***Инженерия программного обеспечения.*** 1968 — software engineering. Было вызвано резким скачком компьютерной техники, резкое повышение сложности программных продуктов. Следовательно, неформальный подход к разработке программных продуктов больше не работал. Появились такие задачи как проектирование программного продукта, ­на примере ООП знаем, что такое проектирование. Появилось управление командой программистов (разработчиков), формальные процедуры тестирования программного кода и прочие, как технические, так и организационные задачи. Под инженерией ПО понимается дисциплина, обхватывающая все аспекты программных продуктов.

**Программные продукты можно разделить на две основные группы.**

**1.** Первая группа охватывает общие программные продукты, это программные продукты, которые созданы компанией разработчиков ПО и свободно доступны на рынке за деньги либо по другой лицензии. Например — ОС, текстовые редакторы и так далее.

**2.** Вторая группа — это программные продукты, которые разработаны на заказ. В основном для решений узких задач конкретного заказчика.

**Модели**

**Модель** — это упрощенное представление реального процесса либо объекта, они нужны для того, так как сложный процесс изучать сложно или невозможно, а модель вполне возможно.

**Модели технологического процесса создания программных продуктов:**

**1.** ***Модели последовательности работ,*** такая модель показывает последовательность этапов, выполняемых при создании программного продукта, зависимость между этими этапами. Каждый этап — это определенный вид работы;

**2.** ***Модель потоков, данных и процессов***. В этом случае процесс разработки программы представляет из себя набор активностей, каждая из которых выполняет преобразование данных;

**3.** ***Ролевая модель***. В этом случае рассматриваются роли участников в проекте, действия, выполняемые разработчиком.

**Показатели качества программного продукта:**

**1.** Удобство сопровождения — качественный программный продукт легко модернизировать;

**2.** Надёжность**;**

**3.** Эффективность — рациональный расход ресурсов, скорость работы;

**4.** Удобство использования;

**5.** Объём документаци**и.**

**СИСТЕМОТЕХНИКА ИНФОРМАЦИОННЫХ СИСТЕМ.**

**Системотехника** — это технология создания систем, которая охватывает процессы внедрения, тестирования, проектирования, сборки и разработки как единого целого.

**Система** — это совокупность подсистем и компонентов, работающих совместно при достижении некоторой общей цели. Ключевым признаком системы является сложности связи между подсистемами и компонентами.

**Подсистема** — это система, входящая в состав другой системы, ключевая особенность подсистемы от компонента — это относительная самостоятельность, относительная автономность работы.

**Компоненты** — это сравнительно небольшие части системы, которые в своей работе тесно взаимодействуют с другими компонентами.

В результате взаимодействия подсистем и компонентов возникают ***интеграционные свойства*** системы, которыми не обладал ни один из компонентов в отдельности, система не сводится к простой сумме своих компонентов, она является некоторым более сложным, более качественным объектом, обладающим новыми свойствами.

**В целом** **интеграционные свойства можно разбить на две группы**:

***Первая группа.*** Функциональные свойства, которые проявляются во время работы системы, то есть это те функции, которые выполняет наш программный продукт как единое целое;

***Вторая группа.*** Нефункциональные свойства, их ещё можно назвать характеристиками системы, это та самая безотказность, производительность, защищённость и прочее.

**На разработку системы влияет ряд как технических, так и ряд совершенно не технических факторов:**

**Эксплуатационный фактор** — ваш программный продукт надо будет внедрять в фирме заказчика, а они жили и прекрасно без вашего продукта, появляется вопрос, нужно ли при внедрении вашего продукта менять производственные процессы, как внедрение системы повлияет на работы фирмы в целом;

**Фактор персонала** — может возникнуть ситуация, когда внедрение вашего продукта снижает требования квалификации сотрудников, ведь мы стараемся всё автоматизировать;

**Организационный фактор —** проблема заключается в том, что внедрение нового программного продукта может изменить структуру полномочий в организации. Пример: есть деканат, который занимается как бы кучей важных дел, мы автоматизируем их часть работы, после этого часть сотрудников теряет свою важность.

**Процесс создания системы:**

***Первый этап*** — определение системных требований.

***Второй этап*** — проектирование системы, на основании спецификации требований определяются свойства системных компонентов.

***Третий этап*** — разработка подсистем, разрабатываются те компоненты, которые были спроектированы на предыдущем этапе.

***Четвёртый этап*** — сборка системы. Это процесс интеграции независимо разработанных подсистемы в единую систему. Есть два основных подхода к сборке системы: это метод большого взрыва, в этом случае все компоненты интегрируются одновременно, основное преимущество — это скорость; второй подход: инкрементная или же последовательная интеграция, в этом случае компоненты добавляются по одному, в этом случае легко найти ошибки, недостаток в том, что это длится дольше.

***Пятый этап*** — инсталляция подсистемы. ***На этом этапе необходимо развернуть программный продукт у заказчика, возникает много интересных моментов***:

— Окружение для которого разработана система может не совпадать с тем окружением, в котором требуется её эксплуатировать, под окружением понимается аппаратное обеспечение, версии ОС и тому подобное.

— Потенциальные пользователи могут враждебно относиться к вашему программному продукту.

— Ваш программный продукт должен работать с теми программами, которые использовались ранее.

— Ввод системы в эксплуатацию. После того как система установлена, когда её можно физически запустить, нужно ввести программный продут в эксплуатацию, то есть нужно обучить оператора (пользователей системы);

***Шестой этап*** — включение системы в бизнес процесс предприятия / организации. На этом этапе могут возникнуть следующие вопросы и трудности, связанные с тем, что спецификация требования была не совсем полной, всё было невозможно предусмотреть. Оказывается, что программный продукт не подходит для использования в этой фирме.

***Седьмой этап*** — сопровождение системы: исправление ошибок, которые неминуемо будут возникать*;* модификация программного продукта.

***Восьмой этап*** — вывод системы из эксплуатации. Программные продукты имеют ограниченный срок жизни. Главный момент, на который стоит обратить внимание — это сохранение данных.

**Классические модели создания программных продуктов.**

**Каскадная модель** — она же водопадная модель. Схематично эту модель можно представить в виде каскадов. Этапы: определение требований; проектирование системы; кодирование и тестирование программных модулей; сборка и тестирование системы; эксплуатация и сопровождение.



Эту модель называют ещё жизненным циклом программного продукта.

**Преимущества каскадной модель**: если требования к программному продукты сформулированы точно и сразу, то тогда каскадная модель оказывается самой эффективной. Каскадная модель предполагает переход к следующему этапу после полного окончания предыдущего этапа и документирования результата.

**Основой недостаток в следующем**: если на одном из этапов требования заказчиков меняются, то в соответствии с каскадной модели необходимо вернуться к первому этапу и повторить весь процесс разработки с самого начала. *Для того чтобы избежать полного перепроектирования системы применяется следующий приём*: начиная с некоторого этапа все предыдущие этапы замораживаются и любые новые требования заказчика реализуются за счёт программистских хитростей (без проектирования). Ещё один недостаток этой модели: негибкое разбиение процесса на отдельные этапы.

