УДК 004.75

Д.А. Сенченко
Вологодский государственный университет

АНАЛИЗ ПАТТЕРНОВ МЕХАНИЗМА ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ В ВИЗУАЛЬНОЙ РАЗРАБОТКЕ

В данной статье представлен сравнительный анализ существующих вариантов к решению проблемы управления зависимостями в проектах визуальной разработки для Unity. Анализ демонстрирует преимущества и недостатки различных подходов. В итоге обнаруживается оптимальный подход к решению данной проблемы в проектах визуальной разработки для Unity.

Визуальная разработка, управление зависимостями, внедрение зависимостей, Unity.

В последние годы можно наблюдать серьезный прогресс в области разработки визуального программного обеспечения. Значительную часть данной области занимают игровые проекты, рынок которых с 2015 по 2020 вырос почти на 30 процентов и составил 129 миллиардов долларов, с которых государство может получить налоги. Кроме того, в последние годы сильно набирают направления корпоративного обучения сотрудников посредством технологий дополненной и виртуальной реальности (AR и VR). Показательной является статистика, которую приводит компания «Северсталь» в своей статье: «В России по данным прошлогоднего исследования, охватившего 100 крупнейших отечественных предприятий, AR/VR-технологии использует 21 % компаний. 40 % из них – в IT-секторе, 33 % – в металлургии, по 25 % – в телекоммуникационной и нефтегазовой сферах». Благодаря технологиям дополненной и виртуальной реальности можно избежать производственных инцидентов и увеличить качество выполняемой работы [1]. С ростом спроса на проекты визуализации стали расти требования заказчика к программному обеспечению, а также стала проявляться специфика их применения. Однако, как и любое программное обеспечение, проекты визуализации должны соответствовать ряду критерий качества, которые обозначены в модели качества программного продукта Боэма:

1. Мобильность.
2. Надежность.
3. Эффективность.
4. Эргономичность проектирования.
5. Тестируемость.
6. Понятность.
7. Изменяемость.

Программное обеспечение, соответствующее данным критериям, будет считаться качественным.

Однако в области визуальной разработки, а особенно в сфере разработки игровых проектов, наблюдается следующая проблема: масштабные проекты содержат большое количество ошибок, кодовая база имеет большой размер и непонятна для разработчиков, а внесение изменений в проект обходится большими денежными и финансовыми потерями для заказчика программного продукта. Данная проблема вызвана тем, что количество зависимостей в проекте выросло, а механизм управления зависимостей работает некорректно.

Стоит отметить, что подобная проблема встречается не только в проектах визуализации, но и в других видах программного обеспечения, таких как банковское ПО, сайт интернет-магазина и системы управления персонала. Разработчики данных проектов для решения проблемы изобрели паттерны управления зависимостями. Однако специфика визуальной разработки вносит определенные трудности при использовании данных паттернов. Научных трудов, которые решают данную проблему с такой спецификой, написано мало, и они не покрывают все аспекты.

При разработке масштабных проектов программного обеспечения архитекторы сталкиваются с новыми трудностями, которые не встречаются при разработке небольших проектов. Трудности эти связаны с увеличением размера кодовой базы и с появлением большего количества зависимостей между компонентами проекта. Игнорирование данных трудностей проектирования приводит к ряду проблем. Особенно это актуально для визуальной разработки, так как, кроме наличия проблем, связанных с отображением интерфейса пользователя (что обыкновенно, если речь идет о веб-приложениях), появляются дополнительные трудности с взаимодействием игровых объектов на сцене.

Во-первых, становится сложно вносить изменения в существующую кодовую базу, так как она имеет большие размеры и сильную связность файлов или

классов. Во-вторых, при появлении разного рода ошибок в работе программного обеспечения, поиск неисправности в кодовой базе усложняется. В-третьих, увеличивается продолжительность обзора кода другими разработчиками с целью модерирования изменения кодовой базы проекта. В-четвертых, что особенно актуально для программного обеспечения визуализации, это ограничения со стороны платформы, которые обостряются при масштабировании программного обеспечения.

Рассматривая данную проблему в контексте среды платформы Unity, можно привести в качестве примера понятие игровых объектов, которые находятся на сцене. Функционирование проектов визуализации построено полностью на них и доставка к ним нужных зависимостей является актуальной задачей для разработчиков.

Цель данной научной статьи – разобрать и проанализировать актуальные паттерны механизма внедрения зависимостей в программное обеспечение визуальной разработки. В процессе анализа необходимо будет установить достоинства и недостатки паттернов, а также их специфику для визуальной разработки. После выполнения этих задач можно будет получить результат научной работы – найти оптимальной паттерн для визуальной разработки на платформе Unity.

