**Концепция чистого кода**

- предполагает написание кода, который легко читать, понимать, сопровождать и расширять.

**Форматирование:**

1. Использовать форматирование в соответствии с [конвенцией](https://www.oracle.com/technetwork/java/codeconventions-150003.pdf) или принятом на проекте стандартом.
2. Последовательность в порядке объявления членов класса: Поля, конструкторы, public методы, private методы.
3. Использовать логическое разделение класса на блоки с помощью дополнительных пустых строк.

**Именование**:

1. Должно раскрывать суть программного члена – должны быть достаточно информативными, чтобы читатель мог сразу понять его назначение.
2. Легко читаемо (не использовать нотации, сокращения, leet (буквы меняются на цифры) и другие вводящие в заблуждение конструкции).
3. Имя должно даваться в соответствии с областью видимости (переменные, используемые в небольших блоках кода, могут иметь короткие имена, тогда как глобальные переменные или переменные класса должны иметь более описательные имена).
4. Если переменная окружена контекстом, то его не нужно дублировать в имени (контекст, в котором используется переменная, часто уже даёт достаточно информации о её назначении).
5. Имена классов (подбираются исходя из их состояния, а не поведения) и переменных существительные, функций/методов – глаголы. Boolean переменные и методы, возвращающие Boolean называются предикативно (с использованием can/is). Утилитарные классы обычно заканчиваются на s (Arrays, Collections и др.)

**Методы**:

1. Длина метода не более 10 строк (в идеале) или чтобы метод помещался на экран. Для достижения использовать **extract till you drop** – методика рефакторинга, при котором функционал распухшего метода выделяется в более мелкие и простые, процесс повторяется до тех пор, пока извлекать станет нечего.
2. Соблюдение SRP – Single Responsibility Principle (также достигается за счет методики **ETYD**)
3. Избегать Магических чисел – не хардкодить в методах значения, а выносить их в именованные константы (создавать для них понятный контекст).
4. Объявлять переменные как можно ближе к их первому использованию (объявление всех переменных в начале вынуждает погружаться в лишний контекст)
5. Использовать подходящую конструкцию условного выражения для упрощения восприятия кода (вроде как switch предпочтительно избегать из-за громоздкости и нетипичного синтаксиса).
6. Использовать описательные переменные или методы для пересечения флагов в условных выражениях.

//без использования описательной переменной  
if (user.isActive() && !user.isSuspended() && user.hasPermission("edit")) {  
 editContent();  
}  
//с использованием описательной переменной  
boolean canEditContent = user.isActive() && !user.isSuspended() && user.hasPermission("edit");  
if (canEditContent) {  
 editContent();  
}

1. Предпочитать позитивные условные выражения (если использовать негативный, то при необходимости проверки положительного утверждения нужно использовать двойное отрицание)

//отрицательный флаг  
if (!(order.isNotProcessed())) {  
 processOrder();  
}  
//положительный флаг  
if (order.isProcessed()) {  
 processOrder();  
}

1. Использовать **guard clause** – техника, которая подразумевает проверку предусловий в начале метода и выход в случае невыполнения условий (улучшает читаемость, избавляет от вложенных if, предотвращает исполнение лишнего кода).

// Без использования guard clause  
if (b != 0) {  
 int result = a / b;  
 System.out.println("Результат деления: " + result);  
} else {  
 System.out.println("Деление на ноль");  
}  
//развернем условие,   
// тем самым вынесем условие для основного кода метода наверх  
if (b == 0) {  
 System.out.println("Деление на ноль");  
 return;  
}  
int result = a / b;  
System.out.println("Результат деления: " + result);

1. В теле функции все должно быть на одном уровне абстракции.
2. Минимизация количества аргументов (затрудняет понимание назначения метода). В идеале метод должен получать аргументы из окружения. Для достижения использовать объекты для группировки связанных аргументов, дефолтные значения и перегрузку методов.
3. Избегать использование Boolean аргументов для методов. Предпочтительным способом будет разделить метод на несколько, но без флага, а его назначение отразить в названии.

public void configure(String host, int port, boolean useSSL) {  
 if (useSSL) {  
 // Логика для использования SSL  
 } else {  
 // Логика для обычного подключения  
 }  
}

//разделенный метод  
public void configureWithSSL(String host, int port) {  
 // Логика для использования SSL  
}  
  
public void configureWithoutSSL(String host, int port) {  
 // Логика для обычного подключения  
}

1. Метод не должен иметь сайд эффектов – то есть метод должен выполнять операцию, для которой он предназначен, не изменяя при этом окружения.
2. Менять объект, переданный в качестве параметров может только void т.к. если возвращается что-то другое, подразумевается, что объект, на котором вызвался метод не изменялся.
3. Не возвращать null:
   1. вызывающий такой метод код обязан проверять на null
   2. в разработке предпочтительным является подход fail fast (раннее сигнализирование об ошибке) – если, например, вызывается метод, который положит возвращаемое значение в коллекцию, то этот null обнаружится спустя время и обнаружить следы его возникновения будет проблематично.