**Вывод**: каскадную модель целесообразно использовать если требования заданы чётко изначально и не меняются.

**Эволюционная модель** — сначала разрабатывается первая версия программного продукта основываясь на требованиях заказчика, потом эта версия передаётся заказчику для испытаний, он смотрит и уточняет требования, получив их, разрабатываем следующую версию на основе новых требований.



Всё начинается с эскизного описания, потом идут три типа процессов: спецификация требований; разработка; аттестация. Между этапами идёт постоянный обмен информации, то есть процессы параллельны. В результате получается начальная версия, какое-то множество промежуточных версий, ну и некоторая финальная версия.

**Отличительной особенностью эволюционной моделью** является то, что процесс разработки, создания требований и аттестации протекают параллельно и между ними происходит постоянный обмен информации. Процесс начинается с эскизного описания проекта, на основе которого разрабатывается начальная версия продукта, эта начальная версия отдаётся заказчику для аттестации, по результатам аттестации уточняются требования, разрабатывается следующая версия и так далее.

**Эта модель предполагает два основных подхода:**

***Первый подход*** — это метод пробных разработок, в этом случае происходит постоянное взаимодействие с заказчиком, для того чтобы получить полную и окончательную версию требований программного продукта, на начальных этапах в этом случае разрабатывают те модули и те части системы, требования к которым понятны и не требуют уточнения, в последствии система эволюционирует, до тех пор, пока не будет создана окончательная версия;

***Второй подход*** — это метод прототипирования, он предлагает следующее решение, где на основе мутных требований заказчика создаётся прототип, после этого отдаём продукт заказчику для опытов.

Эволюционный подход работает намного лучше каскадной модели, если требования заказчика меняются.

**Недостаток подхода**: многие этапы остаются без документации, потому что нет смысла тратить на документирование этапов, например, промежуточную версию программы, так как она будет меняться; поскольку отсутствует изначально чёткий проект, система получается плохо структурированной; ещё одно ограничение — этот подход применим для разработки малых и средних систем с коротким сроком жизни, для больших систем и долгоживущих слишком влияют минусы этого подхода.

Возможно применение гибридных моделей, совмещающих преимущества каскадных и эволюционных. В этом случае места известные из требований разрабатываются по каскадной модели, а темные места по эволюционной.

**Формальная разработка систем —** основная идея заключается в следующем: требования к программному продукту описываются с помощью специальных математических нотаций, математических обозначений, затем эти формальные требования автоматически преобразуются в текст программы.

**Этапы следующие:** определение требований, формальная спецификация, формальные преобразования, сборка и тестирование системы.



**Особенности этой модели:** требования определяются с помощью специальных математических обозначений, вторая особенность заключается в том, что процессы проектирования и кодирования заменяются формальными математическими преобразованиями. Здесь мы не пишем текст программы.

**Преимущества в том**, что самого процесса разработки нет, он заменен формальными спецификациями. Основное преимущество следует из этого: однозначное соответствие программного кода требованиям заказчика, следовательно, тестировать эти модули не надо; экономится время, так как не надо программировать.

**Эта модель имеет ограничения**: она применима для очень узкого класса программ, в общем случае требования программного продукта невозможно чётко формализовать. Разработка спецификации может оказаться слишком трудоемкой.

**Разработка на основе ранее созданных компонентов** — очевидно, что программы каждый раз с нуля не разрабатываются. Часть модулей находится в готовом варианте, либо адаптируются, при этом довольно заметно меняется жизненный цикл программы.

**Этапы:** спецификация требований; выполняется анализ компонентов; модификация требований; проектирование системы с учётом тех компонентов, которые выбраны; разработка и сборка; аттестация.



***Теперь подробней про каждый из этапов.***

— Про первый этап и так понятно.

— Анализ требований: поиск компонентов, которые этим требованиям более или менее отвечают, основная особенность этого этапа в том, что обычно нет компонентов, которые однозначно соответствуют нашим требованиям.

— Модификация требований: анализируются изначальные требования с учётом возможностей выбранных компонентов и изначальные требования модифицируются таким образом, дабы использовать максимум возможности найденных компонентов, если же модификация требований невозможна, то возвращаемся к предыдущему этапу и выбираем другие компоненты.

— Проектирование системы: либо проектируется система новой системы, либо модифицируется уже готовая система, в общем, используется максимум готовых компонентов, а тех что нет — пишем сами.

— Следующие этапы и так понятны, без комментариев.

**Основное преимущество этого подхода**: экономия времени.

**Недостатки этого подхода**: компоненты как правило не имеют полного соответствия требованиям нашего программного продукта, поэтому модификация требований — это неизбежный этап, следовательно, заказчик получает не тот продукт, который заказывал; существенно усложняется, часто становится невозможна модификация системного продукта из-за отсутствия контроля над выпуском новых версий компонентов.

**Вывод**: метод разработки на основе ранее созданных компонентов подходит для тех случаев, когда необходим быстрый выпуск программного продукта и не предвидится существенных модификаций программы.

# **Итерационные модели разработки программных продуктов**

Рассмотренные выше классические методологии имеют свои достоинства и недостатки, но в общем случая использовать их изолированно не получается. Поэтому применяются гибридные модели. В этом случае процесс разработки организуется в итерации. На каждой итерации может применятся та или иная модель.

**Особенность гибридных-итерационных моделей** — процесс разработки спецификаций идет параллельно с процессом разработки самого продукта. Полную спецификацию можно получить только после окончания разработки. Это делает невозможным использования реляционной модели для заказчика.

## **Модель пошаговой разработки**



Спецификация разрабатывается параллельно от разработки продукта. Это позволяет избежать ошибки архитектуры системы.

**Этапы будут следующими**:

— Заказчик в общих чертах определяет те сервисы, которые ему нужно. Каждому требованию присваивается приоритет.

— Определяется количество шагов разработки, таким образом, чтобы по окончании каждого шага реализовывался один системный компонент, который реализует некоторое подмножество функций.

— После завершения шага разработки компонент передается для аттестации заказчику. Компонент интегрируется с ранее созданными компонентами, заказчик проводит испытание программного продукта. По результатам испытания принимается решение: уточнить требования, добавить требования, завершить разработку компонента.

— При это на каждом шаге разработки может применятся своя модель. Если требования определенны четко — то каскадная модель. Если требования размыты — то модель прототипирования.

**Достоинства этой модели:**

**—** Заказчику не нужно ждать полного завершения разработки для того чтобы понять, как работает продукт.

— Наиболее важные функции реализуются на первых шагах.

— Компоненты, созданные на первых этапах заказчик может использовать в виде прототипов, проводить над ними опыты и вводить поправки в требования к продукту, на более поздних этапах.

