Clean Architecture: стоя на плечах гигантов

Перевод статьи: Clean Architecture: Standing on the shoulders of giants (Herberto Graca)

Robert C. Martin (AKA Uncle Bob) опубликовал свои идеи о Clean Architecture в 2012 году в данной статье и читал лекции об этом на нескольких конференциях.

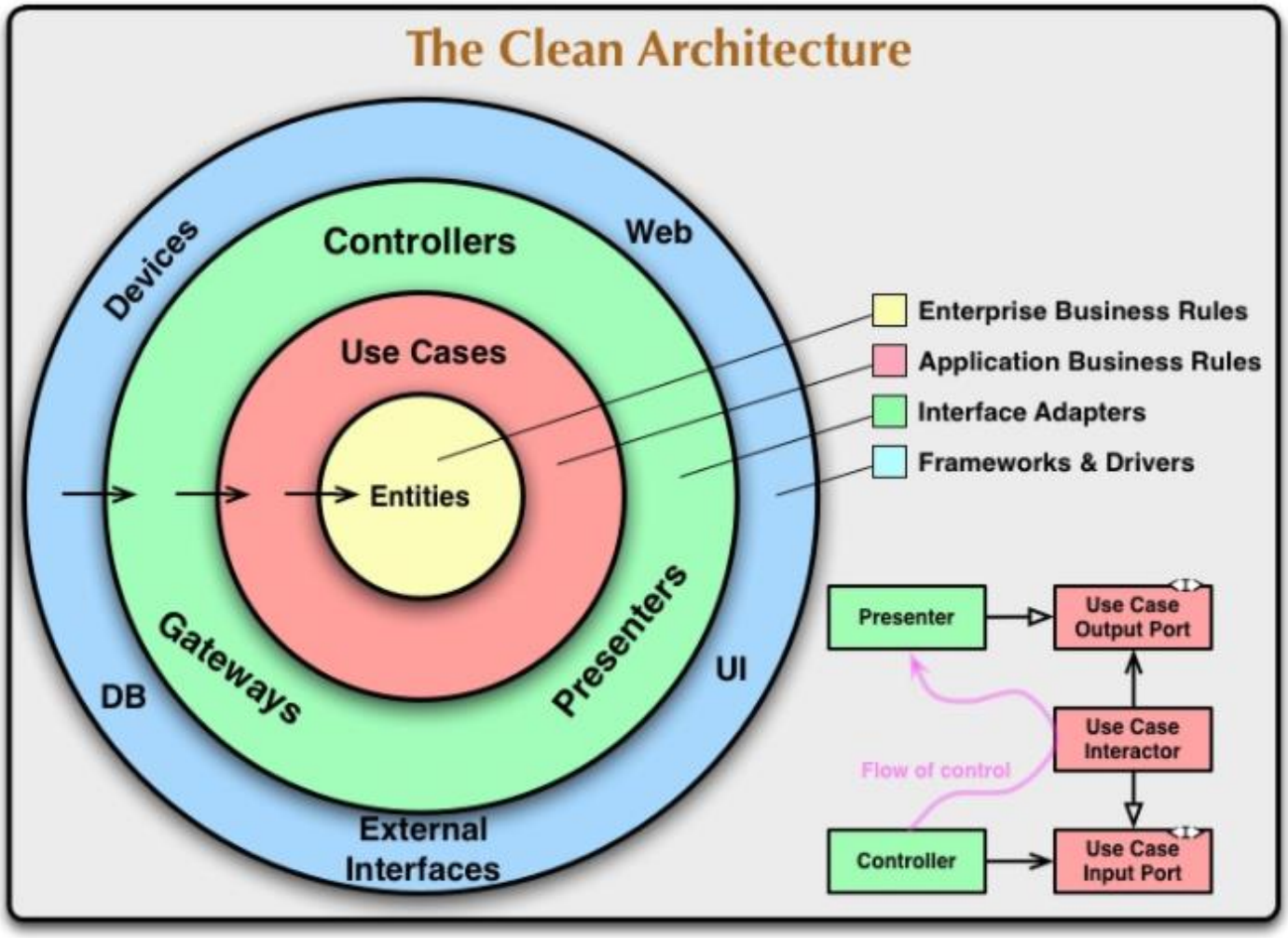
Clean Architecture использует хорошо известные и не очень известные концепции, правила и шаблоны, объясняя, как их объединить, чтобы предложить стандартизированный способ построения приложений.

