Консультации

1. Понедельник и четверг на большом перерыве
2. Понедельник 2-3 пары и четверг 2 ПАРА на лабах

Лекция №1

Введение в технологии программирования

Цикл жизни ПО:

* Обнаружение проблемы
* Выработка требований
* Проектирование
* Кодирование и отладка
* Блочное тестирование
* Интеграция
* Тестирование системы
* Отдача заказчику
* Ввод в эксплуатацию (проверка, везде ли работает)
* Корректирующие сопровождение
* Вывод ПО из эксплуатации

Software engineering (инженерия программного продукта)

Целью технологии программирования (SE) является эффективное создание качественного программного продукта.

Под инженерией программного обеспечения будем понимать инженерную дисциплину, которая охватывает все аспекты создания программного продукта.

Программное обеспечивание – совокупность компьютерных команд сопутствующих документации и необходимых конфигурационных файлов

Программные продукты бывают разные:

* Общие (решают типичные задачи и свободно отдаются на рынок)
* Сделанные на заказ (специфические задачи от заказчика)

Технологический процесс создания программного продукта – совокупность этапов, ведущих к созданию и развитию программного обеспечивания.

Технологический процесс создания программного продукта включает этапы

* Разработка спецификации требований
* Создание программного продукта
* Аттестация (проверка на соответствие требованиям заказчика)
* Сопровождение (модернизация)

До второй половины 60х годов применялся не формальный подход к разработке формальных продуктов (Code and fix). В нем возникали проблемы со сложностью продуктов, то есть при достижении им некоторого размера, сопровождать его стало сложно.

Для управления проектом, для анализа характеристик технологического процесса создания ПО применяются различные модели этого процесса. Под моделью процесса или объекта понимается упрощенное его описание)

Идеальной модели нет, каждая из существующих моделей имеют свои плюсы и минусы. Выбор модели определяется различными факторами: требования проекта, размер команды, квалификация команды

Можно выделить следующие классы моделей технологического процесса создания ПО:

* Модель последовательности работ (совокупность этапов, зависимость между отдельными этапами, самая простая понятная модель)
* Модель потоков данных и процессов (на вход подаются данные и получаем на выходе тоже данные)
* Ролевые модели (роли участников процесса разработки и действия выполняемые участниками в этих ролях)

Классические модели

* Каскадная (этапы идут последовательно, каждый этап завершается составлением документации, каждый следующий начинается после предыдущего, проблема в том, что если мы дойдем до 3го этапа например и изменится требования то придется снова проходиться по всем этапам)
* Эволюционная (формируется начальное ТЗ, происходит разработка, сборка, аттестация, после выпускается 1 версия, отдаем заказчику, он говорит что поправить, снова разработка, сборка, аттестация и опять версия пока не достигнем финальной версии)
* Модель формальных преобразований (ТЗ описывается с помощью математических нотаций, на основе которых автоматически генерируется код программы)
* Разработка на основе раннее созданных компонентов (ускоряет разработку)

CASE технологии – инструментальные средства (программы) для сопровождения различных этапов проектирования и разработки программных продуктов.

Характеристики качественного программного продукта:

* Удобный интерфейс
* Удобство сопровождения
* Переносимость
* Надежность (работа без сбоев)
* Скорость работы
* Время восстановления после отказа

Лекция №2 Основы системотехники вычислительных систем

Системотехника рассматривает процессы создания, проектирования, сопровождения, аттестации систем как единого целого.

Система - это совокупность взаимодействующих и взаимосвязанных подсистем и компонентов, которые работают совметно для достижения общих целей.

Компоненты системы отличаются сложным взаимодействием компонентов.

Подсистема - это относительно самостоятельная часть системы, способная функционировать отдельно от других подсистем.

Модуль (компонент) - это часть системы, которая в ходе работы требует наличие других компонентов и взаимодействует с ними.

Система в целом обладает свойствами, которыми не обладал ни один её модуль по отдельности, это интеграционные свойства системы.

Безотказность системы - это показатель, зависящий от надежности отдельных компонентов и надежности связей между ними.

Удобство эксплуатации - это сложное многопараметричное свойство, характеризующее систему в целом, но отсутствующее у отдельных её компонентов.

Типы интеграционных свойств:

- функциональные - свойства, которые выполняет программный продукт или система в целом;

- не функциональные - не связаны с действиями, которыми управляет системы, а связаны с её реализацией.

Факторы, влияющие на разработку системы:

- эксплуатационный (Потребует ли внедрение ПО изменений в рабочем процессе? Если да, то придётся решать вопрос с обучением персонала);

- фактор персонала (если внедряемое ПО снизит требования к квалификации персонала или кардинально поменяет рабочий процесс или способ работы, то персонал будет препятствовать внедрению ПО);

- организационный (внедрение ПО может изменить стректуру властных полномочий в организации).

Основная проблема заключается в том, что невозможно предусмотреть все потенциальные проблемы и создать методы их устранения.

ПРОЦЕСС ПОЗДАНИЯ СИСТЕМЫ

1. В процесс разработки системы, как правило, включены различные инженерные системы.

2. При разработке системы необходимо стремиться к уменьшению повторных работ.

Этапы:

1) Распределение требований.

На этом этапе определяются и формализуются системные требования (источник - заказчик). Определяются общие функциональные требования, системные свойства (не функциональные), свойства, которые должны отсутствовать у системы. Определяются цели системы.

2) Проектирование системы.

Определение системных компонентов на основе требований.

2.1. Разбиение требований на группы.

2.2 Определение подсистем (группы требований проецируются на множества подсистем, учитывая не функциональные свойства). Меньше компонентова - лучше скорость работы. Больше компонентов - надежнее функциональность.

2.3. Определение функциональных характеристик подсистем. Формируются требования к каждой подсистеме отдельно.

2.4.Определение интерфейса подсистем.

3) Разработка подсистем.

Подсистемы, как правило, разрабатываются параллельно. Разработка возможна либо с нуля, либо на основе ранее созданноых решений. Выполнени модельного тестирования.

4) Сборка системы.

Интеграция подсистем в общую систему. Два подхода:

- методом "большого взрыва" - собираются сразу все подсистемы и после проверяется работоспособность программного продукта (быстрая сборка);

- инкрементная сборка (пошаговая) - системы интегрируются по одной и каждый раз тестируются (долго). Инкрементная сборка лучше!

5) Ввод системы в эксплуатацию.

Подэтап - инсталляция системы (установка заказчику ПП и проверяется работоспособность).

Проблемы:

- окружение, для которого разработан ПП, не совпадает с окружением, в котором его устанавливают (аппаратная и програмная несовместимость);

- пользователи могут враждебно относиться к установке Вашего ПО;

- ПО некоторое время должно работать с ПО, на котором заказчик работал ранее (если системы предполагают использование общих данных, то могут возникнуть проблемы).

Ввод в эксплуатацию предполагает ещё и обучение персонала.

6) Сопровождение.

Большое ПО характеризуется длительным сроком жизни. В течение этого срока возникает необходимость в её модификации - исправлении ошибок или изменении её функциональности.

7) Вывод из эксплуатации.

При выводе системы из эксплуатации стоит обратить внимание на сохранение данных, наработанных в ПО. Также учесть ситуацию, что компоненты удаляемого продукта могут использоваться его новой версией.