— Модель снижает риск общих системных ошибок.

— Наиболее приоритетные требования реализуются первыми. Все остальные компоненты с ними интегрируется. Сборка системы проводится с тестированием. Это позволяет много раз тестировать базовые компоненты.

### **Недостатки модели:**

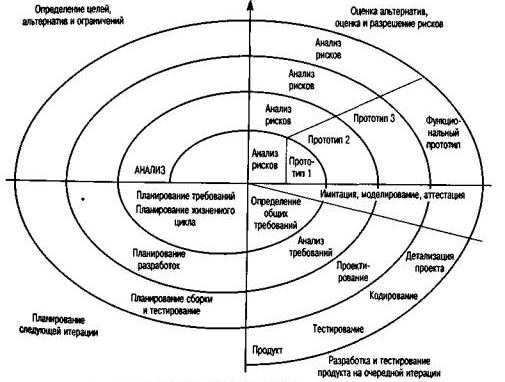
— Могут быть системные сервисы, которые не помещаются в рамки одно компонента.

— Модель пошаговой разработки применяется к сравнительно не большим продуктам. И на каждом шаге реализуется не большой компонент, из-за сжатых сроков.

— Если не известны некоторые требования заранее, возникают проблемы организации общей системы.

## **Спиральная модель**

Весь процесс разбивается на итерацию. Каждый виток спирали – итерация.



**Каждый виток спирали разбит на четыре сектора:**

**Первый.** Определение целей*.* Определяются цели каждой итерации проекта. Кроме того, устанавливаются ограничения на процесс создания ПО и на сам программный продукт, уточняются планы производства компонентов. Определяются проектные риски (например, риск превышения сроков или риск превышения стоимости проекта). В зависимости от "проявленных" рисков, могут планироваться альтернативные стратегии разработки ПО.

**Второй.** Оценка и разрешение рисков. Для каждого определенного проектного риска проводится его детальный анализ. Планируются мероприятия для уменьшения (разрешения рисков). Например, если существует риск, что системные требования определены неверно, планируется разработать прототип системы.

**Третий.** Разработка и тестирование. После оценки рисков выбирается модель процесса создания системы. Например, сети доминируют риски, связанные с разработкой интерфейсов, наиболее подходящей будет эволюционная модель разработки прототипированием. Если основные риски связаны с соответствием системы и спецификации, скорее всего, следует применить модель формальных преобразований. Каскадная модель может быть применена в том случае, если основные риски определены как ошибки, которые могут проявиться на этапе сборки системы.

**Четвёртый.** Планирование. Здесь пересматривается проект и принимается решение о том, начинать ли следующий виток спирали. Если принимается решение о продолжении проекта, разрабатывается план на следующую стадию проекта.

**Существенное отличие** спиральной модели от других моделей процесса создания ПО заключается в точном определении и оценивании рисков. Если говорить неформально, то риск — это те неприятности, которые могут случиться в процессе разработки системы. Например, если при написании программного кода используется новый язык программирования, то риск может заключаться в том, что компилятор этого языка может быть ненадежным или что результирующий код может быть недостаточно эффективным. Риски могут также заключаться в превышении сроков или стоимости проекта. Таким образом, уменьшение рисков — важный элемент управления системным проектом.

**Гибкие (Agile) методологии разработки программных продуктов**

Классические методологии характеризируются, во-первых, слишком большим объемом документации; процесс разработки — это жёсткая и неизменная структура, эти характеристики являются сдерживающими факторами при разработке маленьких программных продуктов, на смену приходят гибкие методологии.

**Предпосылки возникновения гибких методологий.**

***Первый этап***— code and fix: сначала пишем, потом отлаживаем, особенности заключаются в том, что нет никакого единого плана разработки; вместо единого проекта есть набор краткосрочных решений, минус в том, что такие продукты нельзя поддерживать и развивать, ещё одна особенность — слишком продолжительный период тестирования и отладки, это приводит к нарушению планки выпуска продукта.

***Второй этап*** — классическая методология.

***Третий этап*** — гибкие методологии. Плюсы: меньшая ориентация на документацию продукта. Гибкие методологии являются адаптивными, вместо единой структуры применяются подстроенные под каждый проект методологии разработки. Гибкие методологии ориентированы на человека, а не на процесс.

**Особенности гибких методологий разработки программных продуктов.**

**Особенности управления непредсказуемым процессом.** Не нужно делать вид, что требования не меняются, нельзя пытаться втиснуть гибкий процесс в какую-нибудь жесткую методологию. Нужно постоянно владеть актуальной информацией, состоянием по делу на текущий момент времени, состояние наших дел.

Для того чтобы адаптировать процесс к меняющемся требованиям, его нужно организовать итеративно. На начало итерации фиксируется некоторый набор требований, в течении итерации новые требования не применяются, в течение короткого промежутка времени реализуется некоторый набор функций, разработанные в текущей итерации программные модули интегрируется с ранее созданными модулями, в итоге получается очередная версия системы, после проходит аттестация с заказчиком, процесс повторяется. Основной момент при этом — определение длительности итерации, в среднем длится около четырёх недель, итерация должна быть настолько короткой, насколько возможно в рамках текущего проекта.

**Адаптивный заказчик — это одна из особенностей гибких методологий.** Адаптивный процесс требует особых отношений с заказчиком. Если классический подход предполагает по сути дела покупку заказчиком программного продукта, то гибкие методологии говорят о том, что с заказчиком нужно дружить и постоянно взаимодействовать. Важен контроль с заказчиком каждой итерации, каждого этапа разработки, это существенно повышает качество программного продукта. Гибкие методологии предполагают постоянное взаимодействие с заказчиком, вплоть до того, что специалист по программной области принимается в команду.

**Особенности отношения к разработчику.** Классическую модель акцентуируют на процессе, а не на разработчике. Важный шаг, который сделан — люди-программисты поставлены на первое место. Смещаются аспекты в управлении проектом, то есть менеджер не должен учить программиста программировать, не менеджер говорит о сроках, а программист смотрит на требования и говорит, сколько его команде нужно времени — программисту делегируется часть ответственности по управлению проектом.

Гибкие методологии ориентированы на быструю разработку небольших программных продуктов. Как правило, создание таких продуктов предполагает использование небольшой команды разработчиков, каждый из которых обладает высокой квалификацией. Поэтому гибкая методология говорит, что к каждому разработчику нужно подходить индивидуально.

**Особенности управления процессом, ориентированным на разработчика.** Форма организации процесса должна приниматься разработчиком, а не навязываться вне. Все технические решения принимаются исключительно разработчиком без участия менеджера.

