обычно используемые методы регрессионного тестирования включают повторные прогоны предыдущих тестов, а также проверки, не попали ли регрессионные ошибки в очередную версию в результате слияния кода.

Регрессионное тестирование – основная работа тестировщика. Имеет 2 значения, объединенных

идеей повторного использования ранее разработанных тестов.

Задача регрессионного тестирования состоит в том, чтобы убедиться, что ошибка исправлена.

Проводится тест, обнаруживается ошибка. Проводится тот же тест, чтобы убедиться.

Еще одна задача – тестирование целостности программы, а не только исправление одной ошибки.

Убедиться, что исправление одного не испортит другое.

**24. Автоматизированное тестирование программного обеспечения** — часть процесса [тестирования](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) на этапе [контроля качества](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%BE_%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%86%D0%B8%D0%B8) в процессе разработки [программного обеспечения](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5). Оно использует программные средства для выполнения тестов и проверки результатов выполнения, что помогает сократить время тестирования и упростить его процесс. Существует два основных подхода к автоматизации тестирования: тестирование на уровне кода и [тестирование пользовательского интерфейса](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D0%BA%D0%BE%D0%B3%D0%BE_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D0%B0) (в частности, GUI-тестирование). К первому типу относится, в частности, [модульное тестирование](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). Ко второму — имитация действий [пользователя](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C) с помощью специальных тестовых [фреймворков](http://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA" \o "Фреймворк).

**GUI-автоматизация**

Наиболее распространенной формой автоматизации является тестирование приложений через графический пользовательский интерфейс (англ. GUI). Популярность такого вида тестирования объясняется двумя факторами: во-первых, приложение тестируется тем же способом, которым его будет использовать человек, во-вторых, можно тестировать приложение, не имея при этом доступа к исходному коду.

GUI-автоматизация развивалась в течение 4 поколений инструментов и техник:

Утилиты записи и воспроизведения (англ. capture/playback tools) записывают действия тестировщика во время ручного тестирования. Они позволяют выполнять тесты без прямого участия человека в течение продолжительного времени, значительно увеличивая продуктивность и устраняя «тупое» повторение однообразных действий во время ручного тестирования. В то же время, любое малое изменение тестируемого ПО требует перезаписи ручных тестов. Поэтому это первое поколение инструментов не эффективно и не масштабируемо.

Написание сценария (англ. scripting) — форма программирования на языках, специально разработанных для автоматизации тестирования ПО — смягчает многие проблемы инструментов записи и воспроизведения. Но разработкой занимаются программисты высокого уровня, которые работают отдельно от тестировщиков, непосредственно запускающих тесты. К тому же скрипты более всего подходят для тестирования GUI и не могут быть внедренными, пакетными или вообще каким-либо образом объединены в систему. Наконец, изменения в тестируемом ПО требуют сложных изменений в соответствующих скриптах, и поддержка все возрастающей библиотеки тестирующих скриптов становится в конце концов непреодолимой задачей.

Управляемое данными тестирование (англ. Data-driven testing) — методология, которая используется в автоматизации тестирования. Особенностью является то, что тестовые скрипты выполняются и верифицируются на основе данных, которые хранятся в центральном хранилище данных или базе данных. Роль базы данных могут выполнять ODBC-ресурсы, csv или xls файлы и т. д. Управляемое данными тестирование — это объединение нескольких взаимодействующих тестовых скриптов и их источников данных в фреймворк, используемый в методологии. В этом фреймворке переменные используются как для входных значений, так и для выходных проверочных значений: в тестовом скрипте обычно закодированы навигация по приложению, чтение источников данных, ведение логов тестирования. Таким образом, логика, которая будет выполнена в скрипте, также зависит от данных.

Тестирование по ключевым словам (англ. Keyword-based) автоматизация подразумевает разделение процесса создания кейсов на 2 этапа: этап планирования и этап реализации. В этом случае конечный тест представляет собой не программный код, а описание последовательности действий с их параметрами (например, «завести в базе данных пользователя с логином XXX и паролем YYY»). При этом фреймворк отвечает за непосредственную реализацию ключевых слов (действий), а дизайнеру тестов достаточно иметь представление о всём наборе действий, реализованных во фреймворке. Это даёт возможность создавать тесты людям, не имеющим навыков программирования.

**JUnit** — библиотека для модульного тестирования программного обеспечения на языке Java.

Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений — JMock, EasyMock, DbUnit, HttpUnit и т. д.