Лекция №3

Классические модели процесса создания программных продуктов

Элементарный элемент программы – лексема

Каскадная (водопадная) (Waterfall) модель состоит из нескольких последовательных этапов:

РИСУНОК

1) Определение требований

2) Проектирование системы

3) Разработка и тестирование модулей

4) Сборка и тестирование (всех модулей в целом)

5) Эксплуатация и сопровождение

Каждый следующий этап начинается только после завершения предыдущего этапа и документирования его результатов.

1. Определение требований – консультация с заказчиком, выработка требований к продукту
2. Проектирование системы – на основе ТЗ выполняется структурирование требований и разбиение на подсистемы и связи между ними с подробным описанием
3. Разработка указанных модулей и их тестирование
4. Сборка и тестирование
5. Эксплуатация и сопровождение

Основной недостаток каскадной модели - очень тяжелая реакция на изменение требований заказчика, в случае изменения требований программы требуется сначала повторить все предыдущие этапы. Все проектные решения принимаются на ранних этапах, поэтому когда в проектировании найдется ошибка будет уже поздно. Это может привести к низкому качеству итогового программного продукта. Ее целесообразно использовать, если требования изначально определены как можно более полно и формализовано

Каскадная модель исходит из мысли, что требования меняться не будут, поэтому с современными проектами она работает не очень хорошо.

Эволюционная модель – альтернатива каскадной модели и ориентирована на адаптацию меняющимся требованиям.

РИСУНОК

Этапы:

1. Эскизное описание
2. Спецификация требований
3. Разработка
4. Аттестация

Сначала мы составляем некое обобщенное описание проекта (эскизное описание), после мы переходим к следующим 3м процессам, в итоге получается начальная версия, которую отдаем заказчику для проверки. Заказчик возвращает нам какие-то уточненные требования, после чего снова повторяем те же этапы и получаем следующую версию, которую также передаем заказчику и опять же уточняет требования и так далее пока не будет создана конечная версия или пока у заказчика не закончиться бюджет.

В этой модели мы не боимся изменений требований, мы их прямо ждем. Изменение требований не создает проблем.

Аттестация – проверка на соответствие ТЗ.

Эволюционная модель предлагает два способа реализации:

1. Метод пробных разработок - изначально разрабатываем то, к чему хорошо описаны требования. В процессе разработки, за счет постоянного общения с заказчиком определяется требования остальных компонентов системы. Система эволюционирует – обрастает новыми функциями по мере их поступления от заказчика.
2. Метод прототипирования – применяется к тем частям системы, требования к которым описаны не четко, размыто, неоднозначно и тд… Он предлагает создать прототип на основе размытых требований, то есть предложить заказчику свой вариант размытых требований и дождаться уточненных требований.

На каждой итерации может применяться тот или иной подход.

Недостатки – проектирования только в начале, следовательно, проект становится менее структурирован. И постепенно проект становится почти не управляемым, тяжело добавлять новый функционал. Поэтому к большим проектам нельзя использовать его. Либо к средним, либо к малым.

Еще один недостаток, отдельные этапы часто не документируются, это затрудняет сопровождение.

Формальная разработка системы

РИСУНОК

(последовательно соединенные этапы)

Этапы:

1. Определение требований
2. Формальная спецификация
3. Формальное преобразование
4. Сборка и тестирование системы

Он основан на том, что можно описать требования к продукту описываются с помощью специальных математических нотаций (выражений). На основе этих нотаций автоматически генерируется код программы. После этого выполняется сборка и тестирование системы. Программирование без программирования. Такой подход почти не работает. Ограничение в том, что формальное описание можно составить только для очень узкого класса продуктов. Любое приложение, ориентированное на пользователя сложно формализовать.

Плюс в том, что убираются этапы программирования и тестирования модулей и программные модули точно и однозначно соответствуют техническому заданию. Достоинство также скорость разработки.

Разработка на основе раннее созданных компонентов

Этапы:

1) Спецификация требований

2) Анализ компонентов (поиск максимально подходящих компонентов)

3) Модификация требований (требования модифицируются с учетом выбранных компонентов, для того чтобы максимально использовать их возможности и функциональность). Если модификация требований невозможна, то мы возвращаемся к этапу анализа компонентов.

4) Проектирование системы

5) Разработка и сборка

6) Аттестация системы

Достоинства: скорость разработки. Повышение надежности программного продукта, т.к. используются компоненты, которые многократно протестированы в других проектах. Более эффективно используется время и опыт программистов.

Недостатки: не полное соответствие первоначальным требованиям заказчика. Невозможность переписывания компонента => возникают проблемы с сопровождением продукта.

Лекция №4 Итерационные модели разработки программного обеспечения

В нетривиальных проектах целесообразнее использовать гибридные модели. Итерационные модели заключаются в последовательном выполнении итераций.

Спецификация требований также разрабатывается итерационно в ходе разработки самого программного продукта. Окончательные требования появляются к концу разработки продукта.

Модель пошаговой разработки

Компромисс между эволюционной и каскадной моделью. Призвана взять достоинства этих двух моделей. Автор модели – Миллс.

Цель создания модели – уменьшить количество повторно выполняемых работ, а также увеличить для заказчика сроки по принятию окончательных решений по проекту. (То есть мы пытаемся решить проблемы этих моделей)

Первоначально заказчик в общих чертах определяет те сервисы, которые должна реализовывать система. Каждому сервису присваивается приоритет заказчиком (важность). Определяется кол-во шагов разработки, таким образом чтобы в результате каждого шага создавался один законченный компонент системы,который реализует определенное подмножество системных сервисов. Сервисы по шагам распределяются в зависимости от приоритета (наиболее приоритетные на первых шагах).

Схема

Определение плана требований => пошаговая детализация требований => разработка системной архитектуры => Шаг разработки (разрабатываем определенный компонент) => Шаг аттестации (проверяем созданный компонент) => Шаг сборки (интегрировали с основой) => Аттестация системы (после внедрения разработанного компонента в готовую систему тестируем всю систему) => возвращение к шагу разработки пока не закончим разработку системы

Сначала спрашиваем у заказчика какие сервисы нужны в ПО, причем выполняем детализацию требований, разрабатываем архитектуру (выполняется проектирование, определяются компоненты и связи между ними, определяем на каком шаге какой компонент будет разрабатываться), затем итерационно повторяются 4 шага. В результате тестирования заказчиком он уточняет требования.