**Роль бизнес-консультантов**. Поскольку разработка программного продукта выполняется в предельно сжатые сроки, у разработчиков должна быть возможность оперативно получать консультации по предметной области. Поэтому гибкие методологии предполагают завлечение консультантов по предметной области в процесс разработки, в предельном случае предполагается что мы берём и принимаем в нашу команду консультанта по предметной области. Кроме того, постоянная вовлеченность консультанта в процессе разработки позволяет заказчику получать актуальную информацию о создаваемом программном продукте. То есть прямо сейчас, а не через год после окончания проекта.

**Адаптация адаптивного процесса.** Кроме адаптации программного продукта к меняющемся требованиям заказчика можно говорить об адаптации самого процесса разработки. К концу работы проекта форма организации разработки будет отличаться от изначального.

**Экстремальное программирование**



**1. Вся команда.** Все участники проекта работают как одна команда. В нее обязательно входит представитель заказчика, лучше, если это будет реальный конечный пользователь продукта, разбирающийся в бизнесе. Заказчик выдвигает требования к продукту и расставляет приоритеты в реализации функциональности. Со стороны исполнителей в команду входят разработчики и тестировщики.

**2. Игра в планирование.** *Планирование в ЭП проводят в два этапа — планирование релиза и планирование итераций.*

На *планировании релиза* команда программистов встречается с заказчиком, чтобы выяснить, какую функциональность он хочет получить к следующему релизу, то есть через 2-6 месяцев.

*Планирование итераций* проводится каждые две недели, иногда чаще или реже. Заказчик обязательно присутствует: он определяет функциональность на следующую итерацию и вносит изменения в требования к продукту.

**3. Частые релизы версий.** Версии выпускаются часто, но с небольшим функционалом. Во-первых, маленький объем функциональности легко тестировать и сохранять работоспособность всей системы. Во-вторых, каждую итерацию заказчик получает часть функционала, несущую бизнес-ценность.

**4. Коллективное владение кодом.** Любой разработчик может вносить изменения в любой фрагмент кода программы. За каждый участок кода отвечает, как минимум два разработчика. Повышает универсальность программистов.

**5. Парное программирование.** Представьте двух разработчиков за одним компьютером, работающих над одним куском функциональности продукта. Это и есть парное программирование, самая спорная практика XP. Старая поговорка «одна голова хорошо, а две лучше» отлично иллюстрирует суть подхода. Из двух вариантов решения проблемы выбирается лучший, код оптимизируется сразу же, ошибки отлавливаются еще до их совершения. В итоге имеем чистый код, в котором хорошо разбираются сразу двое разработчиков.

**6. Непрерывная интеграция.** Это значит, что новые части кода сразу же встраиваются в систему. Во-первых, сразу видно, как последние изменения влияют на систему. Если новый кусок кода что-то сломал, то ошибку найти и исправить в разы проще, чем спустя неделю. Во-вторых, команда всегда работает с последней версией системы.

**7. Стандарты кодирования.** Использование общего стандарта построения.

**8. Метафора системы.** Выступает в качестве аналога архитектуры системы. Основная цель заключается в том, чтобы каждый участник проекта понимал архитектуру, основные функции и назначение будущего программного продукта. При этом, текущий проект может описываться путём сравнения с ранее выполненными проектами.

**9. Рефакторинг.** Это процесс постоянного улучшения дизайна системы, чтобы привести его в соответствие новым требованиям. Рефакторинг включает удаление дублей кода, повышение связности и снижение сопряжения. ЭП предполагает постоянные рефакторинги, поэтому дизайн кода всегда остается простым.

**10. Простой дизайн.** Простой дизайн в ЭП означает делать только то, что нужно сейчас, не пытаясь угадать будущую функциональность. Простой дизайн и непрерывный рефакторинг дают синергетический эффект — когда код простой, его легко оптимизировать.

**11. Обязательная 40 часовая рабочая неделя:** 5 дневная неделя, 8 часов в сутки.

**Недостатки:**

1. Успех проекта зависит от вовлеченности заказчика, которой не так просто добиться

2. Трудно предугадать затраты времени на проект, т.к. в начале никто не знает полного списка требований

3. Успех ЭП сильно зависит от уровня программистов, методология работает только с senior специалистами

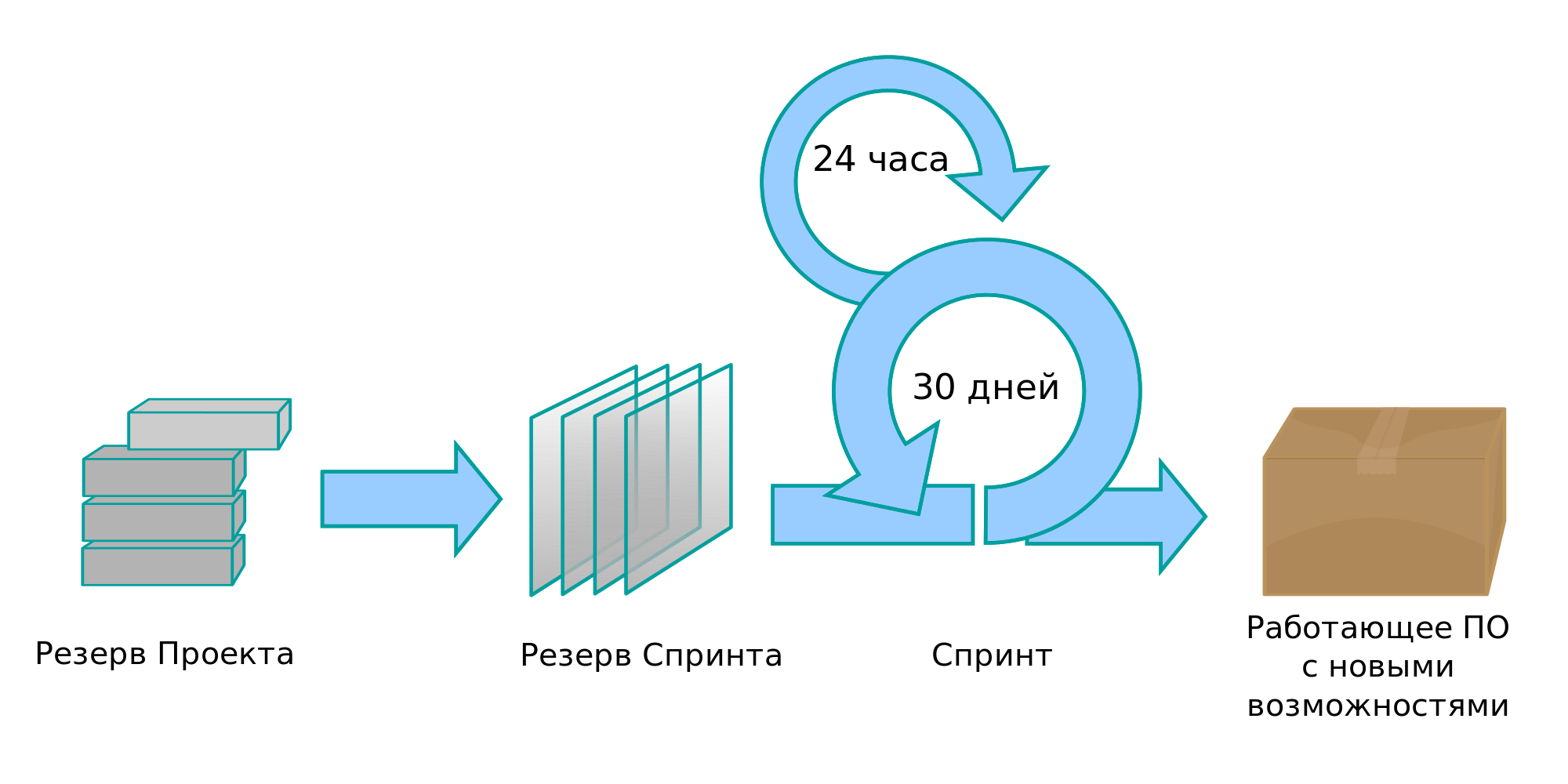
4. Менеджмент негативно относится к парному программированию, не понимая, почему он должен оплачивать двух программистов вместо одного

