**Принципы разработки ПО**

1. YAGNI

*You Aren’t Gonna Need It / Вам это не понадобится*  
  
Этот принцип прост и очевиден, но ему далеко не все следуют. Если пишете код, то будьте уверены, что он вам понадобится. Не пишите код, если думаете, что он пригодится позже.  
  
Этот принцип применим при рефакторинге. Если вы занимаетесь рефакторингом метода, класса или файла, не бойтесь удалять лишние методы. Даже если раньше они были полезны – теперь они не нужны.  
  
Может наступить день, когда они снова понадобятся – тогда вы сможете воспользоваться git-репозиторием, чтобы воскресить их из мертвых.

### 2. DRY

Использование SSOT позволит создать более прочную и понятную кодовую базу.  
  
Дублирование кода – пустая трата времени и ресурсов. Вам придется поддерживать одну и ту же логику и тестировать код сразу в двух местах, причем если вы измените код в одном месте, его нужно будет изменить и в другом.  
  
В большинстве случаев дублирование кода происходит из-за незнания системы. Прежде чем что-либо писать, проявите прагматизм: осмотритесь. Возможно, эта функция где-то реализована. Возможно, эта бизнес-логика существует в другом месте. Повторное использование кода – всегда разумное решение.

3. KISS

*Keep It Simple, Stupid / Будь проще*  
  
Применительно к разработке ПО он значит следующее – не придумывайте к задаче более сложного решения, чем ей требуется.  
  
Иногда самое разумное решение оказывается и самым простым. Написание производительного, эффективного и простого кода – это прекрасно.  
  
Одна из самых распространенных ошибок нашего времени – использование новых инструментов исключительно из-за того, что они блестят. Разработчиков следует мотивировать использовать новейшие технологии не потому, что они новые, а потому что они подходят для работы.

### 4. Big Design Up Front

Многие разработчики считают, что если они не пишут код, то они не добиваются прогресса. Это неверный подход. Составив план, вы избавите себя от необходимости раз за разом начинать с нуля.  
  
Иногда в недостатках и процессах разработки архитектуры должны быть замешаны и другие люди. Чем раньше вы все это обсудите, тем лучше будет для всех.  
  
Очень распространенный контраргумент заключается в том, что стоимость решения проблем зачастую ниже стоимости времени планирования. Чем с меньшим количеством ошибок столкнется пользователь, тем лучше будет его опыт. У вас может не быть другого шанса справиться с этими ошибками.

5. SOLID

Это наиболее известный принцип разработки ПО. Solid — это аббревиатура от:  
  
**S) Single-responsibility principle /Принцип единственной ответственности**  
  
Его важность невозможно переоценить. Каждый объект, класс и метод должны отвечать только за что-то одно. Если ваш объект/класс/метод делает слишком много, вы получите спагетти-код.

**O) Open–closed principle / Принцип открытости-закрытости**  
  
Программные объекты должны быть открыты для расширения, но закрыты для модификации. Речь о том, что нельзя переопределять методы или классы, просто добавляя дополнительные функции по мере необходимости.

**L) Liskov substitution principle / Принцип подстановки Лисков**  
  
Этот принцип гласит, что объекты старших классов должны быть заменимы объектами подклассов, и приложение при такой замене должно работать так, как ожидается.  
  
**I) Interface segregation principle / Принцип разделения интерфейсов**

Объекты не должны зависеть от интерфейсов, которые они не используют  
ПО должно разделяться на независимые части. Побочные эффекты необходимо сводить к минимуму, чтобы обеспечивать независимость.  
  
Убедитесь, что вы не заставляете объекты реализовывать методы, которые им никогда не понадобятся. Вот пример:

**D) Dependency inversion principle / Принцип инверсии зависимостей**  
  
Этот принцип невозможно переоценить. Мы должны полагаться на абстракции, а не на конкретные реализации. Компоненты ПО должны иметь низкую связность и высокую согласованность.

