Паттерны и практики написания кода

IoC, DI, DI-Container. Часть 1

Hollywood Principle

ОПРЕДЕЛЕНИЕ

«Не звони нам, мы сами позвоним тебе» → **Инструмент должен организовываться так, чтобы он предупреждал, когда пользователь захочет сделать некоторое действие. Не нужно выстраивать модель поведения по типу «запроси у пользователя команду и выполни её»**

Приложение должно само выполнять команду пользователя, не дожидаясь от него действий. Ожидается, что оно само понимает, что что-то произошло в системе. Получается, приложение обладает разумом, интеллектом.

В разработке Голливудский принцип определяет поток управления программой. Он констатирует, кто управляет потоком: человек, программа, особые подходы.

Inversion of Control (IoC)

статьи Мартина Фаулера

- В классическом стиле написания кода мы полностью управляем ходом выполнения программы и периодически передаём поток библиотекам
- Инверсия – принципиально другой подход работы с потоком управления. Он проявляется, когда в проекте используется фреймворк
- Выполняя запрос к сайту, фреймворк предзагружает библиотеки, находит нужный роутинг и отрабатывает разные правила. В какой-то момент мы попадаем в контроллер. Это та точка расширения, которая пишется разработчиком
- **В этот момент мы не являемся руководителем потока управлением программы, именно фреймворк определяет как работает приложение.** Это и есть инверсия контроля, когда не мы управляем ходом выполнения программы, а кто-то нами руководит – в нашем случае, фреймворк

Инверсия контроля – это очередная метрика того, кто управляет ходом выполнения программы: мы, библиотека или фреймворк

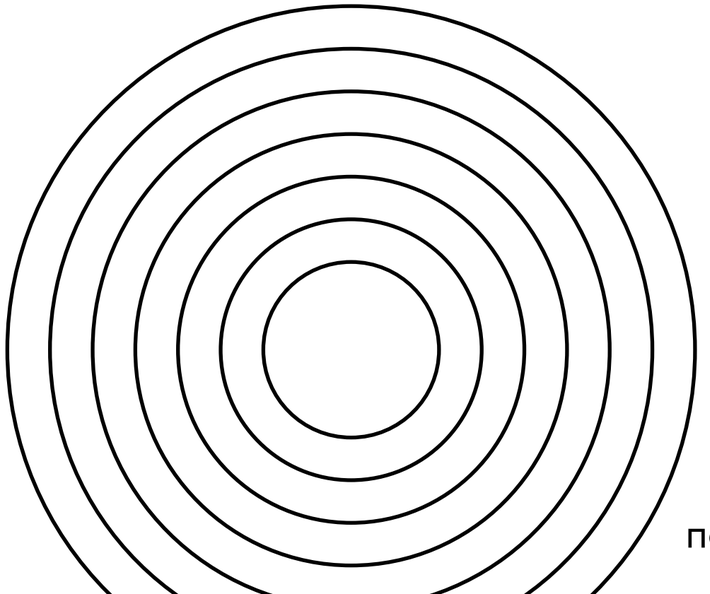
где можно найти IoC

- **паттерны: Фабричный метод, Абстрактная фабрика, Шаблонный метод, Стратегия**
- **в юнит-тестах, когда вызываются setUp() и tearDown() методы**
- **в замыканиях, Dependency Injection и паттерне Service Locator**

книга **«ПРИЁМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ»**

Service Locator

В приложении процесс инстанцирования объектов можно абстрагировать в отдельный слой. Для этого мы вместо создания класса по месту, просим некоторую сущность создать его за нас. Такой сущностью будет Service Locator.



