1. Определение рефакторинга и его цели.

* Определение рефакторинга как существительного:
  + ***Рефакторинг*** *(Refactoring) (сущ.): изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.*
* Определение рефакторинга как глагольной формы:
  + *Производить рефакторинг (Refactor) (глаг.): изменять структуру программного обеспечения, применяя ряд рефакторингов, не затрагивая его поведения.*
* Рефакторинг представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны.

1. Техника безопасности.

Безопасным является такой рефакторинг, который не нарушает работу программы.

Есть несколько вариантов, чтобы обеспечить безопасность рефакторинга:

* Надеяться на свое умение писать код.
* Надеяться на свой компилятор в поиске не замеченных вами ошибок.
* Надеяться на свой комплект тестов в поиске ошибок, не замеченных вами и вашим компилятором.
* Надеяться на разбор кода, который выявит ошибки, не замеченные вами, вашим компилятором и вашим набором тестов.
* Другой подход заключается в определении и создании прототипа инструмента проведения рефакторинга для проверки возможности безопасного рефакторинга программы и в случае положительного ответа его проведения. В результате удается избежать многих ошибок, появляющихся в результате действия человеческого фактора.

1. Понятие технического долга.

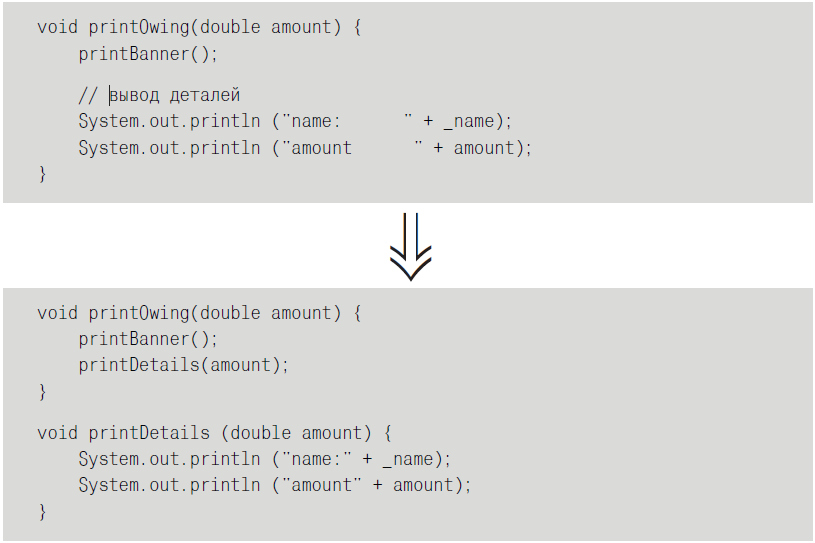
* Технический долг — это накопленный багаж ошибочных архитектурных решений, неэффективных реализаций, мертвых технологий и просто программных ошибок.
* Тот самый момент, когда "проще все выкинуть и написать с нуля" начинают всерьез рассматривать как вариант.
* Технический долг — это когда есть только один гуру, который знает "как это работает".
* Технический долг — это когда при реализации одной малозначительной фичи приходится полность перелопачивать ядро системы, ломая имеющуюся функциональность.
* Под техническим долгом понимается дополнительная стоимость обслуживания и расширения, обусловленная чрезмерной сложностью кода.
* Приближение срока окончания работ – единственный случай, когда можно отложить рефакторинг, ссылаясь на недостаток времени. Однако проведение рефакторинга приводит к росту производительности труда. Нехватка времени обычно сигнализирует о необходимости рефакторинга.
* В классическом понимании, т.е. в том виде, в котором эта метафора была описана Вардом Каннингемом, под техническим долгом понимается осознанное компромиссное решение, когда заказчик и ключевые разработчики четко понимают все преимущества от быстрого, пусть и не идеального технического решения, за которое придется расплатиться позднее. И хотя с точки зрения многих разработчиков ситуация, когда плохое решение может быть хорошим, может показаться безумной, на самом деле, это вполне возможно: если краткосрочное решение позволит компании получить видимые преимущества, выпустить продукт раньше конкурентов, удовлетворить ключевого заказчика или каким-то другим образом получить преимущества перед конкурентами, тогда такое решение совершенно оправданно. Иногда это может быть единственным способом, чтобы долгосрочная перспектива вообще существовала.

