УДК 82-5+80 (042.5) https://doi.org/10.26160/2541-8637-2026-17-251-256

СРАВНЕНИЕ TRANSACTION SCRIPT, ACTIVE RECORD И DOMAIN MODEL

Андреев Р.А.
Хакасский государственный университет им. Н.Ф. Катанова, Абакан

Ключевые слова: объектно-реляционное отображение, транзакционный сценарий, активная запись, доменная модель, преобразователь данных, бизнес-логика, доступ к данным, управление транзакциями.

Аннотация. В статье рассматриваются три подхода к организации бизнес-логики и доступа к данным в прикладных информационных системах: Transaction Script, Active Record и Domain Model. На основе типовых примеров показано, где располагается логика, как осуществляется работа с транзакциями и объектами, а также какие риски возникают при росте сложности предметной области. Особое внимание уделено шаблонам ORM и проблеме N+1. Сформулированы практические критерии выбора подхода в зависимости от типа домена и требований к тестируемости, поддерживаемости и скорости разработки.

COMPARISON OF TRANSACTION SCRIPT, ACTIVE RECORD, AND DOMAIN MODEL

Andreev R.A.
Katanov Khakass State University, Abakan

Keywords: object-relational mapping, transaction script, active record, domain model, data mapper, business logic, data access, transaction management.

Abstract. The paper compares three approaches for structuring business logic and data access in software systems: Transaction Script, Active Record, and Domain Model. Using typical examples, it explains where business rules are placed, how transactions and object changes are coordinated, and what issues appear as domain complexity grows. The discussion highlights ORM patterns such as Identity Map, Data Mapper, and Unit of Work, and addresses the N+1 query problem. Practical selection criteria are proposed depending on domain type and requirements for testability, maintainability, and development speed.

При проектировании прикладных систем значительная часть архитектурных решений связана с тем, где «живёт» бизнес-логика и как она взаимодействует с базой данных. На практике чаще всего встречаются три подхода к организации бизнес-логики и доступа к данным: Transaction Script, Active Record и Domain Model. Их можно рассматривать как разные ответы на одни и те же вопросы: как разделить ответственность между моделями и сервисами, как организовать транзакции, как обеспечить читаемость, тестируемость и поддержку кода при росте требований.

Дополнительный контекст задают паттерны ORM (Object-Relational Mapping), которые определяют способ сопоставления строк базы данных и объектов языка программирования: Identity Map, Data Mapper и Unit of Work. Identity Map хранит ссылки на уже выгруженные сущности и позволяет переиспользовать один и тот же объект в рамках контекста [1]. Data Mapper задаёт правила преобразования результата запроса в объект, отделяя логику преобразования от доменных объектов [2]. Unit of Work координирует изменения

объектов в рамках транзакции и, в упрощённой интерпретации, является абстракцией над подключением к БД, Identity Map и Data Mapper, обеспечивая единый цикл «загрузить–изменить–зафиксировать» [2].

Ниже представлено сравнение трёх подходов с акцентом на их сильные и слабые стороны, влияние на тестирование и поддержку, а также на типичные проблемы.

Transaction Script: логика в сценариях. Подход Transaction Script предполагает, что модели используются преимущественно как контейнеры данных, а вся бизнес-логика выносится в отдельные функции, сервисы или контроллеры. Сценарий, как правило, оборачивается транзакцией и последовательно выполняет шаги use-case: чтение данных, проверка условий, создание/изменение записей, сохранение.

Типовой пример транзакционного сценария (листинг 1) демонстрирует создание и удаление постов пользователя в рамках `transaction.atomic()`, где получение пользователя и сохранение поста выполняются процедурно.

Листинг 1 – Пример Transaction Script (создание/удаление поста)

```
def add_post(user_id: int, title: str, content: str) -> UserPost:
   with transaction.atomic():
       user = User.objects.get(pk=user_id)
       post = UserPost(user=user, title=title, content=content)
       post.save()
   return post

def delete_post(post_id: int, *, user_id: int | None = None):
   with transaction.atomic():
       qs = UserPost.objects.filter(pk=post_id)
       if user_id is not None:
           qs = qs.filter(user_id=user_id)
       deleted, _ = qs.delete()
       if not deleted:
           raise ValueError(f"Post {post_id} not found")
```

Преимущества Transaction Script:

Во-первых, высокая скорость разработки: сценарии просто писать, они хорошо ложатся на CRUD-подход и понятны для небольших задач. Во-вторых, он действительно удобен для вспомогательных use-case’ов, где логика относительно стабильна и не усложняется со временем.