Шаги могут быть разные по времени. (Дольше чем 4 неделе слишком длинная итерация, меньше чем неделя слишком короткая)

На каждом шаге может использоваться та или иная модель разработки, наиболее подходящая (прототипирование, эволюционная или другие)

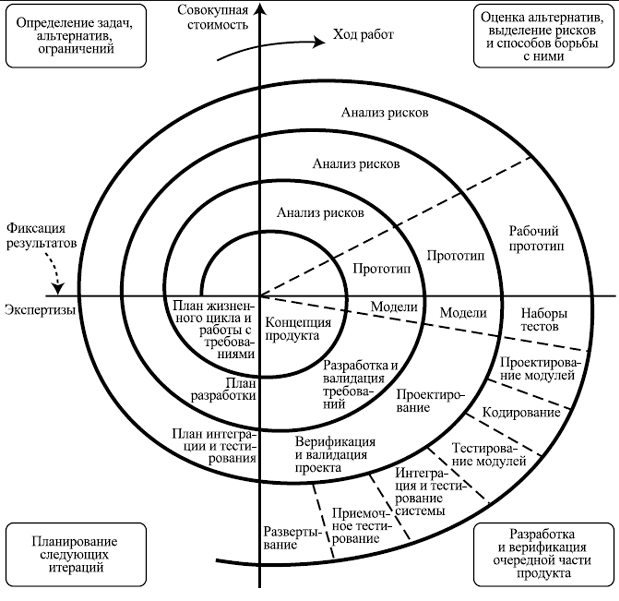
Достоинства: заказчику не нужно ждать окончания разработки чтобы понять что же мы хотим ему подсунуть, может получить представление о продукте, кроме этого, наиболее важные требования реализуются в самом начале и в случае необходимости он может уточнить требования. Также заказчик может использовать компоненты, разработанные на первых шагах использовать в качестве прототипов для проведения с ними испытаний и уточнения требований к следующим компонентам. Кроме того, поскольку система разрабатывается пошагово что снижается риск общих системных ошибок. Мы внедряем компоненты по одному, тестируем их, и после чего уточняются коррективы к последующим компонентам. Поскольку на ранних шагах реализуются наиболее важные сервисы, а все последующие компоненты интегрируются с ними то эти наиболее приоритетные сервисы тестируются многократно. Причем чем приоритетнее компонент, тем больше он тестируется => тем лучше качество его реализации.

Недостатки:

Поскольку на каждом шаге разрабатывается один компонент могут возникнуть нерешаемые проблемы с разработкой слишком сложных сервисов которые не укладываются в рамки одного компонента. Также затрудняется реализация сервисов, которые требуют взаимодействия компонентов.

Спиральная модель разработки

Схема



1 этап – Определение целей, ограничений и альтернатив (лево верх)

2 этап - Анализ альтернатив, оценка и разрешения риска (право верх)

3 этап – Разработка и тестирование (право низ)

4 этап – Планирование следующих итераций (лево низ) (а стоит ли продолжать проект и что надо сделать на след итерациях)

Принципиальным отличием данной модели от ранее рассмотренных является то что процесс разрывается не на этапы, а представляется в виде спирали, каждый виток спирали соответсвует одной итерации разработки.

На первом витке мы принимаем решение, стоит ли разрабатывать проект и тд

Каждый виток спирали включает четыре этапа:

Первый – определение целей, на этапе определяются цели текущей итерации, устанавливаются ограничения на сам продукт и процесс его разработки, уточняются планы создания отдельных компонентов и тд, здесь же определяются проектные риски (возможные проблемы при работе над проектом, факторы которые могут отрицательно сказаться на разработке, например не уложиться в сроки разработки, разработать продукт не отвечающий требованиям надежности) В зависимости от рисков определяются альтернативные пути разработки.

Второй этап каждого витка – выполняется детальный анализ и разрешение рисков (планирование мероприятий которые минимизируют последствия риска либо вообще его ликвидируют).

Третий этап – разработка и тестирование. После оценки рисков выбирается модель разработки, таким образом, спиральная модель позволяет использовать максимально подходящую методологию разработки в зависимости от рисков.

Четвертый этап – анализ выполненной работы, решение о том, нужна ли нам следующая итерация

Итерация – один виток спирали.

Спиральная модель – гибридная, на каждом витке спирали можно использовать максимально подходящую модель разработки исходя из рисков.

Недостатки – сложность, много дополнительных действий с анализом рисков и определению альтернатив. Для простых проектов она избыточна (слишком много расходов)

Спиральная и последовательная модели схожи наличием итераций.

Лекция №4 Гибкие методологии. Создание программных продуктов

История появления методологий:

1) Code and fix

2) Классические методологии (каскадная и тд…)

3) Гибкие методологии (Agile)

Классические методологии отличаются излишней бюрократизацией процесса разработки. Все управленческие решения принимаются менеджером, программист рассматривается как взаимозаменяемое программирующие устройство. (вот тебе задание, сиди работай) Программист не может принимать каких-либо решений по проекту. Слишком большой объем документации. Классические методологии предлагают неизменность самой формы организации процесса разработки. (если выбрали неправильное решение, например, о времени итераций то будем мучатся до конца)

Гибкие методологии призваны решать эти проблемы. Гибкие методологии призваны уважать программиста.

1. Проектирование и разработка

Авторами гибких методологий было предложено поскольку программирование — это сложный творческий вид деятельности, требующий квалификации разработчика, поэтому принято считать, что 90% времени при разработке уходит на проектирование. Проектом является сам программный код. Затратами времени на производство программного продукта можно пренебречь. Проектирование тут не отличается от программирования, это одно и то же.

1. Непредсказуемость требований

Требования к программному продукту будут меняться всегда. Так как требования невозможно четко сформулировать заранее, пока программного продукта нет, его невозможно оценить. Невозможно заранее оценить трудоемкость и стоимость во времени отдельных функций => заказчик не может принять решение, нужна ли ему эта функция. Требования меняются, потому что меняются цели заказчика, меняется рынок и тд.. Надо постоянно иметь актуальную информацию о требованиях заказчика.

Управление непредсказуемым процессом (процесс разработки)

Наличие актуальных сведений о требованиях заказчика. А также все нужно делать итерационно. Такой подход позволяет что-нибудь сделать, кинуть заказчику, получить новые требования и тд. Главный вопрос – длительность итераций. Норма – 4 недели. Надо подбирать длительность итераций под каждый проект.

Адаптивный заказчик

В классических методологиях отношения заказчика с разработчиком выглядит в оплате разработки продукта (покупает у нас продукт). Минус в смене требований, поэтому это не подходит (контракты с фиксированной стоимостью). Предлагается разработчику более тесно работать с заказчиком. Мы получаем заказ на набор какой-либо функциональности, делаем и получаем оплату, заказчик тестирует и добавляет требования, и мы опять же разрабатываем и получаем деньги. Заказчику не надо ждать полгода чтобы получить работающие приложение. Для разработчика хорошо тем, что не будет неожиданностей в изменении требований. Контракты с фиксированной стоимостью заменяются пошаговой оплатой.

Особенности отношения к разработчику

В классических методологиях написание программного кода рассматривается как простая задача, программисты рассматриваются как взаимозаменяемые устройства. Никакие индивидуальности разработчиков не учитываются и не используются. Все организационные решения делаются менеджерами и навязываются программисту. Такой подход содействует плохой мотивации, потенциал разработчика сдерживается. Гибкие методологии предлагают сделать все по-другому, они предлагают рассматривать программиста как высококвалифицированного профессионала с своими индивидуальными особенностями и которого можно использовать на благо проекту => программистов надо ценить и всячески не обижать. Гибкие методологии говорят, что никто лучше программистов не может программировать => принятие решений должно быть делегировано от менеджера к разработчику, сроки разработки определяют сами разработчики, а также выбор технологии для разработки компонентов также принимает разработчик, но на него ложится большая ответственность. Тут больше мотивация к работе.