Для того чтобы понять, как устроен данный механизм, нужно увидеть проблематику управления зависимостями в больших проектах. Поэтому необходимо разобраться с базовыми определениями.

Зависимость – это однонаправленная связь между сущностями, которая означает, что некоторый экземпляр класса вызывает свойства или функции экземпляра другого класса. Это понятие может быть во всех парадигмах программирования.

Зависимости неизбежны в любых программных проектах, где количество файлов или классов больше одного. Роберт Мартин в своей монографии «Чистая архитектура» [2] вводит понятие метрики неустойчивости класса. Ниже представлена формула, по которой вычисляется данная метрика:

$$I = \frac{C_e}{C_a + C_e},$$

где Ce – количество исходящих связей (efferent coupling), а Ca – количество входящих связей (afferent coupling); класс абсолютно стабилен при Ce, равном 0.

Появление зависимостей в проекте вызывает ряд проблем:

- изменение в устойчивых классах, то есть когда у класса метрика неустойчивости будет стремится к 0, могут появляться ошибки во всех исходящих связях;
- когда проект разрастается до 100+ классов, прокинуть необходимую зависимость до получателя через всю архитектуру будет достаточно затруднительно;
- также существуют опасные ситуации, когда устойчивые классы начинают зависеть от неустойчивых классов, что может вести к скрытым ошибкам исходящих связей стабильного класса;
- отсутствие понимания, какие классы должны быть устойчивыми, а какие нет.

Именно для решения данных задач появился термин внедрение зависимостей.

Dependency Injection («Механизм внедрения зависимостей») – это понятие впервые использовал Мартин Фаулер в статье «Inversion of Control Containers and the Dependency Injection Pattern». В этой статье он дает этому механизму следующее определение: «Внедрение зависимостей – это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами. DI – это альтернатива самонастройке объектов, где объектом является экземпляр конкретного класса в парадигме объектно-ориентированного программирования» [2].

Однако в понятие «управление зависимостями» попадает не только внедрение зависимостей, а еще ряд терминов, который также необходимо разобрать.

Среди основных терминов можно выделить инверсию управления – принцип, согласно которому можно отличить библиотеку от фреймворка. Классическая модель подразумевает, что вызывающий код контролирует внешнее окружение, время и порядок вызова библиотечных методов (также известная как pull-модель взаимодействия). Однако в случае фреймворка обязанности меняются местами: фреймворк предоставляет некоторые точки расширения, через которые он вызывает определенные методы пользовательского кода (также известная как push-модель взаимодействия).

Простой метод обратного вызова или любая другая форма паттерна Наблюдатель является примером инверсии. Зная значение понятия IoC, становится ясно, что такое понятие, как IoC-контейнер, лишено смысла, если только данный «контейнер» не предназначен для упрощения создания фреймворков. Для понимания данного механизма ниже приводится специальная схема (рис. 1).

В визуальной разработке на движке Unity данный принцип также используется. Pull-модель используется в большинстве стандартных задач, например перемещение игрового объекта в пространстве (рис. 2).

В качестве примера push-модели можно привести пример наследников класса MonoBehaviour, которые переопределяют методы жизненного цикла. Фреймворк Unity вызывает переопределенные методы этих наследников (рис. 3).