Альтернативой null является: если ситуация нештатная, то бросается исключение (например, objectNotFound). Если семантика метода подразумевает возвращение null как значение, то использовать **Null-object** или Optional. Если метод все-таки возвращает null, то клиент метода должен быть осведомлен об этом с помощью названия метода или аннотации @Nullable. [Материал.](https://javarush.com/groups/posts/1147-pochemu-null---ehto-plokho)

**Комментарии:**

В идеале код должен быть самодокументирован (должен быть понятен без комментария) за счет соответствующего именования его структур. Должны использоваться только в случае, если нельзя раскрыть детали реализации в виде кода (то есть отвечать на вопрос почему – например описание выбранного алгоритма, почему совершаются неочевидные действия в коде, предупреждение о последствиях рефакторинга). Если нет возможности отрефакторить здесь и сейчас, то возможно написание комментария в виде toDO.

**Прочие рекомендации:**

1. Следование закону **Деметры** (принцип наименьшего знания) – каждый программный модуль должен обладать ограниченным знанием о других модулях (знает только о модулях, которые имеют непосредственное к нему отношение). Или же – никакой объект не должен раскрывать свою структуру тем, кто к нему обращается. Направлен на достижение слабой связанности между модулями (law coupling по GRASP паттернам) и улучшение инкапсуляции (минимизации зависимостей между классами).

*Пример:*

public class PaperBoy {  
   
 public void DeliverMagazine(int cost, Customer customer) {  
  
 Wallet wallet = customer.getWallet();  
 //логика взятия оплаты  
 }  
}

Класс PaperBoy напрямую взаимодействует с полем класса Customer и изменяет его состояние. В данном случае класс PaperBoy взаимодействует сразу с двумя классами (Customer и Wallet), в таком случае изменения в классах повлекут за собой необходимость изменения логики класса PaperBoy.

public class PaperBoy {  
  
 public void DeliverMagazine(int cost, Customer customer) {  
 customer.getPayment(cost);  
 //логика взятия оплаты  
 }  
}

В исправленном варианте классу Customer был добавлен метод getPayment(), который инкапсулирует работу с Wallet, теперь изменения в этом классе никак не повлияют на класс PaperBoy.

\*\*Управлять не ногами собаки, если нужно чтобы она подошла, а самой собакой, она знает как ими управлять ☺ \*\*

1. **DRY (don’t repeat yourself) –** каждая часть знания должна иметь единственное непротиворечивое и авторитетное представление в системе.   
   То есть принцип направлен на избегание дублирования данных (удаление избыточности данных) и избегание дублирования кода, написанного ранее (код должен быть написан таким образом, чтобы его можно было переиспользовать).

Дублирование знания приводит к тому, что при необходимости изменения этого знания необходимо изменять все его источники.

1. **KISS (keep it short and simple) –** принцип направлен на достижение понятности кода через его простоту. Под простотой понимается избегание лишнего усложнения, если системе это не требуется в данный момент и использование инструментов по назначению.

Частные примеры:

1. Не реализовывать функционал, который не нужен в данный момент, но в перспективе когда-то может пригодится (он не пригодится).
2. Избегать увеличения уровней абстракции, если этого можно избежать
3. Избегать подключения лишних зависимостей, если от нее нужно несколько функций
4. Избегать излишнего использования паттернов, если они добавляют ненужную сложность.
5. И другие

Исходя из примеров, можно сказать, что принцип говорит о том, что необходимо избегать усложнения, если это возможно, если это невозможно, то подобрать верный способ этого усложнения.

Принцип носит довольно абстрактный характер и не предлагает каких-то конкретных решений, по сути подразумевает все остальные принципы проектирования, чтобы добиться простоты кода.

**SOLID –** акроним, объединяющий 5 принципов проектирования и разработки для написания гибкого, расширяемого и поддерживаемого кода. Заточены на уменьшение связанности

**S**ingle responsibility principle (принцип единственной ответственности) – у класса должна быть одна причина для изменения (1 класс – одна ответственность). Принцип может быть применен на уровне функций, класса, модуля.

SRP также можно охарактеризовать как стремление к высокой связности – *high cohesion* для одинакового функционала и *low coupling* для разного.

Высокая связность (High Cohesion) - означает, что компоненты внутри модуля или класса тесно связаны друг с другом и работают вместе для выполнения определенной задачи.

Низкая связанность (Low Coupling) – означает, что один модуль или класс не зависит от других модулей или классов. Когда модули или классы имеют высокую связанность, изменение в одном из них может потребовать изменений в других.

Зачем – уменьшение ответственности модуля (изменение одной ответственности может повлиять на другую ответственность в другой части системы), упрощение понимания кода, упрощение тестирования, повышение гибкости и возможность переиспользования.

**O**pen-closed principle (принцип открытости и закрытости) – программные сущности должны быть открыты для расширения, но закрыт для изменения.   
То есть добавление новой функциональности, не изменяя существующую за счет добавления новых сущностей.