5. Регулярные встречи с программистами дорого обходятся заказчикам

6. Из-за недостатка структуры и документации не подходит для крупных проектов

**Методология SCRUM**

Является методологий программных продуктов, может использоваться для сопровождения и поддержки программного обеспеченья. Это набор принципов, который позволяет организовать процесс разработки таким образом, что заказчик получает относительно скоро (2-4 недели) очередную версию программного продукта с новой функциональностью, выбранным в соответствии с приоритетами. При этом функции, которые будут реализованы (sprint) фиксируются в начале и не меняются до конца спринта. Таким образом, Scrum борется с меняющимся требованием, так как спринт имеет фиксированную длину и позволяет планировать.



**Действующие роли в методологии Scrum.**

**А) Основные роли**. Свиньи — полностью увлечены в процесс.

**— Владелец проекта (product owner):** представляет интересы заказчика. Представитель заказчика либо менеджер проекта. Поддержка списков требований в актуальном состоянии. Выставлять приоритеты каждому из требований исходя из бизнеса целей проекта. Необязательно чтобы список требований был заполнен. В начале спринта из этого списка выбирает то подмножество, которое команда будет реализовывать на протяжении одной итерации (спринта). Эти требования разбиваются на подзадачи и делается точная оценка времени на реализацию.

**— Скрам-мастер (scrum master):** следит за выполнением всех принципов скрам, разрешает противоречия и защищает команду от отвлекающих факторов. Данная роль не предполагает ничего иного, кроме корректного ведения скрам-процесса. Руководитель проекта скорее относится к владельцу проекта и не должен фигурировать в качестве скрам-мастера.

**— Скрам-команда (scrum team):** кросс-функциональная команда разработчиков проекта, состоящая из специалистов разных профилей: тестировщиков, архитекторов, аналитиков, программистов и так далее. Размер команды в идеале составляет 7±2 человека. Команда является единственным полностью вовлечённым участником разработки и отвечает за результат как единое целое. Никто кроме команды не может вмешиваться в процесс разработки на протяжении спринта.

**Б) Дополнительные роли**. Куры — частичные сотрудники.

— Конечные пользователи программного продукта.

— Продавцы-маркетологи: лица, которые инициируют проект и для кого проект будет приносить выгоду. Они вовлечены в скрам только во время обзорного совещания по спринту (Sprint Review).

— Консультанты.

— Управляющие: люди, которые управляют персоналом.

## **Совещания (meeting)**

**Планирование спринта** — совещание, которое проводится в начале каждого спринта. Выбирается объем работ, который команда обязуются реализовать на протяжении спринта. Каждая задача делится на подзадачи и оценивается в человеко-часах.

**Ежедневное совещание** — в начале каждого рабочего дня проводится не большое 15 минутного совещания. Получение актуальной информации о разработке продукта. Четкое время проведение. Голос только есть у свиней. 3 вопроса:

— Что сделано с момента предыдущего собрания?

— Что будет сделано с сегодняшнего совещания и до следующего?

— Какие проблема мешают достижению спринта?

**Обзор итогов спринта** — совещание проводится по окончанию спринта. Команда демонстрирует инкремент функциональности всем заинтересованным лицам. Длительность ограничена 4-мя часами в зависимости от продолжительности итерации и инкремента продукта.

**Ретроспективное совещание** — все члены команды высказывают своё отношение о прошедшем спринте. Выполняют улучшение процесса разработки (решают вопросы и фиксируют удачные решения). Длительность ограничена 1—3-мя часами. Отвечают на два основных вопроса:

— Что было сделано хорошо в прошедшем спринте?

— Что надо улучшить в следующем?

**Семейство методологий CRYSTAL**

Алистр Коубер — разработчик этой методологии.

В это семейство входит ряд разных методологий, идея состоит в том, что для разных проектов должны использоваться разные модификации этой методологии.

Предполагается следующая градация: по одной оси откладывается количество разработчиков (N), а по второй критичность ошибок (E). Можно построить некоторую решётку, где каждой ячейке таблицы соответствует ячейка (вид) методологии.



Эти методологии больше ориентированы на человека, как и семейство гибких методологий. Основная идея в том, чтобы для каждой команды использовать минимально возможную степень дисциплины.

Решение о выборе той или иной дисциплины выбирается по критерию производительности (максимизируется). Не надо заставлять людей работать по плану (конкретному), лишь проект был закончен вовремя. Используется пересмотр процесса в конце каждой итерации (адаптация адаптивного процесса).

**OPEN SOURCE — открытое программное обеспечение**

Над подобными проектами работает большое количество физически удалённых разработчиков (это не стандартно, гибкие методологии обычно предполагают, что разработчики находятся близко).

Обычно есть несколько координаторов — они играют роль руководителя проекта. Любой из участников может вносить свои изменения исходного кода, но в финальную версию их вносит только координатор. Дополнение, изменение проекта обычно оформляется в виде патч-файлов.

**Роль координатора может быть организована по-разному:**

1. Один координатор на весь проект. Затем, могут быть
2. Отдельные координаторы на отдельные модули. Ещё один вариант:
3. Роль координатора может быть переходящей (улучшается уровень разработчиков).

Большинство участников таких проектов работает не полный рабочий день, а тратят просто своё свободное время.

Особенность отладки — происходит отдельно, может быть задействовано большое количество участников.

**Адаптивная разработка**

Джим Хайсимнг — основатель.

В основе этой методологии лежит три перекрывающихся нелинейные фазы: **обдумывание, сотрудничество и обучение**.

Планирование в адаптивном процессе, меняющемся процессе — это парадокс, так как результаты планирования непредвиденные. Если в обычном планировании отклонение от плана — это ошибки, которые нужно исправлять, то здесь же, отклонение от плана — это правильное проектное решение.