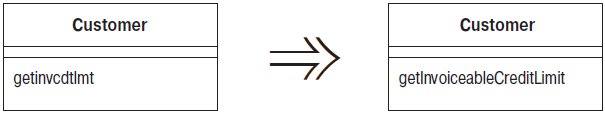
1. Примеры запахов (code smell) и способы их устранения (3-5 примеров).

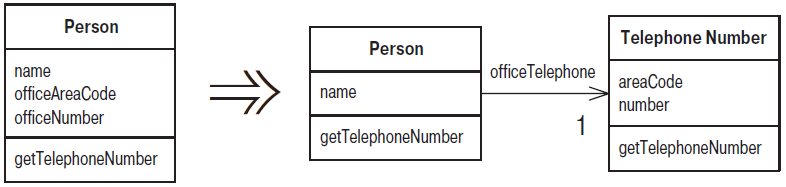
Методы борьбы с:

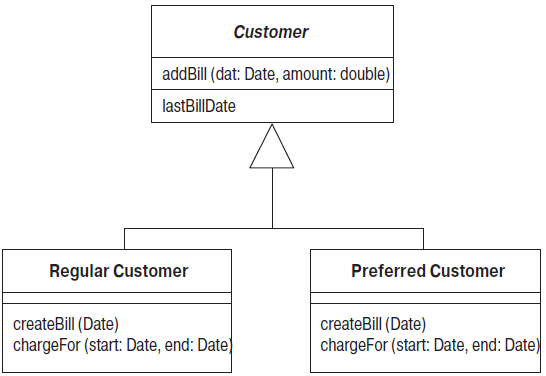
* **Комментариями.**
  + Выделение метода (*Extract Method*).
  + Переименованием метода (*Rename Method).*
  + Введение утверждения (*Introduce Assertion).*
* **Дублированием кода.**
  + Выделение метода (*Extract Method*).
  + Выделение класса (*Extract Class*).
  + Подъем поля (*Pull Up Field).*
  + Формирование шаблона метода (*Form Template Method*).
  + Замещение алгоритма (*Substitute Algorithm*).
* **Длинным методом.**
  + Выделение метода (*Extract Method*).
  + Замены временной переменной вызовом метода (*Replace Temp with Query*).
  + Введение граничного объекта (*Introduce Parametr Object).*
  + Сохранение всего объекта (*Preserve Whole Object).*
  + Замена метода объектом метода (*Replace Method with Method Object*).
  + Декомпозиция условных операторов (*Decompose Conditional*).
* **Большим классом.**
  + Выделение класса (*Extract Class*).
  + Выделение подкласса (*Extract Subclass*).
  + Выделение интерфейса (*Extract Interface*).
  + Дублирование видимых данных (*Duplicate Observed Data*).
* Длинным списком параметров.
  + Замена параметра вызовом метода (*Replace Parameter with Method).*
  + Сохранение всего объекта (*Preserve Whole Object).*
  + Введение граничного объекта (*Introduce Parameter Object).*
* Типами встроенного в имя.
  + Переименованием метода (*Rename Method).*
* Ничего не говорящими именами.
  + Переименованием метода (*Rename Method).*
* Противоречивыми именами.
  + Переименованием метода (*Rename Method).*
* Мертвым кодом.
  + Удаление.

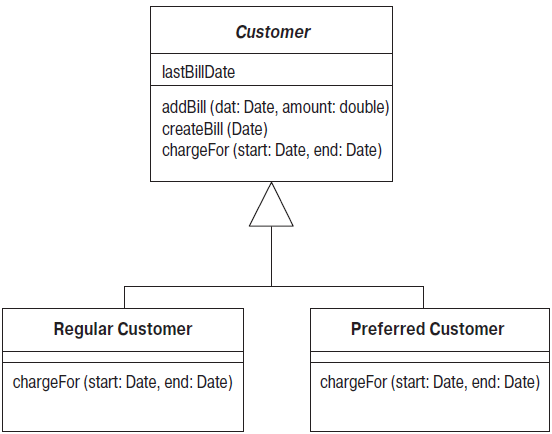
1. Примеры рефакторингов (конкретных приемов), не менее 6 штук.

