**SOLID:**   
S: **Single Responsibility Principle** (Принцип единственной ответственности).   
Каждый класс должен решать лишь одну задачу.   
O: **Open-Closed Principle** (Принцип открытости-закрытости).   
Программные сущности должны быть открыты для расширения, но закрыты для модификации.   
L: **Liskov Substitution Principle**   
Функция должна уметь выполняться для наследников не зная об этом.   
I: **Interface Segregation Principle** (Принцип разделения интерфейса).   
Наследники не должны получать то, что им не нужно.   
D: **Dependency Inversion Principle** (Принцип инверсии зависимостей).   
Абстракции не должны зависеть от деталей.   
Детали должны зависеть от абстракций.

Позволяют писать легко масштабируемый, легко переиспользуемый, слабосвязанный код.

**Single Responsibility Principle**

Существует лишь одна причина, приводящая к изменению класса.

Использовать если изменение в коде, отвечающего за одну ответственность, приводит к необходимости изменения другого кода, отвечающего за другую ответственность.

Принцип служит для разделения типов поведения, благодаря которому ошибки, вызванные модификациями в одном поведении, не распространялись на прочие, не связанные с ним типы.

Но и не следует разбивать на слишком маленькие блоки, это приводит к избыточной сложности и усложнению тестирования.

**Пример:** Класс обработки заказа содержит метод обработки заказа, сохранения в бд и отправки сообщения на email. Нужно разделить на 3 класса.

**Open-Closed Principle**

Принцип служит для того, чтобы делать поведение класса более разнообразным, не вмешиваясь в текущие операции, которые он выполняет. Благодаря этому вы избегаете ошибок в тех фрагментах кода, где задействован этот класс. Поэтому система должна быть гибкой, с возможностью работы в переменных условиях без изменения исходного кода.

**Пример:**

Допустим мы реализуем интернет магазин. У нас есть класс OrderPayment для произведения оплаты заказов. Внутри он использует сервис кредитных карт для проведения платежей. Если мы будем использовать его напрямую, то привяжемся к конкретной реализации. И чтобы добавить другие способы оплаты нам придется менять сам класс (например в if проверять тип оплаты). То есть класс не закрыт для изменения.

Исправить это можно внедрением интерфейса для платежной системы, и метод оплаты заказа будет принимать этот интерфейс. Таким образом мы сможем реализовывать интерфейс, добавляя новые способы оплаты.

Но при наследовании может поменяться интерфейс класса, и придется изменять клиентский код. Есть полиморфная интерпретация принципа. Когда клиентский код работает с классом через интерфейс, и мы добавляем новую функциональность реализуя этот интерфейс, и делегируя задачи предыдущему классу. Либо можно наследоваться от предыдущего класса.

**Liskov Substitution Principle**

любая реализация абстракции (интерфейса) должна быть взаимозаменяемой в любом месте, в котором принимается эта абстракция.

Это означает, что класс, разработанный путем расширения на основании базового класса, должен переопределять его методы так, чтобы не нарушалась функциональность с точки зрения клиента. То есть, если разработчик расширяет ваш класс и использует его в приложении, он не должен изменять ожидаемое поведение переопределенных методов.

Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом. То есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.

Принцип служит для того, чтобы обеспечить постоянство: класс-родитель и класс-потомок могут использоваться одинаковым образом без нарушения работы программы.

**Пример:** Пусть имеется класс доставки. И одним из его полей будет поле адрес. И от него мы захотим наследовать класс самовывоза, чтобы использовать в нем какие-то базовые вещи. Но у самовывоза нет адреса, поэтому мы его игнорируем, а в гетере возвращаем null. Поэтому при использовании типа самовывоза, вместо базового класса, при получении адреса получим null.

Как решение можно вынести специфичные функциональности в отдельный интерфейс. В итоге будет базовый класс доставки, и у него 2 потомка, доставка по адресу, и самовывоз.

**Interface Segregation Principle**

Нельзя заставлять клиента реализовать интерфейс, которым он не пользуется.

Слишком толстые интерфейсы необходимо разделять на более мелкие и специфичные, чтобы клиенты маленьких интерфейсов знали только о методах, необходимых им в работе.

В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

Принцип служит для того, чтобы раздробить единый набор действий на ряд наборов поменьше – таким образом, каждый класс делает то, что от него действительно требуется, и ничего больше.