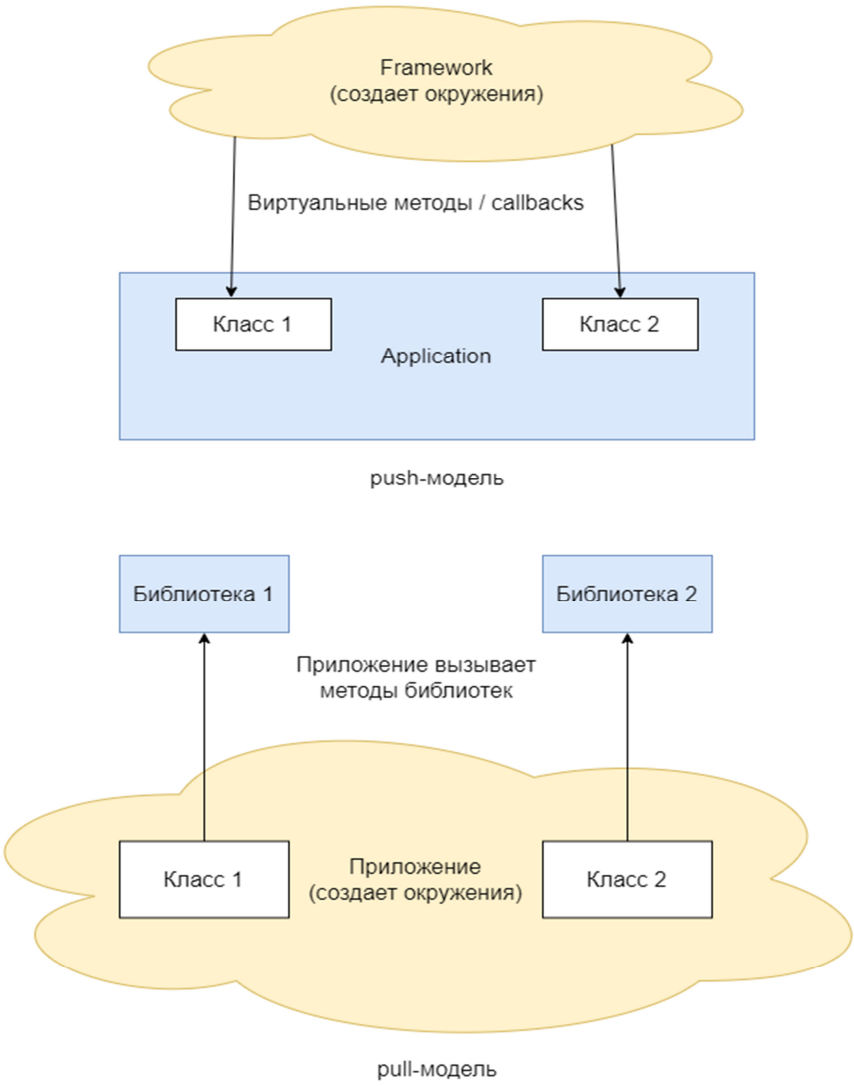

Рис 1. Инверсия управления в работе с библиотеками и фреймворками

```
transform.Translate(0, 0, data.speed * Time.deltaTime);
```

Рис 2. Пример реализации pull-модели

```
Сообщение Unity | Ссылок: 0
void Update()
{
```

Рис 3. Пример реализации push-модели

Следующим важным разделом на пути изучения работы с внедрением зависимостей является анализ основных паттернов при решении данной проблемы.

Первым таким паттерном является достаточно очевидный подход для объектно-ориентированной парадигмы, а именно внедрение зависимостей путем передачи параметров в конструкторе. Данный способ предполагает, что все необходимые зависимости будут переданы экземпляру класса, когда будет вызван конструктор, тем самым реализуя принцип агрегации. Достоинства:

- простота освоения данного паттерна;
- не нужно использовать сторонние библиотеки и фреймворки;
- все зависимости находятся в одном месте, поэтому они становятся явными для разработки, благодаря этому проще становится понять, когда

необходимо выделить отдельную сущность, чтобы уменьшить количество параметров в конструкторе.

К недостаткам можно отнести следующие аспекты:

- громоздкость конструкторов при их вызове;
- если в программном проекте большое количество классов, то использование данного паттерна в чистом виде будет затруднительно.

В случаях, когда используется композиция, появляется закономерный вопрос, а откуда получит класс, поставляющий зависимость, нужный экземпляр класса. Таким образом, проблема становится рекурсивной, и у разработчиков существуют те же пути решения – композиция и агрегация. Но до бесконечности это продолжаться не может. Поэтому существует паттерн Composite Root.

Любое приложение содержит точку входа, с которой начинается его исполнение. В идеальном случае эта точка содержит всего несколько строк кода, которые сводятся к созданию одного (или нескольких) экземпляров самых высокоуровневых классов, представляющих основную логику приложения. Именно здесь нам придется решить, какие абстракции требуются нашим модулям верхнего уровня и именно точка входа приложения является идеальным местом для разрешения всех зависимостей. Данная точка входа и называется Composite Root.

Если проект действительно большой и делится на несколько модулей, то классов типа Compostite Root будет несколько и их порядок будет определятся классом Composite Order.

Основные преимущества паттерна Composite Root:

- нет необходимости подключать сторонние библиотеки и фреймворки;
- используется только внедрение зависимостей через конструктор, что позволяет проще контролировать зависимости в рамках одного класса.

Однако существуют также и недостатки использования данного паттерна:

- каждую зависимость для каждого класса приходится отслеживать в вызовах конструкторов;
- модули Composite Root являются крайне стабильными зависимостями.

Следующим способом внедрения зависимостей является механизм внедрения зависимостей через свойство. Суть его заключается в том, что зависимости передаются через сеттер-свойства. Данный механизм следует применять, когда речь идет о необязательных зависимостях, у которых существует реализация по умолчанию. Стоит также отметить, что DI-контейнеры также поддерживают данный способ внедрения зависимостей.

Преимущества данного механизма:

- когда речь идет о необязательных зависимостях, данный паттерн подходит лучше всего;
- большое количество зависимостей в конструкторе делает код трудночитаемым.