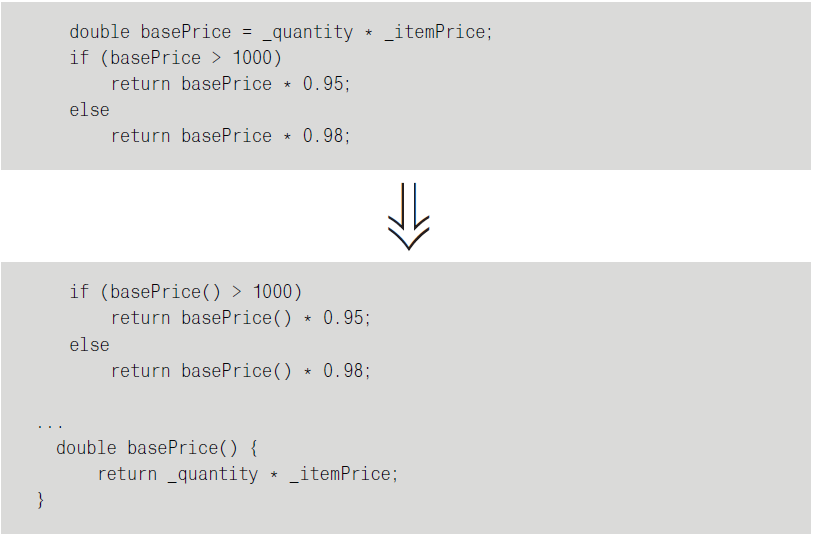
Выделение метода.

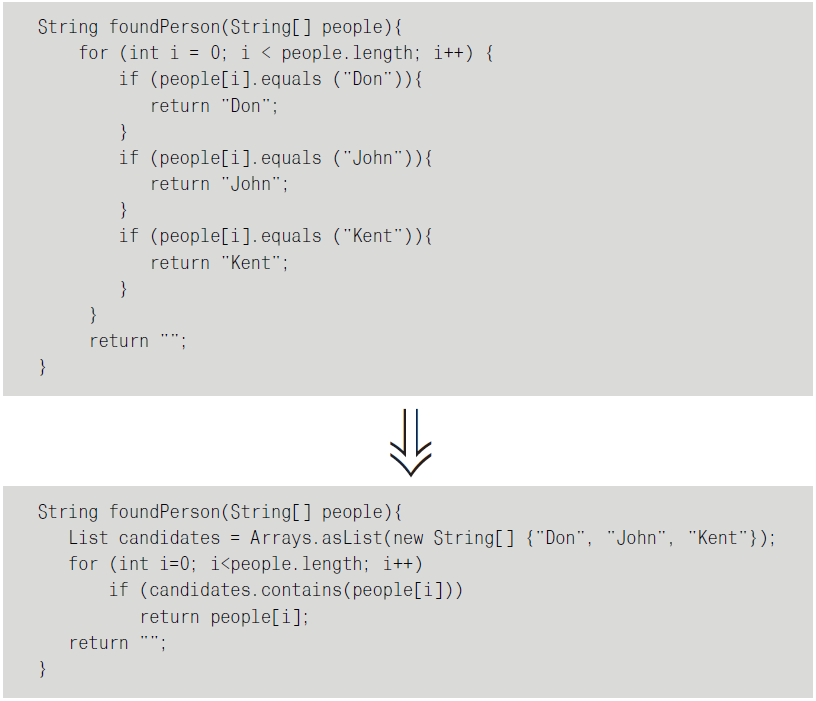
Переименование метода.  


Выделение класса.  


Подъем метода.  




Замена временной переменной вызовом метода.  


Замещение алгоритма.