Для того чтобы преодолеть неопределенность адаптивного процесса, разработчики должны активно сотрудничать между собой. При этом задача руководства не указывать разработчику что ему делать, а в том, чтобы обеспечить возможность коммуникации между разработчиками. В результате такого общения, разработчики сами находят решения возникающих проблем.

В процессе работы над проектом должно происходить постоянное обучение всех участников проекта. Модификация процесса разработки происходит за счёт обучения разработчиков.

**Разработка, управляемая функциональностью**

Джеф Д. Люка и Питр Кодд — авторы методологии.

Упор делается на коротких итерациях, в результате каждой итерации разрабатывается часть системы, каждая итерация в среднем длится две недели.

**Включает пять этапов:**

1. Разработка общей модели;
2. Составление списка требуемых свойств системы;
3. Планирование работы над каждым свойством;
4. Проектирование каждого свойства;
5. Реализация каждого свойства;

**Все разработчики в соответствии с этой методологией делятся на два типа:**

— Владельцы классов.

— Старшие программисты. Именно им поручается реализация каких-то свойств системы. Самостоятельно они ничего не делают, а определяют, анализируют в какие классы нужно внести изменения и выдают задания владельцам этих классов. Старший программист выступает в роли координатора команды владельцев классов.

**ИТОГИ**:

* Гибкие методологии целесообразно использовать, когда предполагается изменение требований проекта, вторая причина: когда требования неизвестны изначально чётко.
* Гибкие методологии предполагают небольшую команду программистов (например, Crystal использовался в командах до 50 человек).
* Гибкие методологии можно использовать тогда, когда есть команда разработчиков высокой квалификации.
* Заказчик должен быть готов участвовать в процессе разработки.
* Заказчик должен быть готов к гибкой стоимости проекта.
* Какую бы мы не выбрали гибкую методологию, она нам всё равно не подойдёт, её всё равно придётся адаптировать под нашу команду и под наш проект в процессе работы.

**АНТИПАТТЕРНОЕ ПРОГРАММИРОВАНИЕ**

Примеры неудачных проектных решений (примеров программирования):

1. Программирование копипастом.

*Проблемы*: ошибки, связанные с недомодифицированостью программного кода.

*Решение*: ни один фрагмент кода не должен присутствовать в коде дважды.

1. Спагетти-код — использование goto, неструктурированная система.

*Причины*: лень, недостаток опыта в разработке, неэффективные code review, удалённая работа большого количества программистов.

1. «Золотой молоток» — мы изучали какой-то один приём, и начинаем его применять везде, к месту, не к месту, просто потому, что он нам нравится.
2. Магические числа — в программном коде иногда попадаются непонятные числа, которые никак не документированы.

*Пример*: z = x + y + 3. Откуда взялась тройка?

*Борьба*: все числа присваивать переменным с говорящими именами.

1. Жёсткое кодирование — в программный код внедряются данные о окружении программы (например, задание пути к файлу).
2. Мягкое кодирование — следствие параноидальной болезни жёсткого кодирования, в программе настраивается абсолютно всё. Излишние усложняется конфигурирование программ.
3. Избыточная сложность.

*Причина*: отсутствие code review.

1. Лодочный якорь. «Чемодан без ручки». В проекте сохранятся куски кода, которые стали ненужными после оптимизации или рефакторинга.
2. Изобретение велосипеда — реализация своего плохого решения, когда есть хорошие аналоги.
3. Поток лавы — на каком-либо этапе разработки вы можете осознать, что некоторая часть кода очень давно не менялась и вообще не документирована.
4. Программирование перебором— применение костылей, чтобы программа заработала здесь и сейчас.
5. Слепая вера — недостаточная проверка корректности входных данных.

*Решение*: всегда проверяйте корректность входных данных.

1. Бездумное комментирование — большое количество лишних и неинформативных комментариев.

*Решение*: код должен сам себя пояснить.

**ТРЕБОВАНИЯ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ**

При составлении требований необходимо учитывать различия между различными **группами требований**:

1. Пользовательские требования — вид требований, который разрабатывается первым.

Особенности: пользовательские требования описываются на естественном языке.

1. Системные требования — разрабатываются на основе пользовательских требований. Представляют собой детализированное формализованное представление пользовательский требований. Для записи системных требований могут применяться специальные системные обозначения, таблицы и другое.

Системные требования также называют системной спецификацией.

Системные требования являются основанием для заключения контракта на разработку программного продукта.

1. Проектная системная спецификация — разрабатывается на основе системных требований, представляет собой результаты ещё большей детализации.

**Предназначение требований:**

Пользовательские требования для:

— заказчика;

* менеджера компании-разработчика;

Системные требования для:

— менеджеров проекта;

* разработчиков системной архитектуры;
* технических специалистов компании-заказчика;

Проектная системная спецификация для:

— разработчиков системной архитектуры;

— разработчиков программного продукта;

**Другая классификация** **требования** делятся на:

1. Функциональные — перечень функций, которые должен выполнять программный продукт, с указанием ограниченных требований к входным данным и так далее. Здесь же может указываться, что программный продукт не должен делать. «Что делать?». «Что не делать?».
2. Нефункциональные — не касаются поведения системы, описывают характеристики системы (её интегральные свойства): производительность, отказоустойчивость.
3. Требования предметной области — описывают то окружение, в котором будет работать программный продукт.

**ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ**

*Проблема*: разность и нечёткость.

Если требования сформулированы неоднозначно, то разработчик их интерпретирует так, чтобы было легче реализовывать. Скорее всего, такая интерпретация требований не совпадёт с тем, что ожидал заказчик.

**Спецификация функциональных требований должна быть**:

1. Комплексной;
2. Непротиворечивой;

а) в техническом задании должны быть перечислены все сервисы (функции) будущей системы;

б) в техническом задании не должно содержаться взаимоисключающих требований;

На практике составить для большого проекта комплексную и непротиворечивую спецификацию требований практически невозможно.

**НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ**

Не касаются тех действий, которые система реализует, они описывают интегральные свойства системы:

— скорость работы;

* размер потребляемой памяти;

Нефункциональные требования могут быть предъявлены не только к программному продукту, но и к процессу разработки, окружению и так далее.



Требования к продукту описывают эксплуатационные свойства программного продукта.

Организационные требования отражают организационные процедуры заказчика и разработчика. Например, стандарты на разработку программного продукта, стандарты на написание кода, выбор языка программирования, платформы и так далее.

+ требования к срокам разработки;

+ требования к документации;

(всё, что касается организационного процесса разработки)

Внешние требования практически не связаны с программированием.

Требования на взаимодействие описывают, с какими системами нашему программному продукту предстоит работать.

Юридические требования описывают, какие юридические нормы не должен нарушать программный продукт.

Особенность нефункциональный требований — их трудно измерить количественно.