**Log4j** — библиотека журналирования Java программ, часть общего проекта «Apache Logging Project».

Log4j первоначально развивался в рамках зонтичного «Apache Jakarta Project», ответственного за все Java-проекты Apache, но впоследствии выделился в отдельный, очень популярный проект журналирования.

**Apache Ant** (англ. ant — муравей и акроним — «Another Neat Tool») — утилита для автоматизации процесса сборки программного продукта. Является платформонезависимым аналогом утилиты make, где все команды записываются в XML-формате.

Ant был создан в рамках проекта Jakarta, сегодня — самостоятельный проект первого уровня Apache Software Foundation.

Первая версия была разработана инженером Sun Microsystems Джеймсом Дэвидсоном (James Davidson (англ.)русск.), который нуждался в утилите, подобной make, разрабатывая первую референтную реализацию J2EE.

Ant, в отличие от другого сборщика проектов Apache Maven, обеспечивает императивную, а не декларативную сборку проекта.

**Apache Maven** — фреймворк для автоматизации сборки проектов, специфицированных на XML-языке POM[2] (англ. Project Object Model).

Maven, в отличие от другого сборщика проектов Apache Ant, обеспечивает декларативную, а не императивную сборку проекта. То есть, в файлах проекта pom.xml содержится его декларативное описание, а не отдельные команды. Все задачи по обработке файлов Maven выполняет через плагины.

**Subversion[3] (также известная как «SVN»[4])** — свободная централизованная система управления версиями, официально выпущенная в 2004 году компанией CollabNet[en].

Цель проекта — заменить[5][6] собой распространенную на тот момент систему Concurrent Versions System (CVS), которая ныне считается устаревшей[7][8][9]. Subversion реализует все основные функции CVS и свободна от ряда недостатков последней.

Subversion используется многими сообществами разработчиков открытого программного обеспечения (в том числе сообществами, ранее использовавшими CVS). В их числе такие известные проекты, как Apache, GCC, Free Pascal, Python, Ruby, FreeBSD, AROS, Blender, Boost, Tor. Subversion также широко используется в закрытых проектах и корпоративной сфере. Хостинг Subversion, в том числе для проектов с открытым кодом, также предоставляют популярные хостинг-проекты SourceForge.net, Tigris.org, Google Code и BountySource.

В 2007 году аналитическая компания Forrester, сравнивая преимущества и недостатки различных систем, оценила Subversion как «единоличного лидера в категории Standalone Software Configuration Management (SCM) и сильного участника в категории Software Configuration and Change Management (SCCM)».[10]

По данным статистики использования пакетов Linux-дистрибутивов Debian[11] и Ubuntu[12], количество активных пользователей Subversion примерно такое же, как у Git, и превосходит аналогичный показатель для CVS, Mercurial и Bazaar (по состоянию на июнь 2011 года).

В качестве официальной документации позиционируется[13] книга издательства O’Reilly Media, выложенная в свободный доступ[14] и дописываемая авторами по мере выхода новых версий SVN. Там же публикуются её переводы на ряд языков, в том числе русский, но при том, что англоязычные версии книги сейчас описывают версии 1.6 и 1.5, на русском языке имеются лишь книги, описывающие версии до 1.4 включительно.

**Git** (произн. «гит»[3]) — распределённая система управления версиями файлов. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано.

Примерами проектов, использующих Git, являются ядро Linux, Android, Drupal, Cairo, GNU Core Utilities, Mesa, Wine, Chromium, Compiz Fusion, FlightGear, jQuery, PHP, NASM, MediaWiki, DokuWiki, Qt и некоторые дистрибутивы Linux (см. ниже).

Программа является свободной и выпущена под лицензией GNU GPL версии 2.

**Шлеер – Меллер**:

деление объектов на пассивные (для хранения данных) и активные(облад. сложным поведением).

Активные описываются автоматом Мура. Их метод состоит из трех этапов:

1. Создание информационных моделей

2. Создание моделей состояния

3. Создание моделей процессов

1 – абстрагирование концептуальных сущностей в терминах объектов и атрибутов. Отношения

между сущностями формализуются в связи, как основа линии поведения, законах и правилах

реального мира. Результат – построение диаграммы классов для подсистем.

2 – на диаграмме классов выделяют активные объекты и для них строиться автомат Мура. Для

каждой модели переходов состояний все описывается(действия).