6. Avoid Premature Optimization

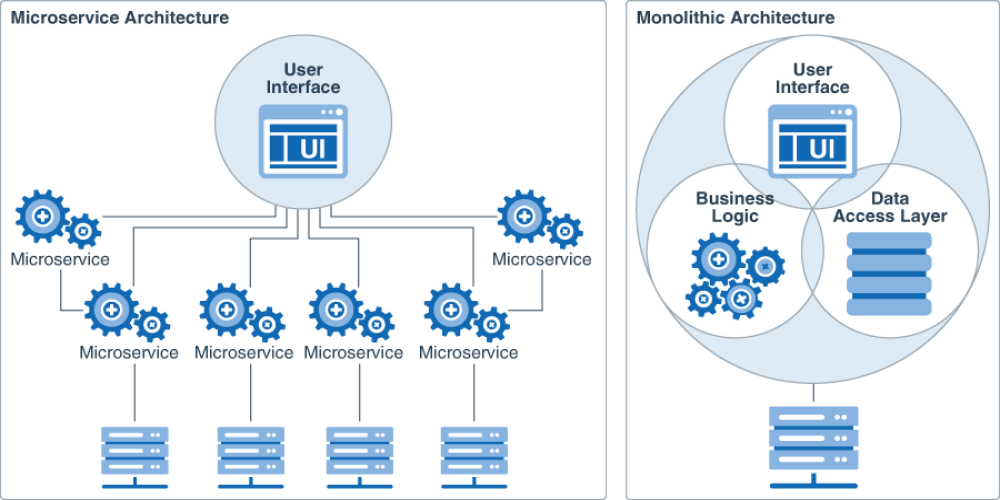
*Избегайте преждевременной оптимизации*  
  
Эта практика побуждает разработчиков оптимизировать код до того, как необходимость этой оптимизации будет доказана. Думаю, что если вы следуете KISS или YAGNI, вы не попадетесь на этот крючок.  
Очень простой пример – масштабирование. Вы не станете покупать 40 серверов из предположения, что ваше новое приложение станет очень популярным. Вы будете добавлять серверы по мере необходимости.

7. Бритва Оккама  
  
Что это значит в мире программирования? Не создавайте ненужных сущностей без необходимости. Будьте прагматичны — подумайте, нужны ли они, поскольку они могут в конечном итоге усложнить вашу кодовую базу.

Микросервис vs Монолит

Монолитное приложение (назовем его монолит) представляет собой приложение, доставляемое через единое развертывание. Давайте представим классический интернет-магазин. Стандартные модули: UI, бизнес-логика и дата-слой. Возможны способы взаимодействия с сервисом: API REST и веб-интерфейс.

При построении монолита все эти вещи будут управляться внутри одного и того же модуля.

Рассмотрите пример на следующем рисунке, где все части находятся в одном и том же модуле развертывания:

Большим преимуществом монолита является то, что его легче реализовать. В монолитной архитектуре вы можете быстро начать реализовывать свою бизнес-логику, вместо того чтобы тратить время на размышления о межпроцессном взаимодействие.

Еще одна вещь — это сквозные (E2E) тесты. В монолитной архитектуре их легче выполнить.

Говоря об операциях, важно сказать, что монолит прост в развертывании и легко масштабируется. Для развертывания вы можете использовать скрипт, загружающий ваш модуль и запускающий приложение.

### Недостатки

Монолиты, как правило, перерождаются из своего чистого состояния в так называемый «большой шарик грязи». Вкратце это описывается как состояние, возникшее, потому что архитектурные правила были нарушены и со временем компоненты срослись.

В монолите практически нет изоляции. Проблема или ошибка в модуле может замедлить или разрушить все приложение.

В микросервисной архитектуре слабо связанные сервисы взаимодействуют друг с другом для выполнения задач, относящихся к их бизнес-возможностям.

Микросервисы в значительной степени получили свое название из-за того, что сервисы здесь меньше, чем в монолитной среде. Тем не менее, микро — о бизнес-возможностях, а не о размере.