1. Что такое TDD, его назначение

* Определение:
  + TDD – техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.
* Назначение:
  + Тесты защищают от ошибок. Поэтому время, затрачиваемое на отладку, снижается многократно. Большое количество тестов помогает уменьшить количество ошибок в коде. Устранение дефектов на более раннем этапе разработки, препятствует появлению хронических и дорогостоящих ошибок, приводящих к длительной и утомительной отладке в дальнейшем.
  + Тесты позволяют производить рефакторинг кода без риска его испортить.
  + Разработка через тестирование способствует более модульному, гибкому и расширяемому коду.
  + Тесты могут использоваться в качестве документации.

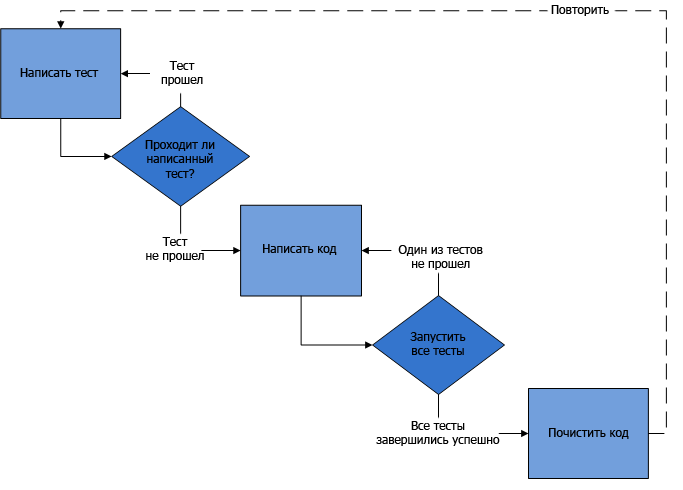
1. Мифы о TDD

* TDD = Unit Testing
* TDD = 100% test coverage (не нужно тестировать: get/set, GUI, Private members)
* TDD = Объем работ \* 2 (уменьшается время на поиск и исправление ошибок, что уменьшает объём работы)
* Unit tests – это все, что нужно

1. Цикл разработки по TDD (в виде рисунка и с максимумом деталей)

С презентации:  


С википедии:



1. Атрибуты хорошего unit-теста (включая AAA)

* AAA
  + Arrange – определить предусловия (инициализация тестовых данных,   
    предварительные установки).
  + Act – действие (собственно то, что тестируется).
  + Assert – постусловия (что должно быть в результате выполнения действия).

Хороший тест:

* Один Assert
* Быстрый (чтобы не отвлекаться – не делать перекур)
* Независимый
  + Друг от друга
  + Окружения
  + Порядка запуска
* Высокое качество = Production Code
* Автоматический

1. Типичные проблемы с GUI (ответственности, тестирование, переносимость).

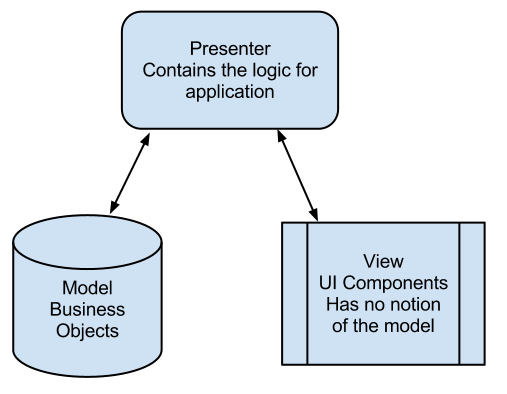
Нарушение SRP – у View 3+ ответственности:

* Ввод данных
* Представление данных
* Получение данных
* События
* Поведение

Из-за того, что логика не отделена от представления следует:

* Сложность тестирования логики приложения.
* Трудность переиспользования кода (логики).

1. Диаграмма Passive View. Ответственности слоев.



Ответственности слоёв:

* Модель (<http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA>Model) представляет собой интерфейс, определяющий данные для отображения или участвующие в пользовательском интерфейсе иным образом.
* Представление (<http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA>View) - это интерфейс, который отображает данные (Модель) и маршрутизирует пользовательские команды (или события) Presenter-у, чтобы тот действовал над этими данными.
* Presenter взаимодействует с Моделью и Представлением. Он извлекает данные из хранилища (Модели) и форматирует их для отображения в Представлении.

