ИССЛЕДОВАНИЕ МЕТОДОВ ПАТТЕРНОВ В JAVA И ИХ РОЛЬ ПРИ РАЗРАБОТКЕ ПРИЛОЖЕНИЙ

RESEARCH PATTERN METHODS IN JAVA AND THEIR ROLE IN APPLICATION DEVELOPMENT



УДК 004.04

Свищёв А.В., Ассистент кафедры практической и прикладной информатики, МИРЭА-Российский технологический университет (РТУ МИРЭА), г. Москва

Беликов И.В., Студент бакалавриата, 2 курс, МИРЭА-Российский технологический университет (РТУ МИРЭА), г. Москва

Svishchev A.V., Assistant of the Department of Practical and Applied Informatics, MIREA-Russian Technological University (MIREA Russian Technical University), Moscow

Belikov I.V., Bachelor student, 2 course, MIREA-Russian Technological University (MIREA Russian Technical University), Moscow

Аннотация

В данной статье рассматривается практическое применение различных видов паттернов в языке программирования Java, которые могут быть использованы в разработке различного программного обеспечения. Детально анализируется типы паттернов, наибольшее внимание уделено поведенческому паттерну цепочка обязанностей, предлагается усовершенствование, позволяющее систематизировать и расширять возможности данного паттерна при помощи дополнительного java класса.

Annotation

This article is dedicated to the topic of application of patterns in Java programming language, that are considered to be used during the process of software development.

Types of patterns are being analyzed in details, most attention is paid to behavioral pattern chain of responsibility, modernization, that helps to systemize and expand patterns' capabilities by additional Java class, is offered.

Ключевые слова: паттерны, java, цепочка обязанностей, модернизация.

Key words: patterns, java, chain of responsibility, modernization.

Применение паттернов в Java представляет собой вполне сложившееся и весьма востребованное современное течение в обширном спектре информационных технологий. Для того, чтобы приложение было написано правильно, читабельность кода была на высоте, необходимо проводить планирование и оптимизацию приложения, в частности, работать с шаблонами проектирования. Зачастую появляются или требуются уточнения, связанные с этим понятием, что вынуждает заниматься поиском той или иной информации, собственно, этому и посвящена настоящая статья.

Определение паттернов в Java. Паттерны проектирования (шаблоны проектирования) - это готовые к использованию решения часто возникающих в программировании задач. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее. Паттерн проектирования, подходящий под задачу, реализуется в каждом конкретном случае. Такой паттерн, будучи примененным неправильно или к неподходящей задаче, может принести немало проблем. Тем не менее, правильно примененный паттерн поможет решить задачу легко и просто.

Существуют различные типы паттернов, такие как:

- порождающие
- структурные
- поведенческие

Как можно понять из названий, каждый из них представляет собой определённый механизм, который позволяет достичь результата строго структурированными и имеющими конкретную задачу методами. Так порождающие паттерны представляют различные способы инициализации объектов, позволяя создавать конкретные из них удобным способом. Структурные паттерны определяют, зачастую строго, отношения между объектами и классами: это

позволяет им взаимодействовать друг с другом и сводит к минимуму возможность случайных ошибок со стороны программиста. Поведенческие паттерны используются для того, чтобы упростить взаимодействие между сущностями.

Существует огромное количество паттернов. Ниже представлен список некоторых из их с кратким описанием.

Порождающие:

Singleton (Одиночка) - ограничивает создание одного экземпляра класса, обеспечивает доступ к его единственному объекту.

Factory (Фабрика) - используется, когда имеется суперкласс с несколькими подклассами и необходимо вернуть один из подклассов на основе ввода.

Abstract Factory (Абстрактная фабрика) - используем “супер-фабрику” для создания фабрики, затем используем созданную фабрику для создания объектов.

Prototype (Прототип) - помогает создать дублированный объект с лучшей производительностью, вместо нового создается возвращаемый клон существующего объекта.

Builder (Строитель) - используется для создания сложного объекта с использованием простых объектов. Постепенно он создает больший объект из малого и простого объекта.

Структурные:

Adapter (Адаптер) - это конвертер между двумя несовместимыми объектами. При использовании данного адаптера объединение двух несовместимых интерфейсов становится возможным.

Composite (Компоновщик) - использует один класс для представления древовидной структуры.

Bridge (Мост) - делает конкретные классы независимыми от классов реализации интерфейса.

Flyweight (Легковес) - вместо создания большого количества похожих объектов, объекты используются повторно.

Decorator (Декоратор) - добавляет новые функциональные возможности существующего объекта без привязки его структуры.

Поведенческие:

Template Method (Шаблонный метод) - определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

Chain of Responsibility (Цепочка обязанностей) - позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами.

Mediator (Посредник) - предоставляет класс посредника, который обрабатывает все коммуникации между различными классами.