Роль бизнес консультанта

Гибкие методологии говорят, что надо постоянно взаимодействовать с заказчиком. В экстремальных случаях может быть включен в команду эксперт по предметной области, его задача оперативно отвечать на вопросы по предметной области. (например, если делаем ПО для строителей, то нужен строитель, который будет отвечать на вопросы)

Адаптация адаптивного процесса

Выше были рассмотрены способы адаптации к меняющимся требованиям заказчика. Гибкие методологии говорят, что адаптироваться может сама форма способа организации процесса разработки.

Адаптация процесса разработки основана на его регулярном анализе и пересмотре. В конце каждой итерации проводится небольшое совещание, на котором команда отвечает на следующие вопросы:

1. Что вы сделали хорошо? (в ходе предыдущей итерации)
2. Чему команда научилась в ходе этой итерации?
3. Что можно сделать еще лучше?
4. Какие проблемы возникли?

Определяя ответы можно легко выяснить что было плохо и последующие итерации начинать, скорректировав процесс разработки.

Лекция №5 Примеры гибких методологий

Экстремальное программирование – методология, ориентированная на быструю разработку маленькой команды. Предлагает использование 12ти принципов которые объединяются в 4 группы. Проще напишешь – понятнее будет читать.

12 приемов экстремального программирования заключаются в 4х группах

1. Короткий цикл обратной связи – программисты должны получать быструю обратную связь с заказчиком. Начинается все с планирования разработки (проектирования). Игра в планирование. Заказчик составляет задачи и дает им важность (приоритеты), а программист определяет стоимость реализации задач. Программист оценивает трудоемкость реализации работы, а не менеджер. Игра в планирование проводится в начале каждой итерации. Разработка через тестирование (Test Driven Development) – идея в том что сначала пишем тесты, а потом продукт. На начальном этапе основываясь на требованиях разрабатываются модульные тесты (программа которая будет тестировать разрабатываемые программы). Затем мы программируем, пишем те модули, для которых уже написали тест. Как только тест выполнился это значит что хватит программировать. Плюсы: экономим время и получаем четкие критерии оценки. Минусы: предполагает высокую квалификацию программиста (чтобы написать тесты), так как торопимся, можем получить не крутой вариант решения.

2)Заказчик на рабочей площадке – предполагает взять специалиста по предметной области на работу. Таким образом обеспечивается оптимальный способ получения ответов на предметную область.

1. Непрерывный, а не пакетный процесс разработки:

1)Непрерывная интеграция – каждый вновь созданный компонент интегрируется в продукт

2)Частый выпуск версий – заказчик постоянно получает очередную (протестированную и работоспособную) версию продукта

3)Рефакторинг – постоянная переработка программного кода – изменение внутренней структуры программного кода, не изменяя его внешнее поведение (работу) Рефакторинг нужно проводить всегда

1. Понимание проекта всеми участниками – дополнительная мотивация всем разработчикам, ибо все знают к чему идем. Все начинается с проектирования

(главное -простота проектирования и разработки – писать надо просто) (Надо использовать простые конструкции, не те, которые сокращают строки кода в одну, главное, чтобы другой разработчик мог разобраться) Метафора системы – перед тем как разрабатывать нужно представить, что это будет (самый лучший способ сравнить с каким-то похожим проектом) Следующий прием – применение стандартов написания кода. Следующий прием – парное программирование – два разработчика сидят за одним ноутом, клавой, мышкой, это хорошо для обучения. Следующий прием – коллективное владение кодом – предполагает, что за каждый участок кода отвечает, как минимум 2 разработчика. Тем самым обеспечивается помехозащищенное программирование.

1. Социальная защита программистов – включает всего 1 прием. Здесь предполагается 40-часовая неделя. 5 рабочих дней по 8 часов каждая. Основная цель – обеспечить единый режим работы для всех программистов.

Лекций №6 SCRUM

Этапы развития SCRUM:

1. Хиротака Такецти и Икудэиро Нонака предложили подход регби для разработки ПО в 1986
2. Подход регби упоминается у Дегрейса Шгаля в 1991г. «Злые проблемы, справедливые решения» и там он уже назван SCRUM
3. Кен Швабер в начале 90-х начал использовать SCRUM в своей компании
4. Швабер, Сазерленд сформулировали принципы SCRUM (что такое SCRUM и как с ним работать). Они основные идеологи SCRUM.
5. Швабер, Бидл написали книгу «Гибкая разработка программного обеспечения с помощью SCRUM» в 2001 году

SCRUM – итерационная методология разработки. Итерация в скраме называется спринт и длится от 1 до 4 неделей. В итоге каждого спринта получается инкремент функциональности продукта, и этот продукт передается заказчику на тестирование

Product Backlog – список требуемых функций с приоритетом, затем из него выбираются наиболее приоритетные требования, которые команда обязуется реализовать в ближайший спринт

Product Backlog

|  |
| --- |
| 1 задача (самая важная) |
| 2 задача (менее вважная) |
| 3я |
| … |

Из Product Backlog делается Sprint Backlog в который идут самые приоритетные задачи

Sprint Backlog

|  |
| --- |
|  |
|  |
|  |

Каждые 24 часа (каждый день) мы собираемся и обсуждаем что получилось после получения временной версии продукта которая отдается заказчику

Роли участника SCRUM процесса

Все участники делятся на две группы, на тех кто полностью в проекте (свиньи), и те кто заинтересованы в конечном результате, но не полностью в нем участвуют (куры).

К первой группе относится владелец продукта (Product Owner), он отвечает за представление интересов заказчика, он не член нашей команды, это представитель заказчика, он должен разбираться в предметной области и разбираться как делаются продукты (типо менеджера, разбирается в предметной области, но не шарит как кодить). Его задача поддерживать в актуальном состоянии Product Backlog и адекватно оценивать приоритеты. Также к свиньям относится Scrum master – он нужен для организации митингов (обсуждений), следит за соблюдением правил и принципов методологий, он не программирует, его задача наладить коммуникацию между разработчиками. Также к свиньям относится функциональная команда (Scrum Team), в которую входит не только разработчики, но и тестировщики и остальные. Команды в скраме обычно маленькие, до 7 чловек. Никто кроме команды в течении спринта не вмешивается в процесс разработки.

Мы получили требования и в течении спринта реализуем их и не отвлекаемся на другие требования.

Куры: пользователи, клиенты, консультанты – те, кто для разработчиков является источником мнений для заказчика. Они непосредственно не занимаются разработкой.