1. Назначение и преимущества применения Passive View.

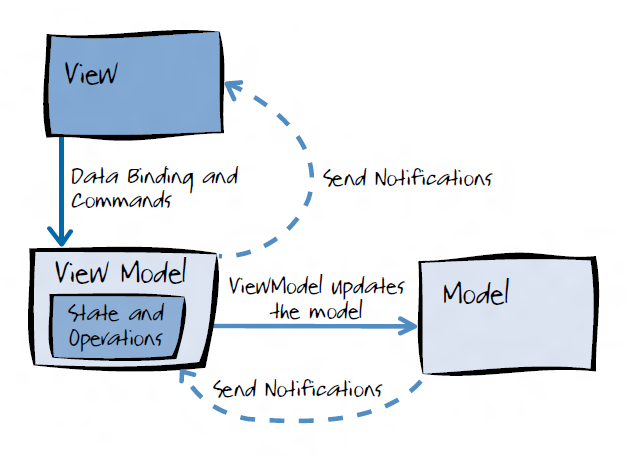
Назначение:

* MVP (Model-View-Presenter) — шаблон проектирования пользовательского интерфейса, который был разработан для облегчения автоматического модульного тестирования и улучшения [разделения ответственности](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE%D1%82%D0%B2%D0%B5%D1%82%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8) в презентационной логике (отделения логики от отображения).

Преимущества:

* Данный подход позволяет создавать приложения с использованием TTD.
* В отличает от MVC, в MVP проще тестировать логику, так как она частично перенесена из View в Presenter.

1. Диаграмма MVVM. Ответственности слоев.



Ответственности слоёв:

* Модель (<http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA>Model), так же, как в классической MVC, Модель представляет собой фундаментальные данные, необходимые для работы приложения.
* Представление (<http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA>View) — это графический интерфейс, то есть окно, кнопки и.т.п. Представление является подписчиком на событие изменения значений свойств или команд, предоставляемых Моделью представления. В случае, если в Модели представления изменилось какое-либо свойство, то она оповещает всех подписчиков об этом, и Представление, в свою очередь, запрашивает обновленное значение свойства из Модели представления. В случае, если пользователь воздействует на какой-либо элемент интерфейса, Представление вызывает соответствующую команду, предоставленную Моделью представления.
* Модель представления (<http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA>View Model) является, с одной стороны, абстракцией Представления, а с другой, предоставляет обёртку данных из Модели, которые подлежат связыванию. То есть, она содержит Модель, которая преобразована к Представлению, а также содержит в себе команды, которыми может пользоваться Представление, чтобы влиять на Модель.

1. Назначение и преимущества применения MVVM.

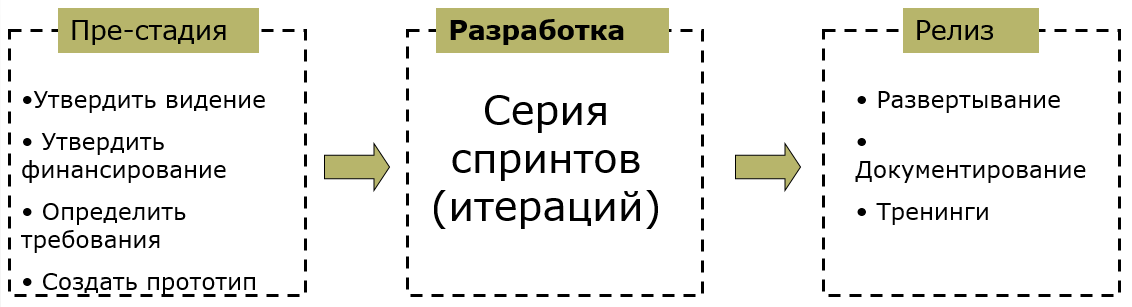
Назначение:

* Шаблон Model-View-View-Model (MVVM) — применяется при проектировании архитектуры приложения в котором присутствует двухсторонний биндинг (синхронизация) элементов управления на модель. MVVM используется для разделения модели и её представления, что необходимо для изменения их отдельно друг от друга. Например, разработчик задает логику работы с данными, а дизайнер соответственно работает с пользовательским интерфейсом.

