Базовые принципы построения дизайна информационных моделей

Введение

Актуальность

Востребованность принципов и шаблонов построения дизайна информационных моделей для приложений возрастает пропорционально развитию приложений, которые с каждым днем используют все больше людей и делают это все дольше, из - за чего приложениям ставятся все больше требований и тем самым увеличивая сложность и так сложных дизайнов современных приложений приближая срок их смерти и увеличивая ресурсы на их поддержку. Эти проблемы делают навыки построения дизайнов на локальном и глобальном уровне приложений для удешевления разработки в долгосрочной перспективе все востребование и, в некоторых анемичных или просто сложных системах, обязательным.

**Объект исследования:** принципы и шаблоны построения дизайна информационных моделей приложений.

**Предмет исследования:** продления жизни приложений.

**Цель проекта:** определить принципы и шаблоны моделей приложений.

**Задачи проекта:**

* изучить принципы и шаблоны построения дизайна моделей приложений;
* соединить изученные принципы и шаблоны в единую модель;
* изложить единую модель в основной части работы;

Методы исследования:

* поисковый (сбор и обработка информации по литературным источникам);
* аналитический (анализ и сопоставление информации);
* использование информационных технологий.

**Форма презентации:** доклад.

**Итоговый продукт:** презентация

Архитектура ПО – планы устройства приложения на разных его уровнях абстракции для удовлетворения множества не функциональных требований учитывая будущие изменения, развитие бизнеса и множество других факторов влияющих на конечный продукт – приложение. Архитектура, неформально, делится на ту, которая проектирует весь дизайн системы, закладывает общий план взаимодействий модулей или просто агрегатов зависимостей, для обобщения системы, базового распределения обязанностей в ней и проведения архитектурных границ системы, позволяя разрабатывать её с более линеризированной сложностью, что, в долгосрочной перспективе, позволяет уменьшить затраты на обслуживание и расширения системы в десятки, а то и в сотни раз, и ту, что распределяет сложность в агрегатах сложностей созданных при введении предыдущего вида архитектуры. Такая архитектура называется тактической, так как её влияние, по большей части, распространяется в меньший временной промежуток и на меньший кусок кода, чем предыдущий вид архитектуры, но именно этот вид архитектуры доступен всем грейдам разработчикам, так она создаётся непосредственно в коде, в отличие от предыущей архитектуры, создающеюся на первом этапе старта приложения - стратегической архитектуры, которую обычно имеют в виду, когда говорят об архитектуре в общем. Хоть присутствует явное разграничение этих двух видов архитектур, оно не является строгим и скорей представляет спектр от менее глобальных решений к более. Влияние архитектуры и архитектурных решений в частности, на приложение, является абсолютным и после их введения цена их отмены или изменения является очень большой, в зависимости от глобальности решения.

Тактическая архитектура. Первым и единственным инструментом на этом уровне проектирования являются контракты, позволяющие размазать зависимости, сегрегировать программные сущности и переиспользовать код. Контракт имеет две формы, одновременно существующие при существовании этого контракта – ковариантную и контравариантную. Ковариантность контракта выражается в возможности расширении действия контракта. Контрвариантность, обратно, в отсутствии такой возможности. Справедливым является утверждение, что ковариантность – не строгая часть контракта, контравариантность – строгая. Эти формы существуют зеркально для разных сторон контракта. Так, если у одной стороны он ковариантный, у другой контрвариантный. Логично, один участник контракта исполняет точную его формулировку, второй участник, отталкиваясь от этой точной формулировки, может разными способами её удовлетворить. Таким образом контракты являются единственным способом создания инкапсуляции и в последствии выражения её в виде полиморфизма. Полиморфизм – код, одникого моделирующий множество настоящих морфизмов. Также существует такое понятие как инвариантность, которое, с точки зрения контрактов, можно представить в виде контрвариантного контракта. Контрвариантного потому что его ковариантная сторона выражена не в коде а при проектировании, тем самым справедливо утверждение, что инвариант – контрвариантный контракт в ваакуме. Разные парадигмы используют разные средства создания контрактов. Так в ООП (Объектно-ориентированное программирование) основным средством создания контрактов является наследование, а именно контрактное наследование, при котором описывается часть логики сущности с контрактами на то или иное поведение. В ООП присутствуют другие средства создания контрактов (инкапсуляции в следствии), в виде сокрытия, которая усиливает ковариантную сторону контракта, с точки зрения самого класса, и, вслетствии, контрвариантную, у пользователя этого класса, тем самым позволяя параметрически обрабатывать несколько состояний одним и тем же алгоритмом, тем самым, с точки зрения типов как множеств, создавать полиморфизм. В ФП (Функциональное программирование) все атомарно и средством создания контрактов являются сами функции. Клюевой разностью ООП и ФП, не в пользу ООП, является именно эта накладная сложность создания контрактов у ООП и его громоздкость, не дающая переиспользовать код в том размере, в котором даёт ФП. Так Soft Code (антипаттерн тактического проектирования, характеризующийся излишней маштабируемостью и не грамотной тратой ресурсов, необоснованно создающей сложность), а именно его абсолютная форма, при бесконечной расширяемостью из любой точки приложения, можно представить в ФП с очень большим оверхедом (накладной тратой ресурсов), но в случае ООП сложность улетает в бесконечность уже на том участке, в котором ФП просто сложно писать код. Так и получается код написаный на ФП может быть от 4 до 10 раз меньше, чем объектно-ориентированный. Все из-за накладной цены создания контрактов в ООП, которой нет в ФП.