По сравнению с монолитом в микросервисах у вас есть несколько единиц развертывания. Каждый сервис развертывается самостоятельно.

В больших компаниях разные сервисы могут принадлежать разным командам. Услуги могут быть повторно использованы всей компанией. Это также позволяет командам работать над услугами в основном самостоятельно. Нет необходимости координировать развертывание между командами. Развивать сервисы лучше с увеличением количества команд.

Микросервисы меньше, и благодаря этому их легче понять и проверить.

Меньшие размеры помогают, когда речь идет о времени компиляции, времени запуска и времени, необходимом для выполнения тестов. Все эти факторы влияют на производительность разработчика, так как позволяют затрачивать меньше времени на ожидание на каждом этапе разработки.

Более короткое время запуска и возможность развертывания микросервисов независимо друг от друга действительно выгодны для CI / CD. По сравнению с обычным монолитом он намного плавнее.

### Недостатки

Все звучит довольно хорошо, но есть и недостатки.

Распределенная система имеет свою сложность: в ней вам приходится иметь дело с частичным отказом, более затруднительным взаимодействием при тестировании (тесты E2E), а также с более высокой сложностью при реализации взаимодействия между сервисами.

Существуют эксплуатационные накладные расходы, а множество микросервисов сложнее в эксплуатации, чем несколько экземпляров сигнального монолита.

Помимо вышеперечисленных сложностей, для микросервисов также может потребоваться больше оборудования, чем для традиционных монолитов. Иногда микросервисы могут превзойти один монолит, если есть его части, которые требуют масштабирования до предела.

Изменения, затрагивающие несколько сервисов, должны координироваться между несколькими командами

**CI/CD**

### Непрерывная интеграция

Разработчики, применяющие непрерывную интеграцию, при каждой возможности выполняют слияние своих изменений с основной веткой. Изменения, внесенные разработчиком, проверяются путем создания сборки и запуска автоматических тестов на этой сборке. С таким подходом вы избегаете сложностей при интеграции, когда нужно ждать дня релиза, чтобы выполнить слияние изменений в соответствующей ветке.

При использовании непрерывной интеграции уделяется большое внимание автоматизации тестирования, в результате которого при интеграции новых коммитов в основную ветку работа приложения не нарушается.

### Непрерывная поставка

[Непрерывная поставка](https://www.atlassian.com/ru/continuous-delivery) является продолжением непрерывной интеграции, поскольку при ней происходит автоматическое развертывание всех изменений кода в тестовой и (или) рабочей среде после этапа сборки.

Это значит, что автоматизирован не только процесс тестирования, но и процесс выпуска продукта, поэтому приложение можно развернуть в любое время одним нажатием.

Теоретически при непрерывной поставке вы можете выпускать релизы ежедневно, еженедельно, каждые две недели или с любой другой периодичностью, актуальной для бизнеса. Однако если вы действительно хотите получить преимущества от непрерывной поставки, следует выполнять развертывание в рабочей среде как можно раньше, обеспечивая выпуск небольших пакетов изменений, в которых легко найти ошибку в случае проблем.

### Непрерывное развертывание

[Непрерывное развертывание](https://www.atlassian.com/continuous-delivery/continuous-deployment) идет на один шаг дальше, чем непрерывная поставка. При этом подходе каждое изменение, которое проходит все стадии производственного процесса, выпускается клиентам. Вмешательство человека не требуется, и развертыванию нового изменения в рабочую среду может помешать только ошибка во время теста.

Непрерывное развертывание — это отличный способ ускорить цикл обратной связи с клиентами и избавить команду от лишнего напряжения, потому что Дня релиза больше не бывает. Разработчики могут сосредоточиться на создании ПО. Они видят, как их код запускается в работу за считанные минуты, стоит только закончить.