Преимущества:

* Не нарушается SRP (Принцип единственной ответственности).
* Данный подход позволяет создавать приложения с использованием TTD.
* Гораздо проще писать unit тесты – вся логика из View перенесена во ViewModel.
* Полезное разделение ответственностей – дизайнер может спокойно работать с view без возможности каким-либо образом изменить или повредить логику.
* Переиспользование кода – Логика из ViewModel может быть использована для нескольких View.

1. Жизненный цикл Scrum-проекта.



1. Жизненный цикл спринта.

Спринт[http://ru.wikipedia.org/wiki/Scrum - cite\_note-10](http://ru.wikipedia.org/wiki/Scrum#cite_note-10) — итерация в скраме, в ходе которой создаётся функциональный рост программного обеспечения. Жёстко фиксирован по времени. Длительность одного спринта от 2 до 4 недель.

* Из резерва проекта выбираются задачи, обязательства по выполнению которых за спринт принимает на себя команда;
* На основе выбранных задач создается резерв спринта. Каждая задача оценивается в идеальных человеко-часах;
* Решение задачи не должно занимать более 12 часов или одного дня. При необходимости задача разбивается на подзадачи;
* Каждый день начинается с Daily Scrum.
* По завершению спринта команда предоставляет потенциальное улучшение продукта.

1. Роли в Scrum, их основные ответственности.

* Product Owner – представляет интересы конечных пользователей и других заинтересованных в продукте сторон.
  + Концепция, Vision
  + Backlog
  + ROI
  + Управление ожиданиями всех заинтересованных лиц
  + Взаимодействие с командой
  + Приемка кода
* Scrum Master – проводит совещания (Scrum meetings) следит за соблюдением всех принципов скрам, разрешает противоречия и защищает команду от отвлекающих факторов. Данная роль не предполагает ничего иного, кроме корректного ведения скрам-процесса.
  + Создает атмосферу доверия
  + Модерирует совещания
  + Устраняет препятствия
  + Отвечает за процесс
* Команда
  + 5-9 человек
  + Кросс-функциональные специалисты: (программисты, тестеры, юзабилисты и пр.)
  + Самоорганизующаяся команда
  + Состав не меняется во время спринта

1. Артефакты в Scrum, правила работы с ними (Product и Sprint backlog, Burndown chart).

* **Product backlog** (Резерв проекта) — это список требований к функциональности, упорядоченный по их степени важности, подлежащих реализации. Элементы этого списка называются «пожеланиями пользователя» (*user story*) или элементами резерва (*backlog items*). Резерв проекта открыт для редактирования для всех участников скрам процесса.
  + Техническое задание
  + Список нужных работ по проекту
  + Объясняет, какой в этом смысл ☺
  + Приоритизирован product owner’ом (в каждом спринте)
* **Sprint backlog** (Резерв спринта) — содержит функциональность, выбранную владельцем проекта из резерва проекта. Все функции разбиты по задачам, каждая из которых оценивается скрам-командой. Каждый день команда оценивает объем работы, который нужно проделать для завершения спринта
* **Burndown chart** (Диаграмма сгорания задач) – диаграмма, показывающая количество сделанной и оставшейся работы. Обновляется ежедневно с тем, чтобы в простой форме показать подвижки в работе над спринтом. График должен быть общедоступен.

1. Основные практики Scrum.

* Планирование (на разных уровнях)
* Длина спринта не более 30 дней
* Самоорганизующаяся команда
* Scrum-совещания
* «Не добавляйте в спринт»
* Демонстрации продукта
* Решение в течение часа
* Преграды уходят в течение дня
* Команда ~7 человек
* Общая комната (предпочтительно)
* Ежедневная интеграция (как минимум)
* Ретроспектива спринта

1. Daily Scrum / Stand-up meeting: назначение, правила проведения.

* Начинать в одно и то же время
* Длительность не более 15 минут
* Проводить каждый день
* Задавать каждому по кругу три вопроса

1. Ретроспектива: назначение, правила проведения.