Принципы тактической архитектуры. Можно выделить три фактически не принадлежащие к этому уровню проектирования, но так как они, по большей части, используются именно здесь, то и располагаться должны именно здесь. Первый – KISS (Keep it simple, stupid), сообщает об негативных чертах Soft Code’а и о том, что не следует создавать излишнюю сложность на пустом месте. Второй - YAGNI (You aren’nt gonna need it), соблюдается при разработке исключительно тех зон, которые нужны прямо сейчас. Третий – DRY (Don’t repeat yourself), соблюдающийся при отсутствии дублирования как кода, так и осей изменений в целом. Эти принципы хоть являются очень простыми, но они часто нарушаются в более неявной форме. Можно выделить отдельные принципы для ООП (из GRASP) для формирования здоровых зависимостей между/в классах и/или объектах. Также существуют отдельные принципы из функционального программирования, указывающие на её абсолютность и математичность. Существуют SOLID практики, которые хоть первоначально трактовались из ООП Робертом Мартином (Дядей Бобом), но их почти в полной мере можно отобразить на ФП (и очень удивится, что оно больше подходит под характеристики ООП, чем само ООП).

Паттерны тактической архитектуры. Существует множество паттернов, как для ООП, так для ФП, но если в ООП это частности реализации, то в ФП это вещи взятые из теории категорий (метаматематики), которые из-за своей матиматичтности можно совмещать и получать более сложные и мощные суммы ФП паттернов. Атомарность ФП даёт возможность более активно использовать контракты и затрачивать на это меньше ресурсов, тем самым расширяемость ФП, использовав всего контракт одной функции, подобно тому, если бы мы скрестили все 23 GoF паттерна, но из-за специфики ООП такое невозможно и все решение так или иначе добивающееся такой расширяемости являлись функциональными. В ООП мы вынуждены идти на компромиссы расширяемости, в ФП – нет. Существуют GRASP паттерны, которые оперируют исключительно зависимостями и являются исключительно паттернами на этом уровне проектирования, которые можно представить в любой парадигме. Полиморфизм тоже является паттерном. Полиморфизм – морфизм выпущенный из контракта.

Примитивные архитектуры. К примитивным архитектурам причисляют те архитектуры, которые при больших системах можно представить в виде тактических шаблонов. Первой такой архитектурой является Transaction Script, который является не столько архитектурой, сколько архитектурным стилем, характеризующиеся отсутствием каких либо архитектурных решений, процедурным кодом и явно-вырежеными транзакциями в коде. Используется в тех случаях, когда отсутствуют оси изменений. Клиент-серверная архитектура, для создания единого приложения для пользователя из двух других приложений (frontend, backend) в вебе. Существует некоторое количество MV шаблонов для создания архитектур маленьких приложений в тех или иных ситуациях.

Монолитные архитектуры. Монолитные архитектуры – архитектуры приложений использующиеся для построения архитектур средней и выше категорий. Являются эволюцией классических MV архитектур.

Rich Domain Model и Anemic/Simple Domain Model. Формально модели из разных парадигм. Anemic/Simple Domain Model часто ругают в книгах множества легендарных писателей, часто называя её антипатерном, но именно она используется в 99% проектах, вместо Rich Domain Model, которую хвалят, так как она, в отличие от Anemic, ориентированна на ООП, когда Anemic на ФП в виде ООП.

Архитектура распределённых систем. Такие архитектуры характеризуются большим разделением модулей, при котором они не взаимодействуют напрямую друг с другом. Обычно взаимодействие происходить через какой-либо интерфейс или прослойку. К таким архитектурам можно присвоить микросервисы, SOA и микрофронтенд.

Архитектурные требования. Навыки определения архитектурных требований – основа архитектуры, без которой все описанные паттерны и принципы не имеют смысла, так как в этом случае при применении все их специфики будут проигнорированы. Обычно невозможно закрыть вообще все требования, так как все инструменты являются компромиссами между ними и даже если бы возможно было применение всех решений одновременно, все равно большое количество требований противоречат друг другу. Выделяют 5 основных требований:

1. Распределение сложности – требование существующие всегда. Результатом является равномерное распределение КПД команды/команд на те участки кода, через которые проходят большие или более активные оси изменений.

2. Скорость разработки – требование в ситуациях с критичным временным ресурсам, например стартапы.

3. Надёжность – требование при присутствии критичных рисков при сбоях. Медицинское оборудование.

4. Перформанс – требование при больших нагрузках. Требование увеличивается параллельно развитию бизнеса.

5. Масштабируемость людских ресурсов – требование при больших проектах. Требование увеличивается параллельно развитию бизнеса.

Существует ещё множество разных требований, но часто они входят в вышеперечисленные.

Вывод. Архитектурные паттерны и принципы могут сильно увеличить долгосрочность системы, но использовать их нужно только в тех случаях, для которых они созданы и не создавать лишнюю сложность на пустом месте. Архитектура создаётся из явных требований, распознание которых является той разницей, которую составляют многие десятки лет опыта коммерческой разработки, но понимание всех сказанных шаблонов обязательно для активной разработки современных приложений.

Список использованных источников

1. Э. Гамма, Р. Хелм, Р. Джонсон и др. Паттерны объектно-ориентированного проектирования: монография – СПб.: Издательский дом ""Питер"", 2020. – 448.
2. Роберт Д. Мартин Чистая архитектура. Искусство разработки программного обеспечения: монография – СПб.: Издательский дом ""Питер"", 2018. – 352.
3. Эрик Эванс Domain-Driven Design: Tackling Complexity in the Heart of Software: монография – Вашингтон: Addison-Wesley Professional, 2003. – 560.
4. Вон Вернон Implementing Domain-Driven Design: монография – Вашингтон: Pearson Education, 2013. – 656.
5. Бек Кент Экстремальное программирование: разработка через тестирование: монография – СПб.: Издательский дом ""Питер"", 2000. – 224.