Стоя на плечах EBI, Hexagonal и Onion архитектур

Основные цели Clean Architecture такие же как и для Ports & Adapters (Hexagonal) и Onion архитектур:

- Независимость от инструментов (framework'ов и т.п.)
- Независимость от механизмов доставки
- Возможность изолированного тестирования

В статье о Clean Architecture была приведена схема, которая использовалась для описания общей идеи:



Как говорит сам Robert C. Martin в своей статье, приведенная выше диаграмма является попыткой интегрировать самые последние идеи архитектуры в единую практическую идею.

Давайте сравним диаграмму Clean Architecture с диаграммами, используемыми для объяснения Hexagonal Architecture и Onion Architecture, и посмотрим, где они совпадают:

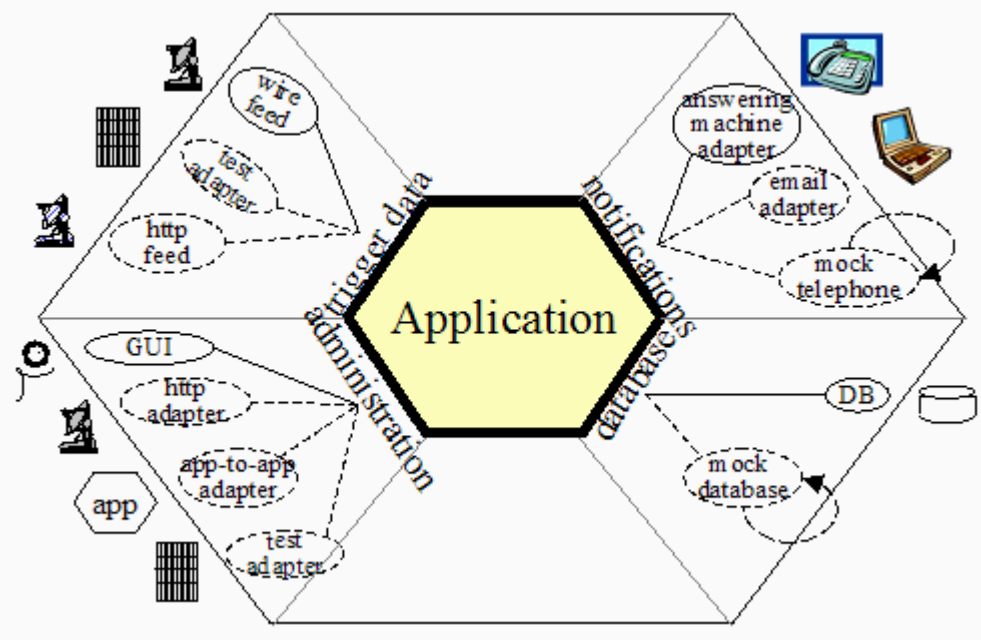


Рисунок - Hexagonal Architecture

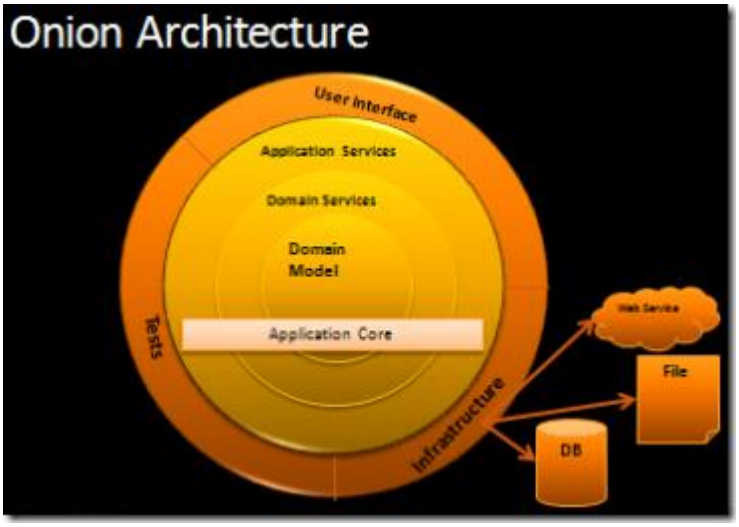


Рисунок - Onion Architecture

Изоляция ядра от инструментов и механизмов доставки

Hexagonal Architecture фокусируется на изоляции ядра приложения от инструментов и механизмов доставки, используя интерфейсы (порты / ports) и адаптеры / adapters. Это также является одной из основ Onion Architecture, как мы видим на схеме UI, инфраструктура, тесты вынесены за границу ядра