Артефакты SCRUM (документация)

1. Product Backlog – документ, содержащий список требований отсортированные по приоритетам (степени важности). Каждый элемент Product Backlog называется User story. Product Backlog открыт для всех участников. Каждый элемент включает в себя основные и дополнительные элементы: основные - идентификатор и название и приоритет, предварительная оценка трудоемкости выполнения (какой объем работ нужно выполнить чтобы реализовать эти требования), способ демонстрации (каким образом это требование будет проверить по окончанию спринта, или заказчик или автоматизированная проверка). Дополнительные : категория (когда требования надо разбивать на некоторые подгруппы), компоненты (базаданных, классы или чет такое, какие компоненты потребуется модифицировать при реализации этого требования), инициатор запроса (кто захотел такой сервис), идентификатор деффекта
2. Sprint Backlog – подмножества элементов Product backlog с наибольшими приоритетами. В одном спринте задач на 160 часов.
3. Burmdown Chart (трудоемкость – сколько человекачасов еще надо)



Типы совещаний

* Планирование спринта – выполняется до начала каждой итерации (выбирается объем работ, который команда реализуется реализовать в течении спринта, затем осуждается каким образом будет реализован этот объем работ, каждый элемент из тех задания, выбранный для текущего спринта и разбивается на подзадачи и оценивается в человека часах). Такое совещание длится от 4 до 8 часов. Это организация рабочего времени.
* Ежедневное совещание – совещание, проводится каждый день в течении спринта в одно и то же время в одном и том же месте. Оно нужно чтобы вся команда держалась в тонусе. В собрании могут высказываться только свиньи, остальные не могут ничего добавлять. Длительность этого совещания не более 15 минут. Каждый участник совещания отвечает на 3 вопроса: что сделано с момента прошлого совещания, что будет сделано до следующего совещания, какие проблемы мешают достижению целей спринта, если проблема есть, то они решаются на отдельном совещании которое организует скрам мастер со всеми разработчиками, которые разрабатывают часть, в которой произошла проблема
* Демонстрация – демо митинг, собирается совещание в конце спринта, команда демонстрирует полученный элемент функциональности заинтересованным лицам. В этом собрании участвует максимальное кол-во слушателей. В демонстрации участвует каждый участник команды, демонстрируя что лично он сделал за прошедший спринт. Длительность – 4 часа.
* Ретроспектива – участники команды высказывают свое отношение к ходу прошлой итерации (спринта). Происходит после демонстрации. Отвечается на два вопроса: что было сделано хорошо, что надо улучить и что не допускать в будущем. По результатам этого совещания модифицируется процесс разработки. Решаются вопросы, как улучшить процесс разработки. Длительность – от 1 до 3 часов.

Когда использовать SCRUM:

* Когда тех задание, либо не понятно, либо задания постоянно модифицируются, либо когда надо быстро запустить проект с минимальной функциональностью.

Когда не надо использовать SCRUM:

* В которых ТЗ совершенно технически не может меняться (гоз заказы)
* Если квалификация команды низкая (неадекв. просчет времени на реализацию)
* Когда некомпетентный менеджер проекта
* Занижены сроки или бюджет разработки

Скрам предпологает гибкий бюджет проекта и адаптивного заказчика.

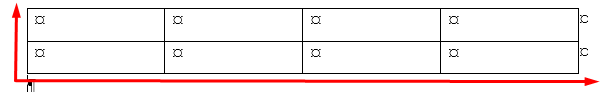
Лекция №7 Семейство методологий Crystals

Автор методологии – Алистер Колберн.

«Выживание в объектно-ориентированном проекте» - одна из лучших работ по итерационной разработке.

Для выбора методолгии надо провести анализ:

1. По Y кол-во участников проекта, по Y критичность



Эти методологии направлены на поиск разумного компромисса между низким и высоким уровнем дисциплины.

По окончанию каждой итерации пересматривается организация процесса разработки.

**Open Source –** исходный код проектов открыт для анализа, либо тестирования всеми желающими.

Для проектов open source используется свой подход в разработке.

Главной особенностью является такого рода проектов то, что в таких проектах участвует огромное кол-во физически-удаленных разработчиков.

Работа организуется следующим образом:

1. У каждого проекта есть координатор (ответственное лицо, которое имеет право вносить изменения в итоговый репозиторий)
2. Прочие разработчики предлагают изменения
3. Такая форма работы требует особой организации изменений в проекте
4. Роль координатора может быть реализована по разному, либо это один координатор на весь проект, либо это отдельный координатор на отдельные модули, либо роль координатора может быть переходящей (от одного разработчика к другому)
5. Большинство разработчиков работают над такими проектами бесплатно, в свободное время, неполный рабочий день
6. Отладка продукта может проводиться параллельно большим кол-вом участников

**3. ASD –** адаптивная разработка

Автор – Джим Хайсмит

Обдумывание, сотрудничество, обучение

Особое внимание выделяется активному сотрудничеству между разработчиками.

Обучение – анализ выполненной работы и вывод о том, как сделать процесс разработки лучше

**4. FDD – feature driven development – разработка**

**Лука Ж.Д., Piter Coad**

Короткие итерации – две недели

Этапы:

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

Каждый из процессов разбивается на задачи

Все участники проекта делятся на две группы:

1. Владельцы классов (Class owners)
2. Старшие программисты (Chief Programmers) – не реализуют свойства программы, они определяют какие классы будут затронуты при реализации функции и распределяют работу между владельцами классов

Книга «UML in Color” Питера Коада

1. **DSDM (Dynamic Systems Develop Method)**

Придумана методология в 94 году.

Основные принципы: Частый выпуск версий, активное взаимодействие с пользователем. Высокий уровень самостоятельности разработчиков при принятии решений. Тестирование в течении всего цикла разработки.

Итерации от 2 до 6 недель

Особое внимание идет на качество кода и адаптацию к меняющимся требованиям заказчика.

Особенности практического использования гибких методологий

* Требования меняются со временем, либо изначально не сформулированы
* Основная ответственность переходит к разработчикам
* Команда состоит из высококвалифицированных и опытных разработчиков
* Готовность заказчика плотно участвовать в разработке (часто взаимодействовать с командой и отвечать на вопросы)

Основное ограничение гибких методологий – объем проекта. (в крайнем случае это методологии Crystal, в них могут работать до 50 чел). Когда стоимость и труд фиксированны.

Лекция №10 Требования к программному обеспечению

Требования бывают:

* Противоречивые
* Нереализуемые
* Размытые
* Четкие

Практически невозможно создать полную спецификацию требований. Еще одна проблема в том, что требования меняются и заказчик сам не понимает, какие требования он бы хотел применить к программному продукту, который ему нужен.

Для упрощения процедуры составления требований используют разные модели

Группы требований:

* Пользовательские
* Системные
* Проектно-системная спецификация

К нам приходит заказчик и говорит какую программу хочет (хорошая, не дорогая, чтобы быстро работала…) такие требования плохо сформулированы, не формализованные требования. Мы берем требования и пытаемся заказчику на технический язык перевести ему что он пытался сказать. Эти формализованные требования – системные требования. Дальше надо выделить детальные требования каждому модулю и тд.

Пользовательские требования – неформализованное описание функциональных и не функциональных требований к продукту. Они касаются исключительно внешнего поведения системы и не касаются внутренней структуры программы. Пользовательские требования обычно описываются на естественном языке, возможно с диаграммами, таблицами.

Пользовательские требования – описание пользователем на своем языке требований к продукту.

Недостатки:

* Отсутствие четкости в изложении требований
* Смешение требований (пользователь понятия не имеет что требования бывают функциональные и не функциональные, он хочет, чтобы программа работала быстро и была красивой, такое требование не функциональное. Смешиваются функциональные и не функциональные требования.)
* Объединение требований (в одном требовании пользователь описывает сразу несколько функций)

Нам с этими требованиями работать, поэтому надо привести их в порядок.