Проводится после завершения спринта.

* Члены команды высказывают своё мнение о прошедшем спринте.
* Отвечают на два основных вопроса:
  + Что было сделано хорошо в прошедшем спринте?
  + Что надо улучшить в следующем?
* Выполняют улучшение процесса разработки (решают вопросы и фиксируют удачные решения).
* Ограничена одним—тремя часами.

1. Почему важно поддерживать код в чистоте.

Потому что в таком коде легче разобраться и его легче поддерживать, сохраняя производительность труда на одном уровне, а не утопая в разборе плохого кода, дополняя его ещё более ужасным при добавлении функционала. Время потраченное на поддержание код в чистоте на много меньше, чем время разбирательства в плохом коде для его исправления или дополнения.

1. Ключевые понятия при разговоре о качестве кода, их использование.

* Дублирование (копии кусков кода).
* Шум (переменные с большим префиксом).
* Единообразие (единый стиль).
* Принцип единой ответственности (Выполнение методами и объектами одной операции).
* Ортогональность.
* Выразительность (содержательность имён).
* Уровни абстракции.

1. Основные правила именования.
2. Хорошие имена – самодокументированный код.
3. Плохие имена – вы не понимаете предметную область вашего приложения.

* Имена должны передавать намерения программиста.
* Избегайте дезинформации (часто встречающихся аббревиатур, малозаметных различий в именах).
* Используйте осмысленные различия (если имена различаются, то они должны означать разные понятия).
* Используйте удобопроизносимые имена.
* Выбирайте имена удобные для поиска.
* Имена переменных – верблюжья запись с маленькой буквы.
* Имена классов – верблюжья запись состоящая из комбинаций существительных.
* Имена методов – глаголы или глагольные сочетания.

1. Основные правила оформления функций. Способы решения каждой из проблем.

* Функция должна делать только одну операцию. Делать её хорошо. И ничего другого делать не должна.
* Нормальная длина функции – длина одного экрана.
* Чем меньше входных параметров у функции, тем лучше. При росте количества параметров функции возрастает сложность её тестирования.
* Возвращать параметр нужно через «return», избегая возвращения через параметры.
* Нужно избегать кодов ошибок. Следует использовать конструкцию «try, catch», по возможности, вынося её в отдельную функцию.
* Избегать операторов ветвления (if, else, switch) заменяя их, по возможности, полиморфизмом.
* Аргументы-флаги – сразу указывают на ветвление и сообщают, что функция выполняет несколько операций.

1. Примеры плохих комментариев, почему они считаются code smell.

* Не комментируйте плохой код — перепишите его (комментарий не исправляет плохого кода).
* Неактуальная информация (большой header).
  + /\*---------------------------------------------------------------
  + -----------------------
  + Created by: NANDA
  + Created Date: 01-AUG-2009
  + Modified by:
  + Procedure Description: Fetches menu items based on the given
  + user permission
  + Input Parameters: LoginEntry user
  + Output Parameters: none
  + ----------------------------------------------------------------
  + --------------------\*/
  + public BEMenuList FetchMenuItems(LoginEntity user) {
  + ...
  + }
* Устаревший комментарий (когда комментарий не соответствует коду).
  + ...
  + // Gets the login user id
  + // Gets the CRM details
  + FetchCrmDetails();
  + ...
* Избыточный комментарий (когда из кода ясно, что он делает).
  + // If the server variable is empty , throw the error
  + message
  + if (loginUserId == null)
  + {
  + throw new Exception(“No User Id”);
  + }
* Плохой комментарий (описывание действия вместо его совершения).
  + public void LoadProperties() {
  + try
  + {
  + var propertiesPath = propertiesLocation +
  + "/" + PROPERTIES\_FILE;
  + var propertiesStream = File.Open(propertiesPath);
  + loadedProperties.Load(propertiesStream);
  + }
  + catch (IOException ex) {
  + //If file with properties doesn’t exist,
  + //default settings are loaded
  + }
  + }
* Закомментированный код (мертвый код следует удалять).
* Дезинформация (когда комментарий не соответствует коду).
  + /\*Auxiliary method: returns control if this.closed is true.
  + Throws exception when timeout is reached\*/
  + public void WaitForClose(long timeoutMillis) {
  + if (!closed) {
  + Wait(timeoutMillis);
  + if (!closed) {
  + throw new Exception("ResponseSender could not be closed");
  + }
  + }
  + }
* Пояснение в нетривиальных ситуациях (когда пояснения будут полезны).
  + // format matched kk:mm:ss EEE, MMM dd, yyyy
  + Pattern timeMatcher = Pattern.Compile("\\d\*:\\d\*:\\d\* \\w\*, \\w\* \\d\*, \\d\*");
* Предупреждения о последствиях (предупреждение других программистов о нежелательных последствиях от каких-либо действий).
* Заметки разработчика (желательно избегать).
  + //TODO…
* Шум (комментарии утверждающие очевидное).