**Количественные показатели нефункциональных требований**

1. Скорость работы — можно измерять в количестве транзакций в секунду, время реакции на действие пользователя.
2. Размер программного продукта.
3. Удобство эксплуатации — измеряется методом экспертных оценок.
4. Среднее время обучения персонала.
5. Количество разделов в помощи.
6. Надёжность — период времени между сбоями.
7. Вероятность выхода системы из строя.
8. Устойчивость к сбоям — время восстановления системы после сбоев; вероятность повреждения данных в случае сбоя; процент событий, приводящих к сбоям программного продукта.
9. Переносимость программного продукта — нельзя использовать большое количество платформо-зависимых функций.

**ПОЛЬЗОВАТЕЛЬСКИЕ ТРЕБОВАНИЯ**

Описывают как функциональные, так и нефункциональные требования к программному продукту.

**Особенности при формулировании**

— отсутствие чёткости изложения;

— смешение требований (отсутствует разделение на функциональные и нефункциональные);

— объединение требований.

**Приёмы «дрессировки» заказчика**

1. Разработать стандартную форму для указания требований, других видов требований не принимать.
2. В разработанной форме включить графу «причины» (зачем я этого хочу?).
3. Делать различия между обязательными и относительными (не рассматривать как необходимость) требованиями.
4. Ключевые понятия выделять (чтобы выделял заказчик).

**СИСТЕМНЫЕ ТРЕБОВАНИЯ**

Являются основной для проектирования программного продукта. Системные требования определяют, что система должна делать вне зависимости от реализации системы.

**Способы записи системных требований**

1. Структурированный естественный язык.
2. Языки описания программ.
3. Графические нотации (UML, например).
4. Математические конструкции.

Это и есть проектно-системные спецификации.

**РАЗРАБОТКА ТРЕБОВАНИЙ**

Это технический процесс, целью которого является создание и утверждение документа под названием «техническое задание», или спецификация требований. Этапы разработки следующие:



1. На этапе **анализа осуществимости** нужно ответить на следующие вопросы:
2. Отвечает ли будущие программный продукт общим и бизнес-целям заказчика (и разработчика)?
3. Можно ли реализовать системы, используя существующие технологии и не выходя за пределы заданной стоимости?
4. Можно ли интегрировать систему с уже используемой заказчиком?
5. Какие проблемы есть у фирмы-заказчика и как наш программный продукт будет помогать их решать?
6. Требует ли наш программный продукт новых технологий, которые ещё не использовались?
7. Нужно определиться с источниками информации о будущем программном продукте. Как правило, это отдельные категории людей, которые будут использовать программный продукт.

*По результатам этого этапа пишется отчёт о осуществимости, в котором:*

— целесообразность работы над проектом;

— могут содержаться поправки (например, в оплате).

1. **Формирование и анализ требований**.

Составляется набор требований к программному продукту с помощью общения с будущими пользователям программного продукта.

Проблемы: пользователи не всегда могут формулировать чего они хотят, могут предъявлять нереальные требования, поскольку не знают, сколько это стоит.

Мы должны учитывать разные точки зрения представителей заказчика.

Каждый из участников формирования требований высказывает свои требования (предпочтения), которые будут противоречивыми. На формировании требований сказывать влияние внутрифирменные политические факторы.

**Процесс формирования требований является циклическим и включает в себя следующие этапы:**



На этапе сбора требований может продолжаться анализ предметной области и производиться общение с возможными пользователями.

На этапе классификации требований весь неупорядоченный набор полученных требований разбивается по группам.

На этапе разрешения противоречий должны быть разрешены все противоречия.

На этапе определения приоритетов определяется важность требований (с помощью общения с заказчиком).

На этапе проверки требований проверяется полнота требований, последовательность и непротиворечивость.

Эти этапы повторяются многократно. Когда собраны все необходимые требования, составляется техническое задание.

На этапе аттестации требований производится проверка требований на полноту и непротиворечивость. Полнота требований определяется представителем разработчика совместно с представителем заказчика, учитывая, как технические факторы, так и ожидания заказчика.

Всё заканчивается созданием технического задания.

**ОПОРНЫЕ ТОЧКИ ЗРЕНИЯ**

При составлении требований к любому нетривиальному программному продукту, приходится учитывать большое количество точек зрения потенциальных пользователей продукта.

Поскольку таких точек зрения слишком много, из них выбираются «опорные» точки зрения.

**Точка зрения может интерпретироваться следующим образом:**

1. Как источника информации о системных данных.
2. Как структура представлений.
3. Как получатель системных сервисов.

На основе понятия точки зрения был создан метод VORD (Viewpoint Oriented Requirements Definition — определение требований на основе точек зрения), здесь точка зрения понимается как получатель системных сервисов.

**Этапы:**

1. Идентификация точек зрения и связанных с ними сервисов;
2. Структурирование точек зрения — строится иерархия точек зрения, общесистемные сервисы связываются с верхними уровнями иерархии и наследуются точками зрения, расположенными по иерархии ниже;
3. Документирование точек зрения;
4. Отображение точек зрения на системные объекты.

**АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ**

Это первый этап проектирования, определяется общая структура системы, определяются основные подсистемы, определяется модель управления взаимодействия подсистем.

Входные данные для архитектурного проектирования — это системные требования.

**Архитектурное проектирование включает в себя три этапа:**

1. Структурирование системы — выделяются основные системы и определяется порядок их взаимодействия;
2. Моделирование управления — разрабатывается базовая модель взаимодействия подсистем и компонентов;
3. Модульная декомпозиция — разбиение подсистем на модули.

На выходе получаем детальный проект системы.

Чёткой границы между этапами нет, работаем до тех пор, пока не будет создан проект системы.

**Подсистема от модуля отличается:**

* масштабом;
* подсистемы — это независимые части системы;
* модуль — менее независимый компонент, тесно связан с другим модулями.

Результатом архитектурного проектирования является документ, детально описывающий архитектуру системы.

**Четыре типа архитектурных моделей, разрабатываемых в ходе архитектурного проектирования:**

1. *Статическая структурная модель* — на ней представлены подсистема и компонента.
2. *Динамическая модель процессов* — на ней показывается, какие процессы протекают во время работы системы (например, диаграмма видов деятельности).
3. *Интерфейсная модель* — описывает сервисы, предоставляемые каждой подсистемой посредством общего интерфейса системы.
4. *Модели отношений* — показывает отношения между компонентами системы (передачи сообщений, данных и так далее, например — диаграмма последовательностей).

Для описания этих моделей используются языки описания архитектуры. Язык UML — унифицированный язык описания архитектуры.

Архитектурное проектирование предполагает выбор некоторой базовой структуры (архитектуры) системы.

