*NOTE FROM SERGIUS: дополнительные сводки полезной информации я ограничил и закрасил серым цветом. Они не относятся к данному вопросу, но поясняют фразы в нем. Для лучшего понимания и усвоения. P.S: Ванос сосет бибу*

**18) Характеристики тестов F.I.R.S.T.**

Чистые тесты должны обладать еще пятью характеристиками, названия которых образуют приведенное сокращение.

***Чистые тесты*** *Какими отличительными признаками характеризуется чистый тест? Удобочитаемостью (ясность, простота и выразительность.).*

**Быстрота (Fast).** Тесты должны выполняться быстро. Если тесты выполняются медленно, вам не захочется часто запускать их. Без частого запуска тестов проблемы не будут выявляться на достаточно ранней стадии, когда они особенно легко исправляются. В итоге вы уже не так спокойно относитесь к чистке своего кода, и со временем код начинает загнивать.

**Независимость (Independent).** Тесты не должны зависеть друг от друга. Один тест не должен создавать условия для выполнения следующего теста. Все тесты должны выполняться независимо и в любом порядке на ваше усмотрение. Если тесты зависят друг от друга, то при первом отказе возникает целый каскад сбоев, который усложняет диагностику и скрывает дефекты в зависимых тестах.

**Повторяемость (Repeatable).** Тесты должны давать повторяемые результаты в любой среде. Вы должны иметь возможность выполнить тесты в среде реальной эксплуатации, в среде тестирования или на вашем ноутбуке во время возвращения домой с работы. Если ваши тесты не будут давать однозначных результатов в любых условиях, вы всегда сможете найти отговорку для объяснения неудач. Также вы лишитесь возможности проводить тестирование, если нужная среда недоступна.

**Очевидность (Self-Validating).** Результатом выполнения теста должен быть логический признак. Тест либо прошел, либо не прошел. Чтобы узнать результат, пользователь не должен читать журнальный файл. Не заставляйте его вручную сравнивать два разных текстовых файла. Если результат теста не очевиден, то отказы приобретают субъективный характер, а выполнение тестов может потребовать долгой ручной обработки данных.

**Своевременность (Timely).** Тесты должны создаваться своевременно. Модульные тесты пишутся непосредственно перед кодом продукта, обеспечивающим их прохождение. Если вы пишете тесты после кода продукта, вы можете решить, что тестирование кода продукта создает слишком много трудностей, а все из-за того, что удобство тестирования не учитывалось при проектировании кода продукта.

**Заключение:** Для «здоровья» проекта тесты не менее важны, чем код продукта. А может быть, они еще важнее, потому что тесты сохраняют и улучшают гибкость, удобство сопровождения и возможности повторного использования кода продукта. Постоянно следите за чистотой своих тестов. Постарайтесь сделать их выразительными и лаконичными. Изобретайте тестовые API, которым отводится роль предметно-ориентированного языка тестирования, упрощающего написание тестов. Если вы будете пренебрежительно относиться к тестам, то и ваш код начнет загнивать. Поддерживайте чистоту в своих тестах.

**19)** **Строение класса**

Каждое определение класса начинается с ключевого слова class, затем следует имя класса, и далее пара фигурных скобок, которые заключают в себе определение свойств и методов этого класса.

Именем класса может быть любое слово, при условии, что оно не входит в список зарезервированных слов PHP, начинается с буквы или символа подчеркивания и за которым следует любое количество букв, цифр или символов подчеркивания.

<?php  
class SimpleClass  
{  
    // объявление свойства  
    public $var = 'значение по умолчанию';  
  
    // объявление метода  
    public function displayVar() {  
        echo $this->var;  
    }  
}  
?>

Мы предпочитаем размещать приватные вспомогательные функции, вызываемые открытыми функциями, непосредственно за самой открытой функцией. Такое размещение соответствует правилу понижения, в результате чего программа читается как газетная статья.

Псевдопеременная $this доступна в том случае, если метод был вызван в контексте объекта. $this является ссылкой на вызываемый объект. Обычно это тот объект, которому принадлежит вызванный метод, но может быть и другой объект, если метод был вызван [статически](https://www.php.net/manual/ru/language.oop5.static.php) из контекста другого объекта.