Во-первых, заказчик, например, приносит 3 листа и говорит, что вот что он хочет, вы можете не понять че там, 1й способ — это использовать стандартные формы для описания пользовательских требований (дизайн документ, в нем все то что нам надо, заказчик заполнит ответы, на конкретные вопросы, которые нас интересуют). 2й способ – различать обязательные и описательные требования. (если пользователь говорит, что хочет такую-то функцию – то это обязательное требование, требование к удобству интерфейса, последовательности действий, работы программы не обязательны) Обязательные требования – то что принципиально для приложения. Не обязательные – для удобства, которыми можно пренебречь. 3й способ – не нужно заставлять пользователя использовать какие-то термины, пусть просто опишет требования на русском.

Допустим мы получили требования, и они норм сформулированы. Переходим к системным требованиям. Они представляют собой формализованное и детализированное представление пользовательских требований.

Системные требования – основание для заключения контракта на разработку. Поэтому они должны содержать максимально полную спецификацию программной системы. Они являются стартовой точкой для проектирования программного продукта.

Системные требования не должны описывать архитектуру системы. Мы описываем какие функции нужны.

Как пишутся системные требования: они могут описываться на естественном языке, но желательно быть приведено к стандартным пунктам и шаблоном для описания функциональности (входные данные, выходные, алгоритмы и тд…) Также системные требования могут быть описаны с помощью языков описания программ – строго структурированные языки. Системные требования также могут описываться с помощью графических нотаций (например, диаграмма прецедентов, или структурная схема алгоритмов). Также можно определить требования с помощью математических нотаций (обозначений).

Проектная системная спецификация – дальнейшая детализация системных требований (она может уже содержать инфу о структуре, архитектуре программного продукта). Именно такая спецификация является входными данными для детального проектирования и последующей разработки ПП (программного продукта). Она никаким образом не касается заказчика, это внутренний документ для команды разработчиков.

Требования формулируются постепенно, начиная с нечетких размытых требований пользовательских, затем стараемся их формолизовать функционально, затем системные требования, затем договор и дальнейшая детализация, затем проектная сист специф.

2я классификация требований – разбиение требований на функциональные и не функциональные и требования предметной области

Функциональные – список сервисов, которые должна предоставлять система. При описании функциональных требований для каждого сервиса описываются входные/выходные данные и тд.. Иногда указывается чего система не должна делать. Функциональные требования могут быть описаны на нескольких уровнях (пользовательские и системные). Основная проблема – нечеткость или размытость требований.

Спецификация функциональных требований должна обладать такими свойствами: быть комплексной (описывать все сервисы ПП) и быть не противоречивой (не предъявлять взаимно исключающие требования к функциям программы).

Следующий вид требований – нефункциональные требования – не касаются функций системы и описывают характеристики системы.



Организационные требования описывают политику и организоционные процедуры заказчика и разработчика. Требования к языку программирования, методологии, срокам разработки, к документации и тд..

Внешние требования описывают факторы внешние по отношению к продукту так и к процессу его разработки. Здесь рассматриваются вопросы по взаимодействию нашего продукта с другими.

Основная проблема нефункциональных требований – сложность в проверке.

Количественные показатели не функциональных требований:

Размер ПП

Удобство эксплуатации (кол-во разделов в справочной системе)

Надежность системы

Устойчивость к сбоям (время восстановления после сбоя, вероятность повреждения данных в результате сбоя, процесс событий преводящих к сбою системы

Переносимость характеризуется кол-вом платформенно-зависимых операторов и модулей. Чем их больше, тем хуже переносимость ПП.

Важнее нефункциональные требования (ибо пользователю должно быть удобно)

Нефункциональные требования являются более критичными для ПП, поскольку невыполнение одного из нефункциональных требований могут сделать программу не пригодной к использованию.

Требования к предметной области – описывают условия, в которых будет работать программный продукт.

Требования бывают пользовательские, системные, проектно-ситемные спецификации.

Лекция №

Лекция №12 Архитектурное проектирование программных систем

После разработки ТЗ необходимо переходить к проектированию.

1 этап проектирования – архитектурное проектирование – представляет собой выбор обобщенная архитектура системы, выбирается модель управления системой и затем подсистемы разбиваются на модули (модульная декомпозиция).

Входными данными для 1го этапа является спецификация требований разработанная ранее (она не должна содержать никаких требований к архитектуре ПП)

В результате архитектурного проектирования мы получим детальную проектную системную спецификацию, которая может быть использована уже как задание для программистов.

Основные этапы архитектурного проектирования

1. Структурирование системы – на этом этапе будущая программная система структурируется и представляется в виде совокупности относительно независимых подсистем. Определяется как будет происходить взаимодействие между системами (порядок взаимодействия) и их интерфейсы. В итоге мы видим какие подсистемы у нас есть и как они между собой связаны.
2. Моделирование управления – разрабатывается базовая модель управления взаимодействием между подсистемами. (Определяем, как мы будем управлять этими подсистемами)
3. Модульная декомпозиция – после определения структуры системы выполняем разбиение подсистем на модули, определяем типы модулей и порядок их взаимодействия. (просто выполняем проектирование классов)

Эти этапы выполняются не последовательно, а накладываются друг на друга. Они также итерационно повторяются пока не будет создана архитектура отвечающая всем системным требованиям.

Результатом архитектурного проектирования является документ содержащий подробное исчерпывающее описание архитектуры системы. (набор диаграмм с подробным описанием)

Как правило разрабатываются следующие 4 типа архитектурной модели:

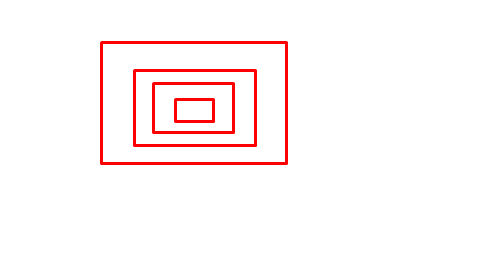
1. Статическая структурная модель – показаны подсистемы и компоненты, которые в дальнейшем будут независимо разрабатываться.
2. Динамическая модель процессов – иллюстрирует организацию процессов во время работы системы
3. Интерфейсная модель – представляются те сервисы (функции), которые предоставляются каждой из подсистем через общий интерфейс.
4. Модели отношений – они описывают взаимоотношение между частями системы (например, потоки данных между подсистемами)

Примером описания архитектуры является набор UML диаграмм.

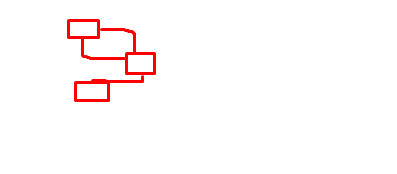
При архитектурном проектировании больших систем как правило недостаточно одной модели, поэтому используется несколько различных архитектурных моделей.

Архитектура системы влияет на нефункциональные свойства системы:

1. Производительность (скорость работы) – если производительность системы является ключевой характеристикой системы, то необходимо архитектуру спроектировать таким образом, чтобы за критические функции системы отвечало, как можно меньше подсистем с минимальными взаимодействиями между ними. Чтобы уменьшить взаимодействие между модулями надо использовать небольшое кол-во крупномасштабных модулей. (Все должно работать по возможности в одном модуле) Такой подход ускоряет систему, но снижает удобство по сопровождению системы, а также снижается надежность т.к. выход из строя модуля ведет к краху всей системы.
2. Защищенность системы – от всяких проникновений извне. Для этого используется многоуровневая архитектура.



А не



Критические компоненты реализованы на внутренних уровнях.

Проверка безопасности выполняется на верхних уровнях.