BDD

* Когда мы работаем по методологии BDD, автотестирование и составление спецификации сопровождает каждый этап цикла разработки ПО, что обеспечивает постоянную актуальность автотестов и документации.
* BDD универсален и может включаться на любом этапе разработки.
* BDD еще полезен тем, что не нужно тратить много времени на написание разного рода документации. При классической схеме разработки нужны, как минимум, спецификации и тестовые сценарии, которые обычно пишут разные люди. В BDD спецификация является тестовым сценарием, одновременно являясь и автотестом. Тестировщикам не нужно писать отдельную тестовую документацию — за них это уже сделал аналитик, написавший спецификацию из конструкций естественного языка (которая читаема и понятна любому члену команды).

## **BDD всем хорош, но почему его не используют?**

Ответ прост: это долго и дорого. С этим утверждением согласятся большинство IT компаний. И поначалу мы не были исключением. BDD неудобен хотя бы тем, что требует привлечения специалистов тестирования уже на этапе проработки требований.

BDD переворачивает с ног на голову классическую схему ведения разработки (TLD). Она плохо реализуема, потому что это сложно. Удлиняется цикл разработки

BDD — это несомненно способ достичь качества. Но не все готовы платить временем и специалистами за это качество.

## **В чем разница**

* TDD хорошо подходит для юнит-тестирования, т.е. для проверки работы отдельных модулей самих по себе. BDD — для интеграционного (т.е. для проверки, как отдельные модули работают друг с другом) и e2e (т.е. для проверки всей системы целиком) тестирования.
* TDD: тесты сразу реализуются в коде, для BDD чаще всего описываются шаги на языке, понятном всем, а не только разработчикам.
* TDD: юнит-тесты пишут сами разработчики. BDD требует объедения усилий разных членов команды. Обычно тест-кейсы (шаги) описываются ручным тестировщиком или аналитиком и воплощаются в код тестировщиком-автоматизатором. В нашей команде мы (фронтенедеры) описываем шаги вместе с тестировщиками, а код тестов пишет фронтенд-команда.
* TDD проверяет работу функций, BDD — пользовательские сценарии.

BDD позволяет всем участникам общаться на одном языке. Будь то BA, DEV или еще кто-то – clarity есть отличительная черта. Имена тестов в bdd представлены в виде юзер стори. Даже ba может просто прочитать код, а также написать код. Но на практике это почти никогда не происходит, ба не пишет код, а если и пишет, то часто неправильно, плюс все это нужно потом поддерживать. Создается язык, словарь грубо говоря какой-то, и потом на нем все пишется все. Но все это не проходит бесследно, так как прежде всего нужно больше времени на разработку и поддержку, соответственно это дорого. Некоторые даже придерживаются мнения, что можно просто делать отчеты на java bdd с помощью того же spok(groovy) или spek(kotlin).

Gherkin

As a store owner

In order to keep track of stock

**I** want to add items back to stock when they're returned.

Scenario 1: Refunded items should be returned to stock

**Given** that a customer previously bought a black sweater from me

**And** I have three black sweaters in stock.

**When** they return the black sweater for a refund

**Then** I should have four black sweaters in stock.

Scenario 2: Replaced items should be returned to stock

**Given** that a customer previously bought a blue garment from me

**And** I have two blue garments in stock

**And** three black garments in stock.

**When** they return the blue garment for a replacement in black

**Then** I should have three blue garments in stock

**And** two black garments in stock.

**Особенности тестирования микросервисов**

Но многие практикующие тестировщики при работе с микросервисами склонны ограничиваться модульными тестами. Тогда как модульный тест нормально покрывает единственную функцию, работающую в очень ограниченных тестовых условиях, микросервис рассчитан на обслуживание аудитории в рамках всего Веба. Поэтому условия тестирования должны быть экстремальными.

Например, в рамках хорошего теста на микроуровне можно одновременно прогнать сто тысяч экземпляров микросервиса и посмотреть, как они поведут себя в таком масштабе. Недостаточно за раз проверять всего одну функцию, применяя к ней единственный модульный тест. Необходимо прогонять такие тесты на тысячах экземпляров функции одновременно, с учетом показателей хостинговой среды, в которой придется работать.