State (Состояние) - объект может изменять свое поведение в зависимости от его состояния.

Observer (Наблюдатель) - позволяет одним обьектам следить и реагировать на события, происходящие в других объектах.

Паттерн цепочка обязанностей. Цепочка Обязанностей (Chain of responsibility) - поведенческий шаблон проектирования, который позволяет избежать жесткой привязки отправителя запроса к получателю. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке. Каждый объект в этой цепочке при получении запроса выбирает: либо закончить обработку запроса, либо передать запрос на обработку следующему по цепочке объекту. Зачастую используется для избегания привязки отправителя запроса к его получателю, что даёт возможность обработать данный запрос несколькими объектами. Область применения данного шаблона достаточно велика, ведь он позволяет сделать так, чтобы объект класса не знал о том, кто будет заниматься обработкой его запроса, это позволяет достичь гибкости при распределении обязанностей между объектами. Ознакомиться с диаграммой классов данного паттерна можно на рис.1.

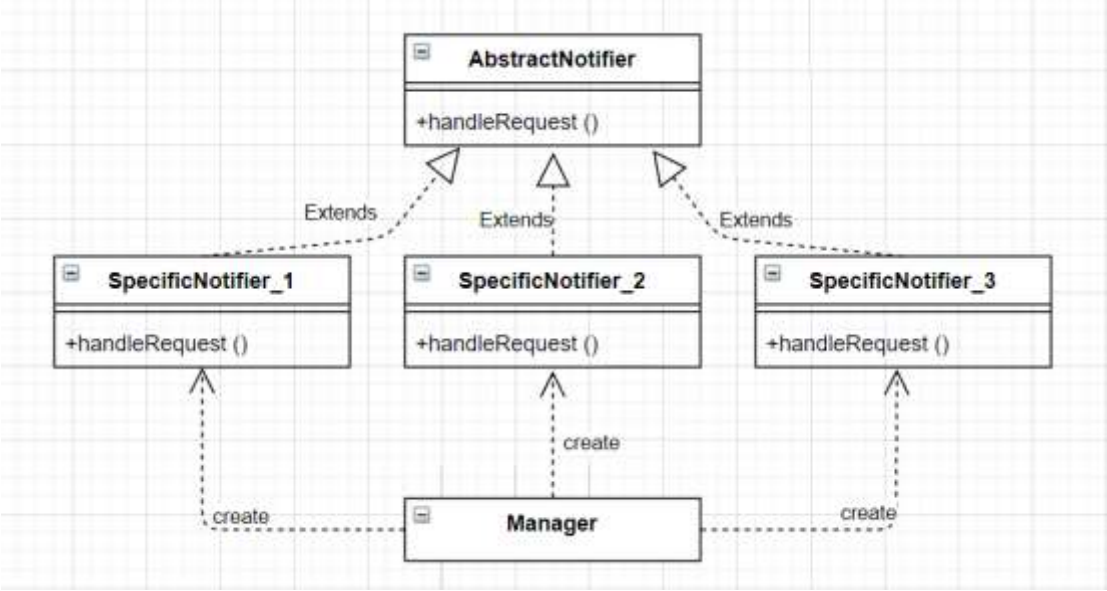

Рисунок 1 - Диаграмма классов паттерна Цепочка Обязанностей (Chain of responsibility)

Как можно заметить на данной диаграмме классов, существует абстрактный класс AbstractNotifler, от которого наследуются классы: SpecificNotifier_1, SpecificNotifier_2, SpecificNotifier_3. Так же присутствует класс Manager, который в свою очередь создаёт внутри себя объекты классов SpecificNotifier_1, SpecificNotifier_2 и SpecificNotifier_3.

Исходя из вышеперечисленного, можно сделать вывод о том, что данная цепочка является отказоустойчивым механизмом с конкретными функциями, реализуемыми ей в полной мере. Однако существует возможность модернизации работы данного шаблона.

Модернизация паттерна цепочка обязанностей. Исходя из вышеперечисленного, может возникнуть вопрос: как быть, если в цепочке некоторые из объектов имеют различные приоритеты. Допустим, у нас имеется сотрудник, до которого требуется донести информацию. Имеются различные способы реализации поставленной задачи, но, как было сказано ранее, у них будет различный приоритет. Пусть написать письмо будет иметь приоритет “несрочное”, отправить смс - приоритет “важное”, а позвонить - приоритет “срочное”. Для решения поставленной задачи мной предлагается ввести дополнительный класс, который будет содержать информацию о приоритетах, а в абстрактный класс AbstractNotifler добавить поле приоритета для определения важности того или иного

действия. Ознакомиться с модернизацией паттерна цепочка обязанностей можно на рис.2.