1. Безопасность – в данном случае, если это ключевая характеристика, то необходимо чтобы за безопасность отвечало как можно меньше подсистем.
2. Надежность – если это ключевое требование к системе, то необходимо чтобы архитектура включала в себя избыточные компоненты, которые дублируют функциональность друг друга. В этом случае выход из строя одного из компонентов не будет сильно влиять на работоспособность системы и повлияет отрицательно на скорость.
3. Удобство сопровождения – в этом случае, ПП должен состоять из небольших компонентов (структурных элементов), которые легко модифицировать. Компоненты создающие данные должны быть отделены от компонентов использующие эти данные, также желательно избегать совместного использования данных.

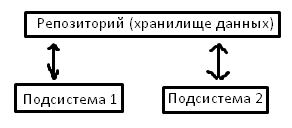
Как видно из рассмотренных выше особенность, разные нефункциональные характеристики программы предъявляют противоречивые требования (взаимоисключающие) на архитектуру программного продукта.

Чтобы это избежать придется искать компромисс между отдельными требованиями. Один из способов – использование разных архитектур для разных подсистем проектируемого ПП.

Лекция№ 13 Структурирование системы

Структурирование – 1 этап арх проектирования – определяется из каких подсистем будет состоять продукт и как они будут связаны между собой. Это все изображается в виде схем, блок-схем с потоками данных между ними.

Модель репозитория – есть некоторое центральное хранилище данных и есть несколько подсистем которые с этими данными работают.

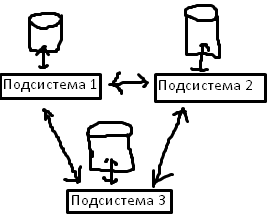


Система состоит из набора подсистем.

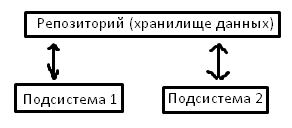
Для каждой из подсистем можно определить входные, промежуточные и выходные данные. Выходные данные из одной подсистемы (результат выполнения) могут являться входными для следующей.

Чтобы организовать хранение и передачи данных есть неск способов:

1. Локальные хранилища данных в каждых подсистемах - Каждая подсистема имеет свое локальное хранилище данных, если данные одной подсистемы нужны другой, то между ними надо организовать передачу данных.



1. Удаленное хранилище данных (репозиторий) – есть некоторая центральная база данных, реализованная специальным способом чтобы удовлетворять все подсистемы. Все операции по администрированию данных выполняются централизованно. Применяется одна и та же политика безопасности для данных всех подсистем. Управление данными выполняется централизовано => не надо тратить ресурсы каждой из подсистем. (Все данные хранятся в центральном хранилище)



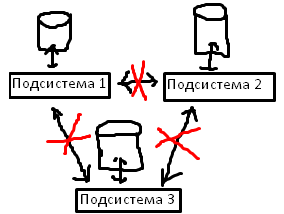
Большой плюс системы репозитория в том, что данные хранятся централизовано => не надо передавать большой объем данных между локальными подсистемами. Повышается надежность системы в целом, так как каждая из подсистем не завязана на остальные. Повышается скорость работы (не надо передавать большие объемы данных между лок хранилищами).

Все операции по администрированию данных выполняются централизовано, не требуя вмешательство подсистем.

Новые подсистемы можно легко добавить в наш ПП, главное чтобы он был согласован с моделью данных на репозитории.

Минусы системы репозитория в том, что для хранения данных используется единая модель данных, которая не может учитывать требования всех подсистем. Основная проблема – снижение скорости работы (каждая подсистеме нужна определенная модель чтобы работать быстрее, но у нас одна универсальная модель). Приходится идти на компромисс между требованиями которые предъявляют подсистемы.

Ослабляется связность подсистем, подсистемы производящие данные не должны знать, как они используются другими подсистемами



Следующий недостаток – поскольку администрирование данных выполняется централизованно => к данным разных подсистем приходится применять одну и ту же политику безопасности, и одни и те же требования надежности хранения данных. В то время как разные подсистемы могут работать с разными данными. (Требования по безоп: к данным которые не критичны придется проводить лишние операции, например, по авторизации).

Проблема этой модели в том, что слишком огромные нагрузки на репозиторий (централизованный сервер) => к нему предъявляются большие требования к надежности. Чтобы избежать этой проблемы существуют распределенные системы.

Данный репозиторий – база данных с необходимыми модулями для решения задач администрирования. ЭТО НЕ GIT и MERCURIAL.

Следующий вариант архитектуры: клиент-сервер



Это распределенная модель, которая предполагает наличие компонентов 3х основных типов:

1. Набор автономных серверов (программная подсистема которая предоставляет сервисы по запросу других компонентов)
2. Набор клиентов – подсистема, которая умеет запрашивать сервис у сервера и получать и интерпретировать результат.
3. Сеть передачи данных (Инфраструктура для передачи данных) – через нее подсистемы передают запрос и входные данные для сервера. (В общем случае клиенты и серверы могут работать на 1м усройстве)

Репозиторий пассивный – не выполняет никаких действий. Только возвращает данные, а подсистемы их обрабатывают.

Сервер активный – выполняет много действий. Вычисления производятся на сервере.

Для запроса сервиса клиент должен знать протокол от сервера (адрес сервера, запрос который ему нужно направить, входные аргументы которые необходимо направить, какой результат ожидается в ответ)

Основное достоинство клиент-серверной архитектуры это ее распределенность. На основе этой архитектуры можно эффективно распределить функциональность системы между большим кол-вом устройств. Эта архитектура позволяет легко внедрять новых клиентов и не требует перестраивать исходную архитектуру. Новые серверы также легко добавляются в систему, для их добавления лишь нужна публикация их протокола. Здесь обеспечивается минимальная связность подсистемы (все компоненты функционируют максимально независимо)

Недостатки – снижение скорости работы т.к. необходимо передавать запросы между физически удаленным клиентом и сервером. Также критичный критерий – надежность передачи данных по сети.

Следующая модель – модель абстрактной машины

Она предлагает реализовать программную систему в виде набора вложенных подсистем, каждый уровень описывает некоторую абстрактную машину реализующую набор команд на основе которых разработан компонент следующего уровня. (например перегон .NET приложения в MSIL код и компилятор который перегоняет этот псевдокод в норм код под систему)

Каждая подсистема реализует некоторый набор команд (АПИ, набор функций) которые реализуют следующие уровни, которые тоже предоставляют функции и т.д.

Плюсы: такая модель обеспечивает хорошую переносимость программных продуктов т.к. за взаимодействие с платформой отвечает только один самый внутренний компонент. Каждый компонент взаимодействует только с двумя соседними уровнями, что упрощает модификацию.

Минусы: скорость работы (снижение скорости работы, т.к. надо перенаправлять запросы по всей иерархии уровней). Усложнение архитектуры. Компоненты верхнего уровня не могут получить доступ к сервисам внутренних уровней. Каждый уровень может взаимодействовать только с двумя соседними

Лекция №

Цель этапа – выбрать модель управления взаимодействием подсистем. Определиться, каким способом будет организована работа всей системы как единого целого. В результате будет простоена динамическая модель управления системы. По этой модели можно проследить развитие потока управления системы и подсистемы.