Наряду с тестированием функциональности микросервиса, также необходимо позаботиться о тестировании единицы развертывания, в рамках которой происходит релиз микросервиса. Как правило, микросервисы развертываются в виде [контейнеров](https://ru.wikipedia.org/wiki/Docker) в рамках той или иной технологии оркестрации, например, [Kubernetes](https://www.piter.com/product_by_id/225047643?recommended_by=instant_search&r46_search_query=kubernetes%3A%20%D0%BB%D1%83%D1%87%D1%88%D0%B8%D0%B5%20%D0%BF%D1%80%D0%B0%D0%BA%D1%82%D0%B8%D0%BA%D0%B8) или [Docker Swarm](https://docs.docker.com/engine/swarm/). Важный аспект оркестрации контейнеров – обеспечение долговременной устойчивости микросервисов.  
Считается, что микросервисы могут отваливаться по целому ряду причин, как из-за отказа хоста, так и из-за ошибок самого микросервиса. Вполне нормально, что в ходе одновременной эксплуатации тысяч контейнеров возникают какие-то текущие неисправности. Важность тестирования в том, что правильный [выход микросервиса из игры](https://12factor.net/disposability) не менее важен, чем его корректное функционирование. Тесты на микроуровне гарантируют, что микросервис будет аккуратно оживать и аккуратно умирать, при любом масштабе системы.

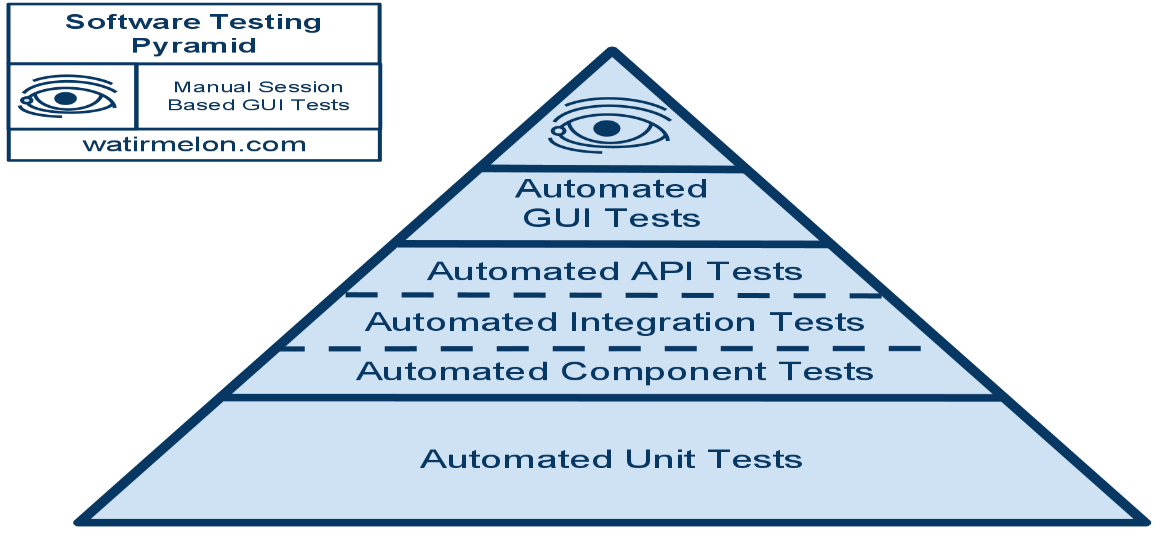
Убедитесь, что абсолютно все происходящее у вас логируется  
Тесты также должны гарантировать, что все существенные события, происходящие в микросервисе, как следует логируются – и, что не менее важно, эти записи логов можно понять. В мире микросервисов записи логов очень важны, в особенности это касается асинхронных MOA, где нет последовательного выполнения поведений. Зачастую только данные логов помогут вам осмыслить, что же творится в приложении.