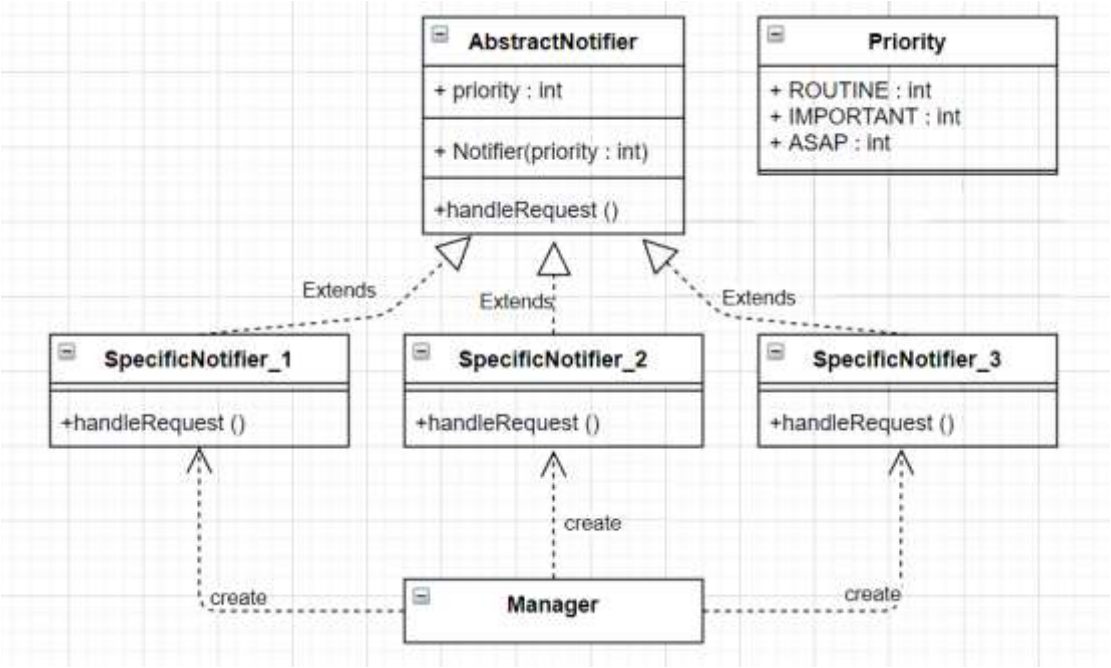

Рисунок 2 - Предложение по модернизации паттерна цепочка обязанностей

Обращая внимание на данную диаграмму классов, можно сделать вывод о том, что при создании объекта класса требуется передать в качестве аргумента число, которое будет являться приоритетом в нашем паттерне. В качестве аргумента для конструктора класса предлагается использовать заранее прописанные поля класса Priority (в данном примере, рис.2, это поля: ROUTINEE, IMPORTANT, ASAP), так как это сильно уменьшает вероятность возникновения ошибки в коде и позволяет разработчику повысить его читабельность, ведь появляется системность в подходе выбора приоритета для того или иного объекта класса.

В данной статье был предложен и рассмотрен новый подход к реализации паттерна «цепочка обязанностей», который превосходит оригинальный паттерн в системности написания и использования.

Литература

1. Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. Приемы объектно ориентированного проектирования. ДМК Пресс, 2007. — 368 с.
2. Вязовик, Н. А. Программирование на Java. 2-е изд. Москва: ИНТУИТ, 2016. — 603 с.
3. Бауэр, К. Java Persistence API и Hibernate. Перевод с английского Д. А. Зинкевич. Москва: ДМК Пресс, 2017. — 632 с.

4. Мацяшек, Л. А. Практическая программная инженерия на основе учебного примера. Перевод с английского А. М. Епанешникова, В. А. Епанешникова. 4-е изд. — Москва: Лаборатория знаний, 2020. — 959 с.
5. Балашова, И. Ю. Современные информационные технологии в проектировании программных систем и комплексов. Пенза: ПГУ, 2019. — 106 с.

Literature

1. E. Gamma, R. Helm, R. Johnson, D. Vlissides. Object-oriented design techniques. DMK Press, 2007 .-- 368 p.
2. Vyazovik, N. A. Programming in Java. 2nd ed. Moscow: INTUIT, 2016 .-- 603 p.
3. Bauer, K. Java Persistence API and Hibernate. Translated from English by D.A.Zinkevich. Moscow: DMK Press, 2017 .-- 632 p.
4. Matsyashek, LA Practical software engineering based on a training example. Translated from English by A. M. Epanshnikov, V. A. Epaneshnikov. 4th ed. - Moscow: Laboratory of Knowledge, 2020 .-- 959 p.
5. Balashova, I. Yu. Modern information technologies in the design of software systems and complexes. Penza: PSU, 2019 .-- 106 p.