Недостатки Transaction Script:

Основная проблема – ослабление объектно-ориентированного подхода: логика «размазывается» по сервисам, а модели остаются «пустыми», что снижает инкапсуляцию. Вследствие этого усложняются тестирование и поддержка: чтобы проверить правила, часто приходится поднимать инфраструктурные зависимости, а сами сценарии разрастаются. Отдельно отмечаются проблемы читаемости и вопрос масштабирования: что делать, когда сценарии становятся слишком большими и начинают пересекаться по ответственности. Также может проявляться проблема N+1: при неосторожной загрузке связанных данных сценарии провоцируют множественные запросы к БД при обходе коллекций.

Active Record: логика внутри модели и доступ к БД. Active Record предполагает, что модель одновременно:

1) хранит данные;

2) содержит бизнес-логику;

3) отвечает за доступ к базе данных.

То есть поведение перемещается ближе к данным: операции, связанные с сущностью, оформляются как методы самой сущности.

Листинг 2 – Пример Active Record

```
class User(models.Model):
   username = models.CharField(max_length=100)
   email = models.EmailField(unique=True)
   created_at = models.DateTimeField(auto_now_add=True)

   def add_post(self, title: str, content: str) -> "UserPost":
       return UserPost.objects.create(
           user=self,
           title=title,
           content=content
       )

   def delete_post(self, post_id: int) -> None:
       deleted, _ = self.posts.filter(pk=post_id).delete()
       if not deleted:
           raise ValueError(f"Post not found for user")
```

Преимущества Active Record:

Скорость разработки сопоставима с Transaction Script, но при этом повышается инкапсуляция: операции «создать пост пользователю» или «удалить пост» принадлежат пользователю как объекту. Подход воспринимается более «ООП-образным»: методы сущности концентрируют поведение и часто повышают удобство использования модели.

Недостатки Active Record:

Ключевой минус – смешение ответственностей и нарушение SRP (Single Responsibility Principle): модель одновременно становится и доменным объектом, и слоем доступа к данным. Это сказывается на тестировании: методы модели нередко завязаны на ORM и базу данных, что затрудняет быстрые модульные тесты. Читаемость может быть «плюс-минус» приемлемой на ранних этапах, но при росте логики возникает тот же вопрос: что делать, когда методов и правил становится слишком много. Кроме того, проблема N+1 также актуальна: модельные методы могут «незаметно» инициировать дополнительные запросы при доступе к связанным объектам, если границы загрузки не контролируются явно [3].

Domain Model: доменная логика отдельно от ORM-моделей. Подход Domain Model отделяет доменные объекты от ORM-моделей. Иначе говоря, ORM-модели становятся техническим отражением схемы данных, а доменные модели – носителем правил предметной области.

Листинг 3 – Пример разделения ORM-модели и доменной модели

```
class User(models.Model):
   username = models.CharField(max_length=100)
   email = models.EmailField(unique=True)
   created_at = models.DateTimeField(auto_now_add=True)

@dataclasses.dataclass
class DomainUserPost:
   title: str
   content: str

@dataclasses.dataclass
class DomainUser:
   id: int
   username: str
   email: str
   created_at: datetime.datetime
   posts:                    list[DomainUserPost]                    =
dataclasses.field(default_factory=list)

   def add_post(self, title: str, content: str):
       self.posts.append(DomainUserPost(title=title,
content=content))
```

Преимущества Domain Model:

Во-первых, сильная инкапсуляция: доменные правила сосредоточены в доменных объектах, которые можно проектировать под смысл, а не под ограничения ORM. Во-вторых, высокая тестируемость: бизнес-логику легко покрывать unit-тестами без базы данных, в том числе проверять edge-case'ы. В-третьих, улучшается читаемость и поддержка, поскольку предметная область выражена через намерения и инварианты, а не через набор запросов. Наконец, при грамотной настройке границ загрузки агрегатов проблема N+1 может быть устранена на уровне архитектуры: объекты загружаются в заранее определённом объёме и составе связей.

Недостатки Domain Model:

Главный компромисс – скорость разработки и порог входа. Требуется больше проектирования и дисциплины, а также инфраструктура для отображения доменных объектов на базу. Отмечается отсутствие «нативной поддержки» доменной модели в некоторых фреймворках, что увеличивает стоимость внедрения.