**Линтеры и форматировщики**

Весь инструментарий, доступный разработчикам Python, можно условно разделить на две группы по способу реагирования на ошибки. Первая группа сообщает о найденных ошибках, перекладывая задачу по их исправлению на программиста. Вторая — предлагает пользователю вариант исправленного кода или автоматически вносит изменения.

И первая, и вторая группы включают в себя как простые утилиты командной строки для решения узкоспециализированных задач (например, проверка docstring или сортировка импортов), так и богатые по возможностям библиотеки, объединяющие в себе более простые утилиты. Средства анализа кода из первой группы принято называть линтерами (linter). Название происходит от lint — статического анализатора для языка программирования Си и со временем ставшего нарицательным. Программы второй группы называют форматировщиками (formatter).

Даже при поверхностном сравнении этих групп видны особенности работы с ними. При применении линтеров программисту, во-первых, необходимо писать код с оглядкой, дабы позже не исправлять найденные ошибки. И во вторых, принимать решение по поводу обнаруженных ошибок — какие требуют исправления, а какие можно проигнорировать. Форматировщики, напротив, автоматизируют процесс исправления ошибок, оставляя программисту возможность осуществлять контроль.



A stub is a simple fake object. It just makes sure test runs smoothly.  
A mock is a smarter stub. You verify your test passes through it.

Here's a description of each one followed by with real world sample.

* **Dummy** - just bogus values to satisfy the API.

Example: If you're testing a method of a class which requires many mandatory parameters in a constructor which have no effect on your test, then you may create dummy objects for the purpose of creating new instances of a class.

* **Fake** - create a test implementation of a class which may have a dependency on some external infrastructure. (It's good practice that your unit test does **NOT** actually interact with external infrastructure.)

Example: Create fake implementation for accessing a database, replace it with in-memory collection.

* **Stub** - override methods to return hard-coded values, also referred to as state-based.

Example: Your test class depends on a method Calculate() taking 5 minutes to complete. Rather than wait for 5 minutes you can replace its real implementation with stub that returns hard-coded values; taking only a small fraction of the time.

* **Mock** - very similar to Stub but interaction-based rather than state-based. This means you don't expect from Mock to return some value, but to assume that specific order of method calls are made.

Example: You're testing a user registration class. After calling Save, it should call SendConfirmationEmail.

Stubs and Mocks are actually sub types of Mock, both swap real implementation with test implementation, but for different, specific reasons.

* **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
* **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an [in memory database](https://martinfowler.com/bliki/InMemoryTestDatabase.html) is a good example).
* **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
* **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
* **Mocks** are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

!!Cerberus for validating schema!!

**Принципы ООП:**

* Данные структурируются в виде объектов, каждый из которых имеет определенный тип, то есть принадлежит к какому-либо классу.
* Классы – результат формализации решаемой задачи, выделения главных ее аспектов.
* Внутри объекта инкапсулируется логика работы с относящейся к нему информацией.
* Объекты в программе взаимодействуют друг с другом, обмениваются запросами и ответами.
* При этом объекты одного типа сходным образом отвечают на одни и те же запросы.
* Объекты могут организовываться в более сложные структуры, например, включать другие объекты или наследовать от одного или нескольких объектов.

**What is Docker**

Простыми словами, Докер это инструмент, который позволяет разработчикам, системными администраторам и другим специалистам деплоить их приложения в песочнице (которые называются контейнерами), для запуска на целевой операционной системе, например, Linux. Ключевое преимущество Докера в том, что он позволяет пользователям **упаковать приложение со всеми его зависимостями в стандартизированный модуль** для разработки. В отличие от виртуальных машин, контейнеры не создают такой дополнительной нагрузки, поэтому с ними можно использовать систему и ресурсы более эффективно.

### Что такое контейнер?

Стандарт в индустрии на сегодняшний день — это использовать виртуальные машины для запуска приложений. Виртуальные машины запускают приложения внутри гостевой операционной системы, которая работает на виртуальном железе основной операционной системы сервера.