приложения. Clean Architecture имеет схожие черты располагая UI, инфраструктуру, тесты и т.д. во внешнем по отношению к ядру приложения слое, что делает ядро приложения независимым от framework'ов и библотек.

Направление зависимостей

В Hexagonal Architecture ничто в явном виде не говорит нам о направлении зависимостей. Тем не менее, мы можем легко сделать вывод: приложение имеет port (интерфейс), который должен быть реализован или использован adapter'ом. Таким образом, adapter зависит от интерфейса, т.е. от ядра приложения, которое находится в центре. Т.е. то, что снаружи, зависит от того, что внутри, следовательно, направление зависимостей - к центру. То же справедливо и для Onion Architecture.

В Clean Architecture, напротив, достаточно отчетливо видно, что зависимости направяются к центру. Отсюда вытекает **принцип инверсии зависимостей (SOLID: DIP)** на архитектурном уровне: *ничто во внутреннем слое (кругу) не может знать о чем-то во внешнем слое*.

Кроме того, *когда мы передаем данные через границу* (boundary – интерфейс, который используется для взаимодействия внешнего и внутреннего слоя), *эти данные всегда инкапсулируются в том виде, который наиболее удобен для внутреннего слоя*.

Слои

Hexagonal Architecture демонстрирует нам только 2 уровня: внутреннюю и внешнюю часть приложения. Onion Architecture, напротив, демонстрирует смесь из слоев приложения определенных в DDD (Domain-Driven Design):

- *Application Services* содержит логику вариантов использования приложения (use case'ы)
- *Domain Services* содержит объекты, которые инкапсулируют логику предметной области (domain logic), которая не принадлежит сущностям (Entities) или объектам-значениям (Value Objects)
- *Domain Model* содержит сущности (Entities) и объектам-значениям (Value Objects), которые отражают понятия и логику предметной области.

В сравнении с Onion Architecture, Clean Architecture поддерживает слои Application Services (Use Cases) и Domain Model (Entities), но в ней не упоминается слой Domain Services. Однако, читая статью Роберта Мартна, мы понимаем, что он рассматривает Entity (сущность) не только как сущность в смысле, в котором это принято в DDD, но и как любой объект домена (предметной области): "Сущность может быть объектом с методами или набором структур данных и функций". В действительности, он объединил слои Domain Services и Domain Model для упрощения диаграммы.