→ Задача Service Locator в том, чтобы хранить и возвращать объект по запросу. **Способы хранения информации о создании класса при этом ничем не регламентированы**

→ Service Locator не имеет канонической реализации. Важна именно идея того, как создаются объекты системы, а не то, как это будет выглядеть в коде. **В таком подходе создаваемый класс начинает определяться локатором, а не внутри реализуемого метода**

→ **Благодаря этому подходу, система становится слабо связанной и частично удовлетворяется Low Coupling.** Это позволяет разделить бизнес-логику и процесс создания классов. Так у нас упрощается сопровождение и расширение кода. При этом тестирование кода в разы улучшается

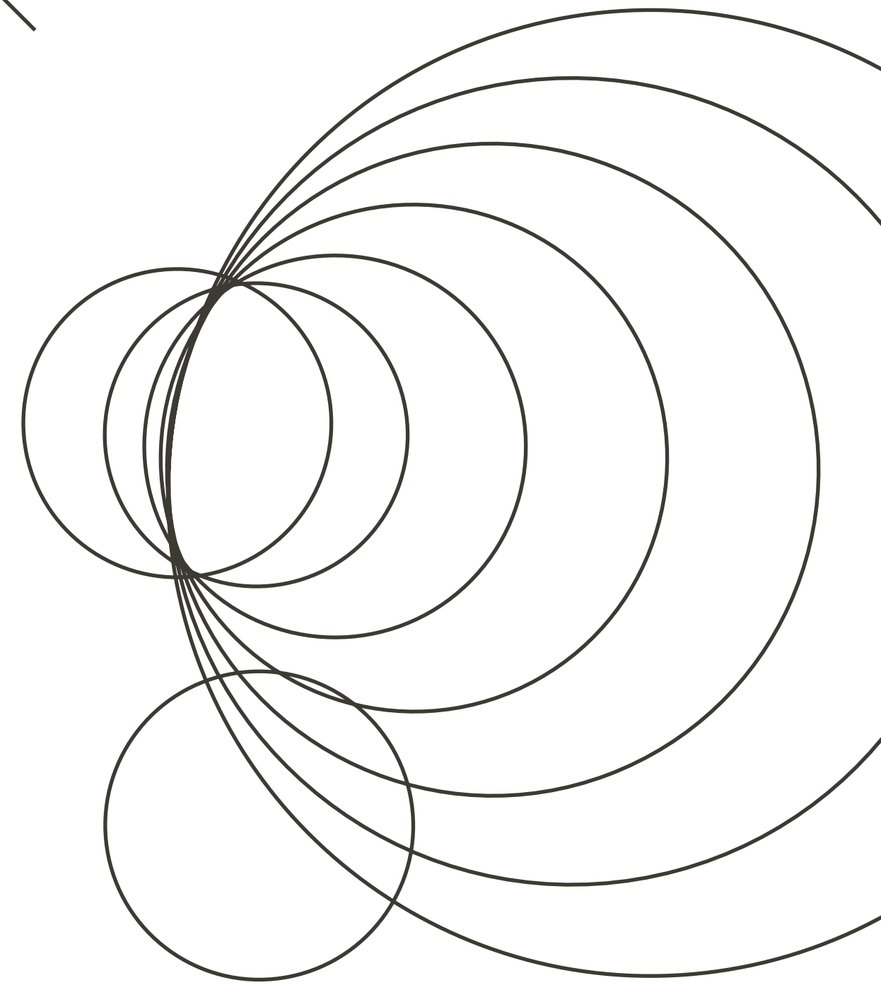
плюс подхода

\+ **Инстанцирование классов отделено от основного кода.** Они входят в свою специальную абстракцию → в тестах проще подменить одну реализацию на другую; не нужно знать, как создавать классы и какие аргументы передавать в конструктор

минус подхода

— **Вместо зависимости от конкретного класса мы во всем приложении начинаем зависеть от Service Locator.** Появляется единая точка отказа → слабая связанность решается только частично

К 2010 году популярность паттерна упала к минимуму, появились более продвинутые практики. Service Locator стал считаться антипаттерном, рекомендуем не использовать его в своих приложениях. К такому решению подталкивает то, что он является аналогом глобального объекта, к которому обращаются из всех частей приложения.




avito.tech