Виртуальные машины отлично подходят для полной изоляции процесса для приложения: почти никакие проблемы основной операционной системы не могут повлиять на софт гостевой ОС, и наоборот. Но за такую изоляцию приходится платить. Существует значительная вычислительная нагрузка, необходимая для виртуализации железа гостевой ОС.

Контейнеры используют другой подход: они предоставляют схожий с виртуальными машинами уровень изоляции, но благодаря правильному задействованию низкоуровневых механизмов основной операционной системы делают это с в разы меньшей нагрузкой.

### Почему я должен использовать их?

Взлет Докера был по-настоящему эпичным. Не смотря на то, что контейнеры сами по себе — не новая технология, до Докера они не были так распространены и популярны. Докер изменил ситуацию, предоставив стандартный API, который сильно упростил создание и использование контейнеров, и позволил сообществу вместе работать над библиотеками по работе с контейнерами. В статье, опубликованной в  [The Register](http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/) в середине 2014 говорится, что Гугл поддерживает больше **двух миллиардов контейнеров в неделю**.

**ETL и ELT**

1. ETL – это процесс извлечения, преобразования и загрузки данных. ELT – это процесс извлечения, загрузки и преобразования данных.
2. В рамках ETL данные перемещаются из источника данных в промежуточное хранилище данных.
3. Для выполнения основных преобразований ELT использует хранилище данных. Нет необходимости в промежуточном размещении данных.
4. ETL может помочь обеспечить конфиденциальность и соответствия требованиям, очищая конфиденциальные и безопасные данные еще до их загрузки в хранилище данных.
5. ETL может выполнять сложные преобразования данных и может быть более рентабельным, чем ELT.

ETL и ELT необходимы в анализе о данных, потому что источники информации, независимо от того, используют ли они структурированную базу данных SQL или неструктурированную базу данных NoSQL, редко используют одинаковые или совместимые форматы. Следовательно, вам необходимо очистить, обогатить и преобразовать свои источники данных, прежде чем объединить их в анализируемое целое. Таким образом, ваша платформа бизнес-аналитики сможет анализировать данные для получения аналитических сведений.

Независимо от того, используете ли вы ETL или ELT, процесс преобразования/интеграции данных включает следующие три этапа:

* Извлечение**(E):** извлечение относится к извлечению исходных данных из исходной базы данных или источника данных. С помощью ETL данные попадают во временную промежуточную область. С ELT они сразу попадают в систему хранения озера данных.
* **Преобразование (T):** преобразование относится к процессу изменения структуры информации, так что она интегрируется с целевой системой данных и остальными данными в этой системе.
* **Загрузка (L):** загрузка относится к процессу размещения информации в системе хранения данных.

**Непрерывный процесс с четко определенным рабочим процессом:** сначала ETL извлекает данные из однородных или разнородных источников данных. Затем он помещает данные в промежуточную область. **Оттуда данные проходят процесс очистки, обогащаются и трансформируются и, наконец, сохраняются в хранилище данных**.

Одно из крупнейших преимуществ ETL перед ELT связано с преструктурированной природой хранилища данных OLAP. После структурирования/преобразования данных ETL позволяет проводить ускоренный, более эффективный и стабильный анализ данных. А вот ELT не так идеален, когда задача требует быстрого анализа.

**Что такое ELT?**

ELT означает «Извлечь, загрузить и преобразовать». В этом процессе данные для основных преобразований используются через хранилище данных. Это означает, что нет необходимости в промежуточном размещении данных. Для всех типов данных, включая структурированные, неструктурированные, полуструктурированные и даже необработанные данные ELT использует облачные решения для их хранения.

**Самые значительные преимущества ELT**

Основное преимущество ELT перед ETL связано с гибкостью и простотой хранения новых неструктурированных данных. С помощью ELT вы можете хранить любой тип информации, даже если у вас нет времени или возможности сначала преобразовать и структурировать ее, обеспечивая немедленный доступ ко всей вашей информации, когда вы этого захотите. Кроме того, вам не нужно разрабатывать сложные процессы ETL перед загрузкой данных, что экономит время разработчиков и аналитиков данных при работе с новой информацией.