Модели управления можно разбить на 2 типа:

1. Модели централизованного управления

Есть некоторая центральная подсистема которая координирует работу остальных подсистем.

1. Управление основанное на событиях

Центральной подсистемы нет и все поведение управляется внешними событиями

Централизованное управление

Первая модель, которая относится к этому типу, это модель вызова-возврата. Эта модель используется в классических языках (Pascal, C)

Основное ограничение это что такая модель может применяться только к последовательному выполнению. Сложность обработки исключительной ситуации

Следующая модель, относящаяся к централизованной – модель диспетчера.

В этой модели есть некий центральный модуль, которая полностью контролирует работу остальных модулей (запускает их на выполение, приостанавливает выполнение и тд.)

В момент времени когда контроллер запустит очередной модуль определяется только внутренним состоянием системы, никакие внешние взаимодействия не смогут на это воздействовать.

Контроллер работает постоянно. Такой подход позволяет разрабатывать модели из паралельно работающих модулей

Эта модель более универсальна.

Для систем реального времени модель диспетчера не подходит. Может использоваться там, где скорость реакции не критична.

**Управление основанное на событиях**

Управление производится внешними воздействиями, которые невозможно планировать

Событие рассматривается как сообщение, которое передается одной подсистемой и может быть обработано одной из других подсистем

При добавлении подсистемы в общую систему в обработчики сообщений регистрируются те сообщения, которые может принимать и обрабатывать добавляемая подсистема

При необходимости запросить какие то действия, подсистема отправляет сообщение которое принимается обработчиком сообщений

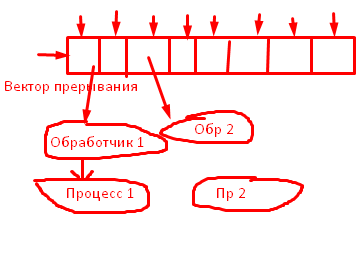
Обработчик Передает той подсистеме, которая может обработать это сообщение, при необходимости он организует двухточечное взаимодействие между подсистемой отправившей сообщение и подсистемой которая его принимает и обрабатывает

Подсистема, отправляющая сообщение не знает какой подсистемой оно будет обработано.=> отсуствует жесткая связь между отправителем и обработчиком сообщения.

С точки зрения надежности идет нагрузка на обработчик.

Основной недостаток – никак не гарантируется время обработки сообщения (подсистема передающая сообщения не знает когда будет ответ.

Управление на основе прерываний



Используется в системах реального времени, где скорость реакции – основное требование

Основное примущество: скорость реакции на события.

Недостатки – сложность отладки подобных систем

Можно предложить гибрид централизованной модели и прерывания. Обычный ход программы управляется диспетчером, а особо важные воздействия обрабатываются как прерывания

На этапе модульной декомпозиции решается вопрос разбиения подсистем на модули.

Объектно-ориентированная декомпозиция – определяем из каких объектов будет состоять подсистема.

Модель потоков данных – подсистему можно разбить на отдельные модули пользуясь моделью потоков данных. Подсистема представляется в виде набора фильтров (преобразователей), через которые проходят данные, эти фильтры выполняют некоторое функциональное преобразование данных. Эти фильтры могут повторяться (повторно использоваться) Могут применяться поэлементно, а могут пакетно. Такую модель также называют конвеерной

Фильтры могут быть соединены последовательно, паралельно.

Модификация – добавление нового фильтра

Лекция № Основы проектирования интерфейсов пользователей

Все изучается с изучения и анализа действий пользователя

После чего создаем некий прототип интерфейса. Он передается пользователю, если ему не нравится мы снова выполняем проектирование, после устраивания мы переходим к динамическому прототипу ( с кнопками) также пользователь оценивает и мы переходим к реализации 

Первый принцип – учет знаний пользователем. В интерфейсе должны быть термины и понятия, взятые из предыдущего опыта пользователей вашего программного продукта.

Второй принцип – принцип согласованности – однотипные операции должны выполняться одним и тем же способом

Следующий принцип – минимум неожидонастей – поведение программного продукта должно быть предсказуемл

Следующий- способность к восстановлению – должны быть предусмотрены способы восстановить данные после ошибочных действий ( Ctrl + Z).

Следующие – должно быть составлено руководство пользователя

Следующее – необходимо учитывать разнородность пользователей (пользователи разного уровня квалификации)

Способы взаимодействия с пользователем

Основное достоинство подхода – интуитивно, простота изучения

Недостаток – сложная реализация

Следующий способ – выбор из меню - + в том что пользователю сложно что-то сломать так как возможности пользователя ограничены. Недостаток в том что для опытных пользователей это слишком медленный способ, гораздо быстрее будет через например сочетание клавиш

Антипатерны – примеры плохой практики проектирования и разработки

Первый антипаттерн – программирование копипастом – нам лень писать что-то заного, мы копируем подходящий блок кода и копируем его в несколько мест => из-за этого усложняется модификация т.к. надо везде во всех блоках модифицировать код, снижается качество и поддержка кода.

Чтобы с этим бороться надо дублирующийся код выносить в метод

Следующий антипаттерн – спагети код – слабоструктурированная и плохо спроектированная система. Причина возникновения – лень, и редкий review кода, также недостаток опыта в ООП. Чтобы бороться с этим надо выполнять рефакторинг.

Следующий антипатерн – золотой молоток – изучили мы один паттерн и решили что можно его применять везде. Причина в недостаточной компетенции. Чтобы с этим бороться можно для каждой задачи предлагать несколько вариантов решений

Следующий антипаттерн – magic numbers. Причина – спешка при разработке.

Чтобы бороться – код cod review

Следующий антипаттерн – хардкодинг – жесткое кодирование (внесение данных об окружении программы в реализацию) (Например нужно читать из файлы, мы задаем имя файла в глобальном ппути указывая всю иерархию) Чтобы бороться – code review

Следующий антипаттерн – мягкое кодирование (софт кодинг) – как хардкодинг но наоборот, боимся переборщить потому абсолютно все обеспечиваем без хардкода. Бороться через код review

Следующий антипаттерн – ненужная сложность (любая избыточность, лишние проверки)

Следующий антипаттерн – лодочный якорь – есть проект, мы его оптимизировали ,есть классы, которые жалко стирать и мы их не стираем, а они не нужны. Их надо удалить

Следующий анитипаттерн – изобретение велосипедов. Мы изобретаем велосипед потому что считаем себя самым умным.

Следующий антипаттерн – одноколесный велосипед, либо велосипед с квадратными колесами – есть чужое хорошее решение, а мы изообретаем свое плохое

След – поток лавы - например комментарии «не знаю как это работает но менять не надо» (блок кода который хз как работает) Может настать время когда надо будет добавть функционал в этот блок кода, а мы не сможем. Решение –рефакторинг (code view)

След – программирование перебором – есть теория, есть эксперимент, когда есть алгоритм, который работает неправильно и мы пытаемся его подогдать под правильное решение

Антипаттерн – безумное комментирование – сопровождать комментарии которые устаревают – лишняя трата времени, лучше писать код, который понятен без комментариев

След – Слепая вера – нодостаточная проверка входных данных – все данные поступающие извне могут быть не верными, поэтому надо их проверять.

След – Универсальный объект – писать алгоритмическую программу на языке ооп (все действия программы в одном классе)