Зачем – не нужно регрессионное тестирование (тестируется только дописанный функционал), обратная совместимость – ранее написанный код наверняка используется и его изменение может привести, во-первых, к ошибкам в этом коде, во-вторых к неожиданному поведению для клиентов.

Два подхода:

Майер – расширять функционал через наследование, при этом если базовый класс реализует интерфейс, то можно как следовать его реализации, так и изменить ее (вызывает сайд эффекты) – схож с паттерном protected variation (GRASP).

Дядя Боб – расширение функционала через динамическую диспетчеризацию и использование полиморфизма. Клиентский код зависит от неизменного интерфейса, при этом есть возможность делегировать работу старому коду. Отражается в паттернах Strategy и template method.

**L**iskov Substitution Principle (принцип подстановки Барбары Лисков) – подтипы должны быть совместимы со своими базовыми типами. То есть функции, которые используют базовый тип должны иметь возможность использовать подтипы базового типа, как если бы они использовали базовый тип (клиент не должен сталкиваться с разным поведением при использовании разных наследников одного и того же типа).

контрактные

Вводит ограничения для дочернего кода при использовании наследования ([подробно](https://habr.com/ru/articles/739734/)):

1. *Сигнатура:*

Аргументы – тип аргумента не должен быть ковариантен (в Java поддерживается на уровне языка, т.к. переопределение метода требует идентичные аргументы, а их расширение/сужение это уже перегрузка. См. пример.)

Возвращаемый тип – должен быть ковариантным (также поддерживается на уровне языка – не позволяет использовать контрварианты). Пример

Исключения – исключения не должны быть контрвариантны и не могут выбрасывать больше исключений, чем родитель.

1. *Настройки:*

Инварианты – утверждение относительно свойств объекта, которое должно быть истинным для всех допустимых состояний объекта. Можно усиливать инварианты.

1. *Свойства:*

Предусловия – можно ослаблять

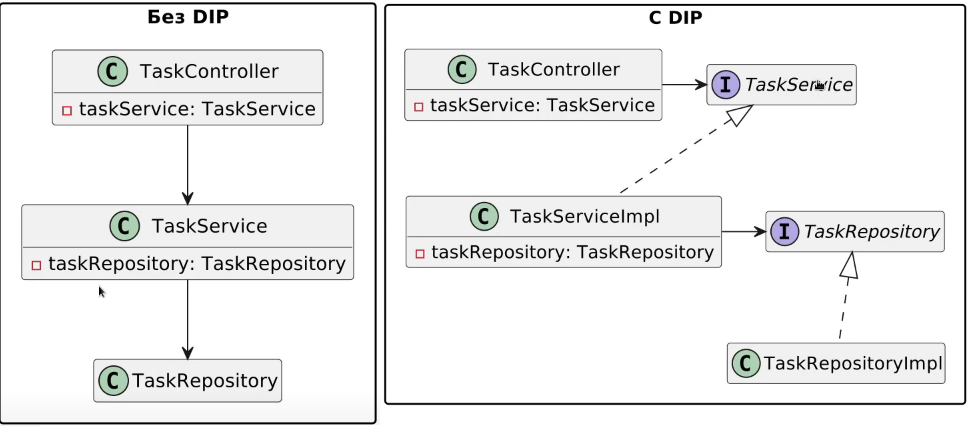
Постусловия – может усилить

**I**nterface Segregation Principle (принцип разделения интерфейсов) – клиент не должен зависеть от интерфейсов, которые он не использует. Достигается за счет декомпозиции интерфейсов на более специализированные.

Использование принципа увеличивает связность (cohesion) и уменьшает связанность (coupling). Схож с RSP использованием декомпозиции и приводит к тем же последствиям, что и RSP. Также связан с LSP, так как нарушение ISP приводит к нарушению LSP.

**D**ependency Inversion Principle (принцип инверсии зависимостей) – модули высокого уровня не должны зависеть от модулей низкого уровня, они должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

В общем смысле подразумевает избегание прямой зависимости между классами, вместо этого класс более высокого уровня должен зависеть от интерфейса, который будет реализован этим классом.



DIP и OCP связаны, так как условием их соблюдения является выделение неизменяемого интерфейса. Их несоблюдение ведет к высокой связанности между компонентами, что затрудняет добавление нового функционала, не изменяя текущую реализацию (без DIP – невозможно добавить новый функционал TaskService, т.к. придется изменить либо сам класс TaskService, либо добавить код в TaskController)

Видеоматериалы: [1](https://www.youtube.com/playlist?list=PLBheEHDcG7-nJsPVaejwcth7r9WhWWPdI), [2](https://www.youtube.com/playlist?list=PLs_aLxm3VDLuMvKZRuhKzOIuqBfah6SdU), [3 (приоритетный)](https://www.youtube.com/playlist?list=PL7GaNwuzA5StUNiOJFulHSD1SKkLdQYIx).