**Запахи кода**

Запахи кода — это ключевые признаки необходимости [рефакторинга](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3)

Дублирование кода — это использование одинаковых структур кода в нескольких местах. Объединение этих структур позволит улучшить программный код

#### Длинный метод

Среди объектных программ дольше всего живут программы с короткими [методами](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Чем длиннее процедура, тем труднее её понять. Если у метода хорошее название, то не нужно смотреть его тело[[3]](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D1%81_%D0%B7%D0%B0%D0%BF%D0%B0%D1%88%D0%BA%D0%BE%D0%BC#cite_note-_e4ecfa6ba667ff4e-3).

Следует придерживаться эвристического правила: если ощущается необходимость что-то прокомментировать, нужно написать метод. Даже одну строку имеет смысл выделить в метод, если она нуждается в разъяснениях[[7]](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D1%81_%D0%B7%D0%B0%D0%BF%D0%B0%D1%88%D0%BA%D0%BE%D0%BC#cite_note-_8526c0bcd71330f1-7).

* Для сокращения метода достаточно применить «Выделение метода» (Extract Method);

#### Большой класс

Когда [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) реализует слишком обширную функциональность, стоит подумать о вынесении некоторой части кода в подкласс. Это избавит разработчиков от чрезмерного количества имеющихся у класса атрибутов и дублирования кода[[7]](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D1%81_%D0%B7%D0%B0%D0%BF%D0%B0%D1%88%D0%BA%D0%BE%D0%BC#cite_note-_8526c0bcd71330f1-7).

* Для уменьшения класса используется «Выделение класса» (Extract Class) или «Выделение подкласса» (Extract Subclass). При этом следует обращать внимание на общность в названии атрибутов и на то, использует ли класс их все одновременно

#### Длинный список параметров

В длинных списках [параметров](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) трудно разбираться, они становятся противоречивыми и сложными в использовании. Использование [объектов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) позволяет, в случае изменения передаваемых данных, модифицировать только сам объект. Работая с объектами, следует передавать ровно столько, чтобы метод мог получить необходимые ему данные[[8]](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D1%81_%D0%B7%D0%B0%D0%BF%D0%B0%D1%88%D0%BA%D0%BE%D0%BC#cite_note-_8526c0bcd71330f2-8).

* «Замена параметра вызовом метода» (Replace Parameter with Method) применяется, когда можно получить данные путём вызова метода объекта. Этот объект может быть полем или другим параметром.

## **Что такое распределенная система?**

На базовом уровне распределенная система — это совокупность компьютеров, которые работают вместе, образуя единый компьютер для конечного пользователя. Все эти распределенные машины имеют одно общее состояние и работают одновременно.

Они могут выходить из строя независимо, не повреждая всю систему, как и микросервисы. Эти взаимозависимые автономные компьютеры связаны сетью, чтобы легко обмениваться информацией, общаться и обмениваться информацией.

Примечание. Распределенные системы должны иметь общую сеть для подключения своих компонентов, которые могут быть подключены с помощью IP-адреса или даже физических кабелей.

В отличие от традиционных баз данных, которые хранятся на одной машине, в распределенной системе пользователь должен иметь возможность связываться с любой машиной, не зная, что это только одна машина. Большинство приложений сегодня используют ту или иную форму распределенной базы данных и должны учитывать их однородный или неоднородный характер.

Как правило, существует три типа распределенных вычислительных систем со следующими целями:

* Распределенные информационные системы: распределяйте информацию по разным серверам с помощью нескольких моделей связи.
* Распределенные всеобъемлющие системы: используйте встроенные компьютерные устройства (например, мониторы ЭКГ, датчики, мобильные устройства)
* А также распределенные вычислительные системы: компьютеры в сети обмениваются данными посредством передачи сообщений.