**Инкапсуляция**

Мы предпочитаем объявлять переменные и вспомогательные функции приватными, но относимся к ним без фанатизма. Иногда переменную или вспомогательную функцию приходится объявлять защищенной, чтобы иметь возможность обратиться к ней из теста. С нашей точки зрения тесты исключительно важны. Если тест из того же пакета должен вызвать функцию или обратиться к переменной, мы используем защищенный или пакетный уровень доступа. Тем не менее начинать следует с поиска способа, сохраняющего приватность. Ослабление инкапсуляции всегда должно быть последней мерой.

**Классы должны быть компактными!**

Первое правило: классы должны быть компактными. Второе правило: классы должны быть еще компактнее. Но как и в случае с функциями, компактность должна стать основным правилом проектирования классов. И для классов начинать следует с вопроса: «А насколько компактными?» Размер функций определяется количеством физических строк. В классах используется другая метрика; мы подсчитываем ответственности.

Имя класса должно описывать его ответственности. В сущности, имя должно стать первым фактором, способствующим определению размера класса. Если для класса не удается подобрать четкое, короткое имя, вероятно, он слишком велик. Чем туманнее имя класса, тем больше вероятность, что он имеет слишком много ответственностей.

**20) Принцип единой ответственности (SRP)**

**Принцип единой ответственности (SRP)** утверждает, что класс или модуль должен иметь одну — и **только одну — причину для изменения.** Этот принцип дает нам как определение ответственности, так и критерий для оценки размера класса. Классы должны иметь одну ответственность, то есть одну причину для изменений. Если класс имеет несколько назначений, его нужно разделить на отдельные классы.

**Принцип единой ответственности** — одна из самых важных концепций в объектно-ориентированном проектировании. Кроме того, его относительно несложно понять и соблюдать. Но как ни странно, принцип единой ответственности часто оказывается самым нарушаемым принципом проектирования классов. Мы постоянно встречаем классы, которые делают слишком много всего. Почему?

Заставить программу работать и написать чистый код — совершенно разные вещи. Обычно мы думаем прежде всего о том, чтобы наш код заработал, а не о его структуре и чистоте. И это абсолютно законно. Разделение ответственности в работе программиста играет не менее важную роль, чем в наших программах.

К сожалению, слишком многие из нас полагают, что после того, как программа заработает, их работа закончена. Мы не переключаемся на усовершенствование ее структуры и чистоты. Мы переходим к следующей задаче вместо того, чтобы сделать шаг назад и разделить разбухшие классы на отдельные блоки с единой ответственностью.

В то же время многие разработчики опасаются, что множество небольших узкоспециализированных классов затруднит понимание общей картины. Их беспокоит то, что им придется переходить от класса к классу, чтобы разобраться в том, как решается более крупная задача.

Однако система с множеством малых классов имеет не больше «подвижных частей», чем система с несколькими большими классами. В последней тоже придется разбираться, и это будет ничуть не проще. Так что вопрос заключается в **следующем:** *хотите ли вы, чтобы ваши инструменты были разложены по ящикам с множеством небольших отделений, содержащих четко определенные и подписанные компоненты? Или вы предпочитаете несколько больших ящиков, в которые можно сваливать все подряд?*

Каждая крупная система содержит большой объем рабочей логики и обладает высокой сложностью. Первоочередной целью управления этой сложностью является формирование структуры, при которой разработчик знает, где искать то, что ему требуется, и в любой момент времени может досконально знать только ту часть системы, которая непосредственно относится к его работе. Напротив, в системе с большими, многоцелевыми классами нам неизбежно приходится разбираться с множеством аспектов, которые в данный момент нас не интересуют. Еще раз выделю **основные моменты**: *система должна состоять из множества мелких классов, а не из небольшого числа больших. Каждый класс инкапсулирует одну ответственность, имеет одну причину для изменения и взаимодействует с другими классами для реализации желаемого поведения системы.*