План ответа:

* Формулировка, раскрыть все слова названия.
* Как достигается.
* Негативные последствия нарушения.

1. SRP (Single responsibility principle - Принцип единственной обязанности) – На каждый объект должна быть возложена одна единственная обязанность. Другими словами, класс или модуль должны иметь одну и только одну причину измениться. В противном случае при изменении одной функции класса, другие его функции могут прийти в негодность.
2. OCP (Open/closed principle – Принцип открытости/закрытости) – Программные сущности должны быть открыты для расширения, но закрыты для изменения. То есть, код может быть переиспользован через наследование, и если вы появляются новые функции, то для их реализации должен быть создан класс наследник. В противном случае, изменения в исходном коде потребуют проведение пересмотра кода, его повторного тестирования, чтобы получить право на использования его в программном продукте, что ведёт к финансовым и временным затратам. Код, подчиняющийся данному принципу, не изменяется при расширении и поэтому не требует таких трудозатрат.
3. LSP (Liskov substitution principle – Принцип подстановки Барбары Лисков) – Объекты в программе могут быть заменены их наследниками без изменения свойств программы. Другими словами, для всего лишь повторного использования кода из базы, наследование применять нельзя. При нарушении данного принципа сильно сужается простор для манёвра архитектора в дальнейшем поддержании и рефакторинге наследуемого класса.
4. ISP (Interface-segregation principle – Принцип разделения интерфейса) – Много специализированных интерфейсов лучше, чем один универсальный. Другими словами, клиенты не должны зависеть от интерфейсов, которые они не используют. При нарушении этого принципа система теряет гибкость при внесении изменений в логику работы и становится мало пригодной для рефакторинга.
5. DIP (Dependency inversion principle – Принцип инверсии зависимостей) – Зависимости внутри системы строятся на основе абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. В противном случае, в программе будет большая связанность, что приведёт к большим затратам сил и времени при каком-либо её изменении.
6. Закон Деметера (что можно, чего нельзя)

Основной идеей является то, что объект должен иметь как можно меньше представления о структуре и свойствах чего угодно (включая собственные подкомпоненты).

Общее описание правила: *Объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B и у объекта B есть доступ к объекту C*.

Более формально, Закон Деметры для функций требует, что метод М объекта О должен вызывать методы только следующих типов объектов:

* собственно самого О
* параметров М
* других объектов, созданных в рамках М
* прямых компонентных объектов О
* глобальных переменных, доступных О, в пределах М

Другими словами: «Используйте только одну точку».

Преимущества:

* Код разработанный с соблюдением данного закона делает написание тестов более простым, а разработанное программное обеспечение менее сложно при поддержке и имеет большие возможности повторного использования кода. Так как объекты являются менее зависимыми от внутренней структуры других объектов, контейнеры объектов могут быть изменены без модификации вызывающих объектов (клиентов).

Недостатки:

* Недостатком Закона Деметры является то, что иногда требуется создание большого количества малых методов-адаптеров (делегатов), для передачи вызовов метода к внутренним компонентам.