Unit of Work и практика на примере SQLAlchemy. Для реализации Domain Model часто используют ORM с выраженными паттернами Unit of Work и Data Mapper. В материалах показано, что SQLAlchemy реализует Unit of Work и предоставляет явный интерфейс жизненного цикла изменений объектов: `add`, `flush`, `refresh`, `delete`, `commit` [4]. Важно, что изменения не обязательно фиксируются «in place» немедленно: они могут ожидать явной команды со стороны клиентского кода.

Пример взаимодействия с сессией (листинг 4) иллюстрирует управляемую фиксацию изменений.

Листинг 4 – Операции Unit of Work в SQLAlchemy

```
session.add(post)
await session.flush()
await session.refresh(post)
await session.delete(post)
await session.commit()
```

Декларативное описание сущностей распространено и хорошо документировано: сущности описываются классами, а связи – через `relationship`, где можно управлять каскадами и стратегиями загрузки [3].

Императивный стиль обычно сложнее в понимании и реализации, но он предоставляет гибкий контроль над маппингом: отдельно объявляются таблицы, отдельно – «чистые» Python-классы, затем настраивается соответствие между ними. Такой подход позволяет моделировать сложные случаи, включая композиции и тонкую настройку свойств [3].

Отдельно подчёркнуто, что при настройке мапперов границы выгружаемых объектов задаются статично: один раз определяется, в каком состоянии и с какими связями будет загружаться объект. Это делает невозможным возникновение N+1 в рамках выбранных границ, но требует более глубокого понимания домена для определения границ агрегата; задача является инженерно сложной и не имеет универсальных шаблонных решений.

Также показано, что в зависимости от `select` результатом запроса могут быть либо сырые строки, либо Python-объекты; при этом благодаря Identity Map изменения объектов отслеживаются и при `commit` сохраняются в базе [3].

Сравнение трёх подходов можно свести к нескольким критериям: скорость разработки, инкапсуляция, тестируемость, поддерживаемость и устойчивость к росту сложности.

Transaction Script целесообразен там, где требуется быстро реализовать ограниченный набор сценариев, логика невелика и редко меняется. Он особенно удобен для вспомогательных use-case’ов и «generic» доменов, но при росте сложности ведёт к разрастанию сценариев, ухудшению читаемости, снижению инкапсуляции и усложнению тестирования.

Active Record – естественный выбор для приложений, где ORM-модель является центральной единицей разработки, а логика умеренна. Он повышает локальность поведения, но при росте домена превращает модели в «божественные объекты», смешивая слой хранения и слой правил, что усложняет эволюцию и тестирование. В обоих подходах Transaction Script и Active Record риск N+1 остаётся существенным и требует отдельной дисциплины загрузки данных.

Domain Model лучше всего подходит для сложных частей приложения, где бизнес-правила богаты, меняются, содержат исключения и требуют широкой проверки. За счёт отделения доменной логики от инфраструктуры упрощаются модульные тесты и повышается качество кода. Цена – более высокий порог входа, большая проектная работа и необходимость инструментов/практик, а также аккуратное определение границ агрегатов, чтобы контролировать загрузку данных и исключать N+1.

Транзакционный сценарий (Transaction Script) и активная запись (Active Record) оправданы в областях, где логика меняется редко и её немного, особенно

в support и generic доменах. При усложнении предметной области эти подходы начинают проигрывать из-за слабой масштабируемости логики, трудностей тестирования и риска ухудшения читаемости. Доменная модель (Domain Model) требует большего первоначального усилия и квалификации, но существенно упрощает разработку и изменение сложной бизнес-логики, повышает тестируемость и поддержку кода. В сочетании с паттернами ORM и продуманными границами агрегатов доменная модель обеспечивает устойчивую архитектуру для core-доменов и снижает риск N+1.

Список литературы

1. Fowler M. Patterns of Enterprise Application Architecture. – Addison-Wesley, 2002.
2. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. – Addison-Wesley, 2003.
3. SQL Alchemy Documentation: ORM, Unit of Work, relationship loading strategies, imperative mapping [Электронный ресурс]. – Режим доступа: https://docs.sqlalchemy.org/.

Сведения об авторе:
Андреев Роман Андреевич – аспирант.