**Этот выбор влияет на ряд характеристик будущего программного продукта:**

1. Производительность — если критическим показателем является скорость работы, то наилучшая архитектура — чем меньше связей между компонентами, тем лучше работает; архитектуру, включая как можно меньшее количество подсистем, отвечающих за операции критические для системы.
2. Защищённость — если это наиболее критический показатель, то целесообразно реализовать программный продукт в виде многоуровневой архитектуры.
3. Безопасность (защита от взлома) — если это наиболее критический показатель, то архитектура должна быть такой, чтобы операции, влияющие на безопасность, реализовывались в как можно меньшем количестве подсистем.
4. Надёжность (чтобы реже «падало») — если это наиболее критический показатель, то применяем архитектуру, включающую избыточные компоненты; то есть, чтобы моно было заменять компоненты, не останавливая работу системы.
5. Удобство сопровождения — если это наиболее критический показатель, то архитектуру системы следует проектировать на уровне мелких структурных компонентов, которые можно легко изменять. Компоненты, создающие данные, должны быть отделены от компонентов, использующие данные. Следует также избегать структуры совместного использования данных.

Эти требования противоречивы.

**1) Структурирование системы**

Предполагает представление системы в виде взаимодействующих подсистем.

**Пример архитектуры системы – модель** **репозитория**. Такая архитектура системы, в котором хранение данных реализовано централизованно.



Альтернатива — у каждой подсистемы данные хранятся отдельно.

Централизованное хранилище данных предполагает, что одна и та же модель данных (структура БД) используется для всех подсистем.

Плюсы: избегаем необходимости передавать большие объемы данных от одной подсистемы к другой.

Минус: поскольку используется одна и та же модель данных, это является неэффективно, так как одна система хочет быстро искать, другая добавлять, а третья — делать быстро какие-то запросы.

Минус: Могут возникнуть проблемы с добавлением новой подсистемы, так как её структура данных может быть не адаптируемой.

Плюс: подсистемы, которые создают данные, не должны ничего знать о том, как эти данные используются, то есть обеспечивается слабая связанность подсистем.

Плюс: все операции по администрированию данных реализуются централизовано и каждой из подсистем не нужно трогать ресурсы.

Минус: с другой стороны, это означает, что одна и та же политика обслуживания данных применяется к данным всех подсистем.

Плюс: модель репозитория прозрачна с точки зрения добавления новых подсистем. Новая подсистема просто должна быть совместима с системой по методу хранения данных.

**2) Модель «клиент-сервер»**

**Эта модель распределенной системы, которая включает в себя три основных компонента:**

— набор автономных серверов (программных компонентов), предоставляющих сервисы другим подсистем.

— набор клиентов, то есть таких компонентов, которые умеют обращаться к серверу, запрашивать сервис и сохранять его результат.

— сеть, связывающую клиентов с сервисами.

Плюс: в систему легко добавлять нового клиента и сервер

Плюс: всё хорошо с надёжностью, так как выход из строя первого сервера не влияет на другие.

Минус: если перегрузка сети и выход из строя каналов, то отдельные сервисы являются недоступными, то есть надёжности каналов связи — узкое место.

Дальнейшее развитие: это сервис-ориентированные архитектуры (SOA). Идея: помнить, что программный продукт разрабатывается для обслуживания бизнес-процессов и разрабатывать архитектуру для этого (компонентов бизнес-процессов).

**3) Модель абстрактной машины (модель многоуровневой архитектуры)**



**Жирный прямоугольник** — интерфейс. Реализуется компонент 1, пишется его интерфейс, по нему пишется компонент 2, его интерфейс и так далее, пока не будет реализован интерфейс системы.

Каждый компонент представляет собой новый уровень, он реализует свой набор сервисов, каждый компонент реализуется на основе только компонента нижнего уровня. В результате каждый компонент взаимодействует только с двумя соседями.

Модель называется «моделью абстрактной машины», так как каждый компонент реализует абстрактную машину и предоставляет свой машинный язык.

Основное преимущество модели абстрактной машины:

+ переносимость программного продукта

+ защищенность программного продукта (сбой на внешних компонентах не влияет на базовые функции)

**Минусы:**

- скорость работы, так как необходимо делегировать выполнение вплоть до самого глубоко уровня

- модель не позволяет компонентам внешних уровней напрямую обращаться к сервисам внутренних уровней.

**Модели управления.**

Под моделью управления будем понимать порядок взаимодействия подсистем и правила передачи управляющих воздействий между подсистемами и модулями.

**Можно выделить модели централизованного управления и модели управления, основанные на событиях.**

**Централизованное** **управление**.

Рассмотрим два примера.

Первый пример – **модель** **вызова-возврата**. Работает следующим образом.



Из одной процедуры вызываются другие процедуры и потом возврат к вызвавшему модулю.

Основной недостаток (он же достоинство) – только для последовательно работающей программы (последовательно запускаемых модулей). Да и параллельные процессы особо не опишешь.

Плюсы – просто отлаживать.

**Модель диспетчера.** Это альтернатива модели вызова-возврата. Позволяет организовывать параллельно работающие процессы.



Сия модель позволяет реализовывать как последовательно, так и параллельно работающие системы. Может применяться в системах реального времени, если требования к скорости реакции не очень строгие.

Контроллер или диспетчер определяет время запуска подсистем, ориентируясь на значения внутренних переменных. Так же он контролирует генерацию данных в одной подсистеме для того, чтобы передать их на обработку в другую подсистему. Пожалуй, диспетчер является единственным модулем, работа которого не прерывается.

Плюсы – универсальность.

Минусы – более сложна в отладке и создании.

**Модели управления, основанные на событиях.**

Принципиальное отличие от моделей централизованного управления заключается в том, что решения о запуске подсистем применяются не на основе внутренних состояний системы, а на основе воздействий извне. Так же порядок поступления внешних событий невозможно предугадать точно.

**Модель передачи сообщений.** Сообщение передаются всем подсистемам.



Есть набор подсистем, каждая из которых умеет реагировать на определенные сообщения. Если в системе генерируется сообщение, то оно передается обработчику сообщений и подсистеме способной его обработать.

Плюсы – в такую систему просто интегрировать дополнительные подсистемы.

Несколько подсистем могут обрабатывать одни и те же сообщения, что может привести к конфликтным ситуациям.

Систему реального времени не построишь, увы и ах.

**Системы обработки прерываний.**

При возникновении того и иного прерывания система моментально переходит к обработчику.

Недостатки: могут быть аппаратные ограничения на количество прерываний. Сложность отладки.

Штатный режим описывается моделью диспетчера, а аварийные и критические – моделью прерываний.

**Модульная декомпозиция**

**Модель потоков данных.** Предполагает, что модель представляется в виде набора модулей, на вход которых поступают определенные данные, на выходе – преобразованные данные. Так же называется моделью конвейерной обработки.

Плюсы – интуитивная понятность, простая модификация, универсальность.

Минусы – формат данных должен быть универсальным, чтобы данные могли приниматься всеми модулями.