Изолированное тестирование

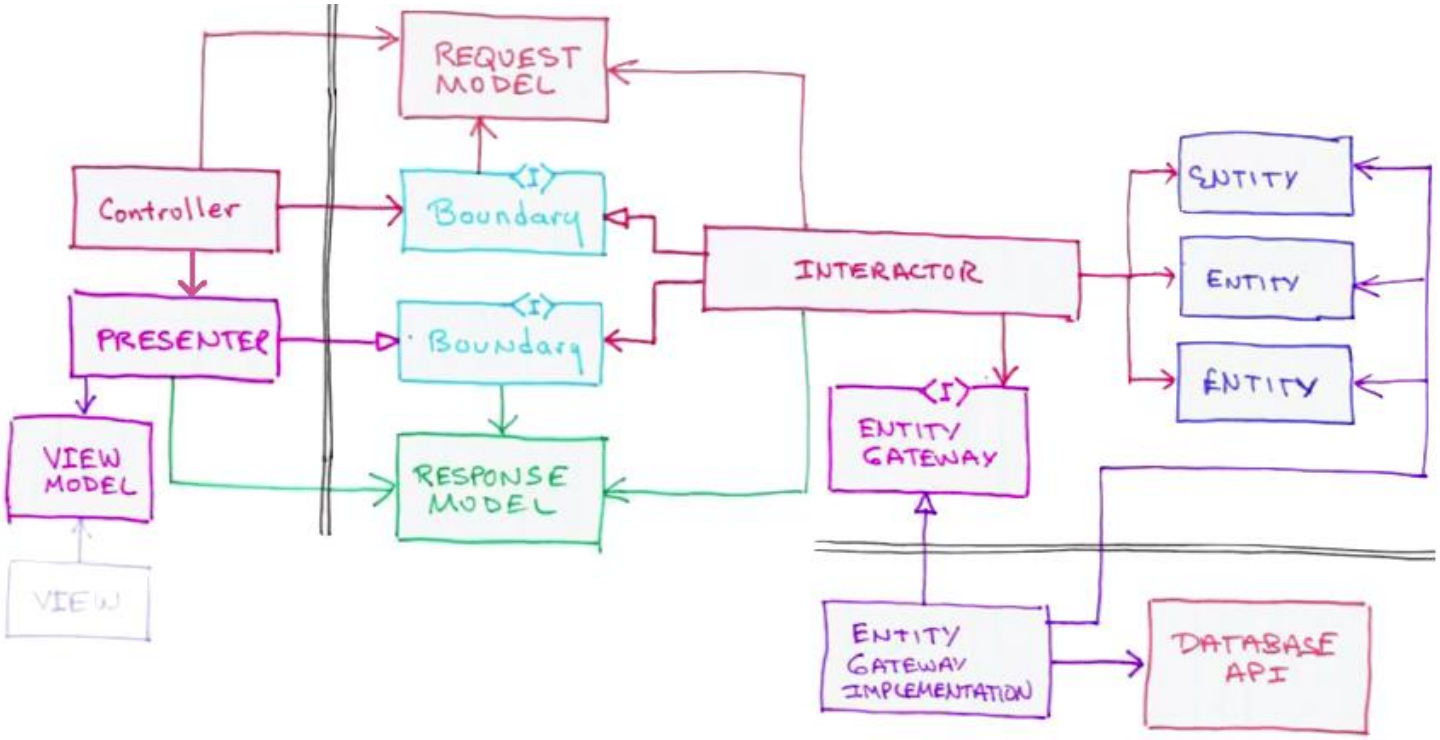
Во всех трех стилях архитектуры правила, которые они соблюдают, обеспечивают изоляцию логики приложения и домена от framework'ов, библиотек и т.п. Это означает, что во всех случаях мы можем легко создать заглушки (mocks, stubs) для внешних инструментов и механизмов доставки, и тестировать код ядра приложения в изоляции, без использования каких-либо запросов к БД, API, web-сервисам и т.п.

Как вы можете видеть, Clean Architecture включает в себя правила, относящиеся к Hexagonal Architecture и Onion Architecture, и не привносит ничего нового. Тем не

менее в нижнем правом углу диаграммы Clean Architecture мы можем видеть одну небольшую дополнительную даграмму...

Стоя на плечах MVC и EBI архитектур

Небольшая дополнительная даграмма в нижнем правом углу диаграммы Clean Architecture поясняет поток управления. Данная диаграмма не дает нам достаточно информации, но этот вопрос расскрывается в статье Роберта Мартина и его лекции на конференции.



На приведенной выше диаграмме мы можем видеть Controller и View из MVC. Все что оганичено черными двойными линиями внутри представляет собой Model из MVC. Такая модель (Model) представляет собой EBI архитектуру (мы можем наблюдать Entities, Boundaries и Interactor), “Application” в Hexagonal Architecture, “Application Core” в Onion Architecture, и уровни “Entities” и “Use Cases” в Clean Architecture.

Поток управления начинается с HTTP-запроса, который достигает Controller'а и далее Controller выполняет следующие действия:

1. Получает запрос и "разбирает" его;
2. Создает REQUEST MODEL на основе данных "разобранного" HTTP-запроса;
3. Вызывает метод INTERACTOR'а (который внедрен в Controller по средствам интерфейса INTERACTOR'а (Boundary или "Use Case Input port" на дополнительной маленькой диаграмме)), передав этому методу в качестве параметров REQUEST MODEL и PRESENTER (который реализует второй интерфейс Boundary или " Use Case Output port" на дополнительной маленькой диаграмме);
4. Далее INTERACTOR выполняет следующие действия:
   4.1 Использует ENTITY GATEWAY IMPLEMENTATION (которая была внедрена в INTERACTOR по средствам интерфейса ENTITY GATEWAY) для поиска и получения необходимых сущностей (Entities);
   4.2 Управляет взаимодействиями между этими сущностями;
   4.3 Создает RESPONSE MODEL на основе данных, которые являются результатом выполнения доменной логики инкапсулированной в сущностях предметной область (Entities);
   4.4 Передает RESPONSE MODEL в PRESENTER;
   4.5 Возвращает PRESENTER Controller'у;
5. Далее Controller использует PRESENTER для создания ViewModel;
6. Привязывает ViewModel к View;
7. Возвращает View клиенту в качестве ответа на HTTP-запрос.

Также поток управления может быть проиллюстрирован следующим примером кода. Допустим у нас есть Controller, метод которого принимает данные формы записи на прием к ветеренару. В соответствии с описанием варианта использования (use case) мы должны:

- сохранить запись на прием и вернуть клиента на страницу клиента, если форма заполнена корректно;
- или вернуть клиента на страницу оформления записи к ветеренару, если форма записи на прием заполнена некорректно.

В данном случае интерфейс Boundary (“ Use Case Output port” на дополнительной маленькой диаграмме), который реализуется PRESENTER’ом может иметь вид:

```
public interface UserInterfacePort {
    void promptForCorrectVisit();
    void presentOwner();
}
```

Тогда PRESENTER будет иметь следующий вид:

```
public class ViewNameUiAdapter implements UserInterfacePort {
    private String viewName;

    @Override
    public void promptForCorrectVisit() {
        this.viewName = "pets/createOrUpdateVisitForm";
    }

    @Override
    public void presentOwner() {
        this.viewName = "redirect:/owners/{ownerId}";
    }

    String getViewName() {
        return viewName;
    }
}
```

INTERACTOR (VisitService) содержит метод .scheduleNewVisit(), который согласно пункту 3, принимает в качестве параметров REQUEST MODEL (Visit) и Boundary (UserInterfacePort, который реализуется PRESENTER’ом (ViewNameUiAdapter))

```
public class VisitService {
    private final VisitRepository visits;

    public VisitService(VisitRepository visits) {
        this.visits = visits;
    }

    public void scheduleNewVisit(Visit visit, BindingResult result, UserInterfacePort userInterfaceP
        if (result.hasErrors()) {
            userInterfacePort.promptForCorrectVisit();
        } else {
            this.visits.save(visit);
            userInterfacePort.presentOwner();
        }
    }
}
```

Метод Controller'a, который использует INTERACTOR и вызывает его метод .scheduleNewVisit(), будет иметь следующий вид:

```
@RequestMapping(value = "/owners/{ownerId}/pets/{petId}/visits/new", method = RequestMethod.POST)
public String processNewVisitForm(@Valid Visit visit, BindingResult result) {
    ViewNameUiAdapter viewNameUiAdapter = new ViewNameUiAdapter();
    visitService.scheduleNewVisit(visit, result, viewNameUiAdapter);
    return viewNameUiAdapter.getViewName();
}
```

Как можно заметить в данном примере не используется Boundary ("Use Case Input port" на дополнительной маленькой диаграмме), а вместо этого Controller использует VisitService напрямую. В данном случае это сделано для упрощения, т.к. мы не собираемся подменять VisitService какой-то другой реализацией.

Единственная вещь во всем этом, по поводу которой я немного сомневаюсь – это использование PRESENTER'a. Я предпочитаю, чтобы INTERACTOR возвращал данные в каком-то DTO, а не вводил объект PRESENTER, который заполняется данными.

Заключение

Я бы не назвал Clean Architecture революционной, т.к. она не привносит какую-то новаторскую идею или шаблон. Однако, я бы сказал, что это концепция первостепенной важности, потому что она:

- отражает каким-то образом забытые концепции, правила, понятия и шаблоны;
- поясняет эти концепции, правила, понятия и шаблоны;
- показыает нам как все эти концепции, правила, понятия и шаблоны объеденяются для того, чтобы предоставить нам способ создания сложных и в тоже время легко поддерживаемых и легко расширяемых программых продуктов.