Недостатки:

- в отличие от внедрения зависимостей в конструктор, нам необходимо делать проверку на null значения свойства;
- многие новички передают через свойства обязательные зависимости, что является ошибкой;
- для выставления значения по умолчанию приходится использовать композицию, и если у нас таких зависимостей большое количество, то это сделает код класса громоздким.

В качестве альтернативы можно использовать второй конструктор, в котором будет передана недостающая зависимость.

Следующим паттерном, который применяется для внедрения зависимостей, является передача зависимости через аргументы метода или, другими словами, Method Injection. Суть этого механизма заключается в передаче необходимой для работы метода зависимости через аргументы. Стоит отметить, что эта зависимость используется только при работе метода, и поэтому можно назвать ее динамичной. Получается, что зависимость определяет режим работы метода.

Основные достоинства данного подхода:

- в случае ошибки сужается зона ее поиска до одного метода;
- делает работу метода более гибкой;
- позволяет избавиться от лишних зависимостей в конструкторе класса.

Основные недостатки данного подхода:

- ни один из популярных Dependency Injection контейнеров не поддерживает данный паттерн. А значит, автоматизация внедрения зависимостей невозможна;
- если метод использует зависимость, которую не используют другие члены класса, то появляются определенные конфликты с Single Responsibility Principle – принципом единства ответственности. Возможно, стоит подумать над тем, чтобы выделить еще один класс.

Также необходимо рассмотреть последний подход – DI-контейнер. Суть этого паттерна заключается в использовании сторонних библиотек и фреймворков для автоматизации внедрения зависимостей. Благодаря DI-контейнерам, можно автоматизировать такие паттерны, как внедрение зависимостей в конструктор и внедрение зависимостей в свойства класса. Существует несколько примеров DI-контейнеров, ниже будет исследован один из них, который используется в Android – библиотека Dagger 2.

Основой библиотеки Dagger 2 является два основных элемента:

1. Модуль – здесь хранятся данные о том, каким способом поставить необходимую зависимость получателю.

2. Компонент – интерфейс, в котором хранится информация о том, куда поставляются необходимые зависимости.

Основные преимущества использования DI-контейнера:

- автоматизация процессов, связанных с внедрением зависимостей, что очень полезно для больших программных проектов;
- уменьшается шанс ошибиться при внедрении зависимостей;
- возможность решать нетривиальные задачи по внедрению зависимостей.

Основные недостатки использования DI-контейнера:

- появляется сильная зависимость от сторонней библиотеки, при изменениях в которой могут возникнуть скрытые ошибки;
- отсутствие поддержки паттерна внедрения зависимостей в метод.

В рамках данной статьи были рассмотрены зависимости между классами в программных проектах, а также проанализированы различные паттерны по их внедрению – их сильные и слабые стороны. Для выявления оптимального подхода к внедрению зависимостей были использован сравнительный научный метод, то есть были приведены достоинства и недостатки всех паттернов. Стоит отметить, что данные паттерны будут также актуальны и для визуальной разработки, однако оптимальным является использование DI-контейнера, так как он позволяет автоматизировать управление зависимостями, что решает проблемы, связанные с зависимостями между компонентами программного обеспечения. В качестве подтверждения данного утверждения можно привести факт того, что большинство коммерческих компаний, которые задействованы в визуальной разработке на платформе Unity, используют данный паттерн. А в данной научной работе было обосновано направление, которое выбрали коммерческие компании.

Литература

1. «Северсталь» расширяет применение VR-технологий в обучении персонала» – CNevs. – https://www.cnews.ru/news/line/2022-12-08_severstal_rasshiryaet_primenenie (дата обращения: 31.04.2023). – Текст : электронный.

2. Механизмы внедрения зависимостей. – Мартин Фулер. – https://martinfowler.com/articles/injection.html (дата обращения: 31.04.2023). – Текст : электронный.

2. Мартин, Р. Чистый код / Р. Мартин. – 1 изд. – Санкт-Петербург : Питер, 2010. – 464 с.

3. Мартин, Р. Чистая Архитектура / Р. Мартин. – 1 изд. – Санкт-Петербург : Питер, 2018. – 352 с.

4. Бадд, Т. Объектно-ориентированное программирование в действии / Т. Бадд. – 1 изд. – Санкт-Петербург : Питер, 1997. – 464 с.

D.A. Senchenko
Vologda State University

ANALYSIS OF DEPENDENCY INJECTION MECHANISM PATTERNS IN VISUAL DEVELOPMENT

This article provides an analytical review of the existing patterns for dependency control in visual development for Unity projects. The analysis highlights the advantages and disadvantages of each option. This article finds the optimal approach for Unity projects using for visual development.

Visual development, dependency control, dependency injection Unity.