**Пример:** пусть есть интерфейс методов для фигуры, например объем, площадь и тд. Тогда при реализации этого интерфейса двумерной фигурой потребуется реализовывать метод Объем, который не имеет смысла для двумерной фигуры. Решенем будет разделить интерфейс на 2: Объем и Площадь.

Пусть имеется интерфейс доставка. И среди прочих есть есть метод GetAddress. При реализации этого интерфейса классом самовывоза, у нас появляется зависимость от метода getAddress, который мы не используем.

Решение : разбить на несколько интерфейсов. Базовый интерфейс доставки без адреса, и дополнительный интерфейс который будет содержать адрес.

**Dependency Inversion Principle**

Модули верхнего уровня не зависят от модулей нижнего уровня. Оба типа модулей должны зависить от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Программное обеспечение нужно разрабатывать так, чтобы различные модули были автономными и соединялись друг с другом с помощью абстракции.  
Этот принцип служит для того, чтобы устранить зависимость классов верхнего уровня от классов нижнего уровня за счёт введения интерфейсов.

Пусть имеется класс оформления заказа. Он использует какую то конкретную доставку заказа. В класс доставки передаем заказ. Получается что мы зависим от конкретной доставки (конкретной реализации). Если появится необходимость дополнять доставку различными деталями, это потребует изменений и в классе оформления заказа. Следовательно наш класс зависит от деталей.

Поэтому введем интерфейс доставка, и от него уже будет зависеть оформление заказа. То есть там, где будет описана доставка, там будут и детали.

IoC - это принцип ООП, используемый для уменьшения связанности между классами и объектами.

Предполагает что каждый компонент системы должен быть как можно более изолированным от других компонетов. Не полагаться на детали реализации других компонентов.

Также архитектурное решение интеграции, упрощающее расширение системы, при котором поток выполнения контролируется внешним компонентом (контейнер, фреймворк), что может уменьшить взаимосвязь компьютерного кода.

В классической программе программист сам решает, в какой последовательности делать вызовы процедур, то есть сам управляет потоком работы приложения.

Фреймворк берет управление потоком на себя, и вызывает код программиста, когда это потребуется (фреймворк управляет кодом программиста, а не программист управляет фреймворком). Программист размещает свой код в специально отведенных для этого местах(типо контроллеров). Происходит утеря контроля над выполнением кода – это и называется инверсия управления.

IoC — это основное различие между фреймворком и другой формализацией повторно используемого кода — библиотекой — набором функций, которые вы просто вызываете из своей программы. Фреймворк — это оболочка, которая предоставляет предопределенные точки расширения. Вы можете вставить свой собственный код в эти точки расширения, но фреймворк определяет, когда этот код будет вызван.

Чтобы следовать принципам инверсии управления необходимо:

* Реализовать компоненты, отвечающие за одну конкретную задачу.
* Компоненты должны быть максимально независимыми друг от друга.
* Компоненты не должны зависеть от конкретной реализации друг друга.

Таким образом упрощается переключение между различными реализациями, более легкая тестируемость.

Способы реализации:

* Factory
* Service Locater
* Dependency injection

**Factory –** Делегирование создания объектов специальному классу Фабрике. Который принадлежит интерфейсу Фабрика. Таким образом мы сможем менять производимые объекты меняя фабрики. Также имеется возможность добавить какой-нибудь код при создании объектов (например предоставить зависимости).

Хотя код и не зависит от низкоуровневых модулей, он зависит от класса фабрики.

**Service Locater –** реестр объектов (компонентов), который:

* Отдает по запросу необходимые объекты;
* Доступен в любом месте приложения.

Он знает как получить все объекты, которые, возможно, потребуются. Service Locater не создает объекты. Обычно они добавляются напрямую, например через конфигурационный файл.

Плюсы: позволяет компонентам не привязываться друг к другу.

Минусы: локатор – глобальный объект, может вызывать неявное поведение.

**Внедрение зависимости** (*Dependency injection*, DI) — процесс предоставления внешней зависимости программному компоненту. объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

Эти зависимости передаем в качестве параметров конструктора, или сетера. При создании объекта зависимости встраиваются внутрь этого объекта. Чтобы не зависеть от конкретных объектов, в качестве принимаемых параметров указываем интерфейс. Обычно внедрение происходит во время выполнения.