Число кандидатов заранее не определено, а подбирать участников можно во время выполнения.

Паттерн ***наблюдатель*** определяет и отвечает за зависимости между объектами. Классический пример наблюдателя встречается в схеме модель/вид/контроллер языка Smalltalk, где все виды модели уведомляются о любых изменениях ее состояния.

Прочие паттерны поведения связаны с инкапсуляцией поведения в объекте и делегированием ему запросов. Паттерн ***стратегия*** инкапсулирует алгоритм объекта, упрощая его спецификацию и замену. Паттерн ***команда*** инкапсулирует запрос в виде объекта, который можно передавать как параметр, хранить в списке истории или использовать как-то иначе. Паттерн ***состояние*** инкапсулирует состояние объекта таким образом, что при изменении состояния объект может изменять поведение. Паттерн ***посетитель*** инкапсулирует поведение, которое в противном случае пришлось бы распределять между классами, а паттерн ***итератор*** абстрагирует способ доступа и обхода объектов из некоторого агрегата.

## 2.2. Паттерн «Цепочка обязанностей»

**Назначение**

Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

**Мотивация**

Рассмотрим контекстно-зависимую оперативную справку в графическом интерфейсе пользователя, который может получить дополнительную информацию по любой части интерфейса, просто щелкнув на ней мышью. Содержание справки зависит от того, какая часть интерфейса и в каком контексте выбрана. Например, справка по кнопке в диалоговом окне может отличаться от справки по аналогичной кнопке в главном окне приложения. Если для некоторой части интерфейса справки нет, то система должна показать информацию о ближайшем контексте, в котором она находится, например о диалоговом окне в целом.

Поэтому естественно было бы организовать справочную информацию от более конкретных разделов к более общим. Кроме того, ясно, что запрос на получение справки обрабатывается одним из нескольких объектов пользовательского интерфейса, каким именно – зависит от контекста и имеющейся в наличии информации.

Проблема в том, что объект, инициирующий запрос (например, кнопка), не располагает информацией о том, какой объект в конечном итоге предоставит справку. Необходим какой-то способ отделить кнопку-инициатор запроса от объектов, владеющих справочной информацией. Как этого добиться, показывает паттерн ***цепочка обязанностей***.

Идея заключается в том, чтобы разорвать связь между отправителями и получателями, дав возможность обработать запрос нескольким объектам. Запрос перемещается по цепочке объектов, пока один из них не обработает его. Первый объект в цепочке получает запрос и либо обрабатывает его сам, либо направляет следующему кандидату в цепочке, который ведет себя точно так же. У объекта, отправившего запрос, отсутствует информация об обработчике. Говорят, что у запроса есть анонимный получатель (implicit receiver).

Предположим, что пользователь запрашивает справку по кнопке `Print` (печать). Она находится в диалоговом окне `PrintDialog`, содержащем информацию об объекте приложения, которому принадлежит (диаграмма объектов показана на рисунке 2.1).

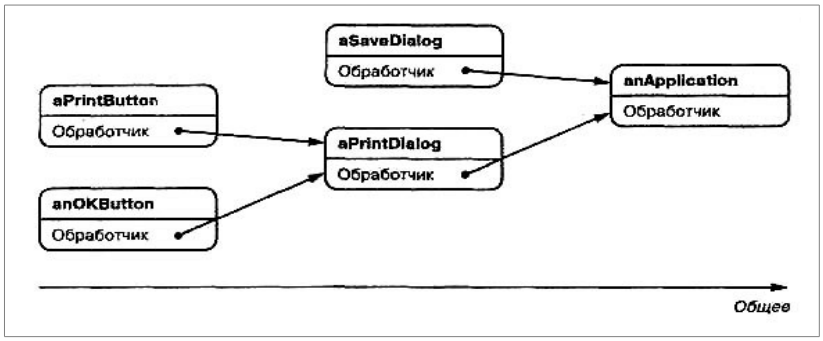


Рисунок 2.1 – Диаграмма объектов

В данном случае ни кнопка `aPrintButton`, ни окно `aPrintDialog` не обрабатывают запрос, он достигает объекта `anApplication`, который может его обработать или игнорировать. У клиента, инициировавшего запрос, нет прямой ссылки на объект, который его в конце концов обработает.

Диаграмма классов показана на рисунке 2.2. Чтобы отправить запрос по цепочке и гарантировать анонимность получателя, все объекты в цепочке имеют единый интерфейс для обработки запросов и для доступа к своему преемнику (следующему объекту в цепочке). Например, в системе оперативной справки можно было бы определить класс `HelpHandler` (предок классов всех объектов-кандидатов или подмешиваемый класс (mixin class)) с операцией `HandleHelp`. Тогда классы, которые будут обрабатывать запрос, смогут его передать своему родителю.

Для обработки запросов на получение справки классы `Button`, `Dialog` и `Application` пользуются операциями `HelpHandler`. По умолчанию операция `HandleHelp` просто перенаправляет запрос своему преемнику. В подклассах эта операция замещается, так что при благоприятных обстоятельствах может выдаваться справочная информация. В противном случае запрос отправляется дальше посредством реализации по умолчанию.

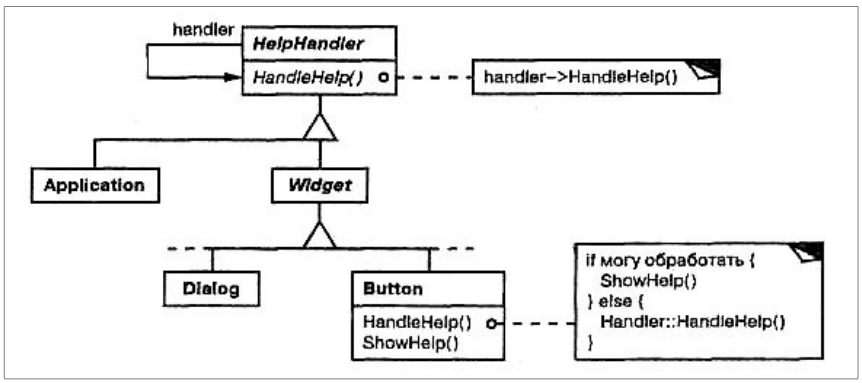


Рисунок 2.2 – Диаграмма классов

**Применимость**

Паттерн «Цепочка обязанностей» целесообразно применять, когда:

- есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- нужно отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
- набор объектов, способных обработать запрос, должен задаваться динамически.

**Структура**

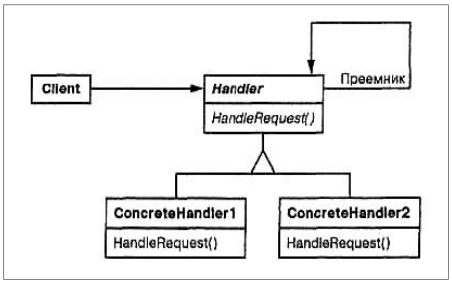


Рисунок 2.3 – Диаграмма классов паттерна
«Цепочка обязанностей»


Рисунок 2.4 – Типичная структура объектов

**Участники**

`Handler` (`HelpHandler`) – обработчик:

- определяет интерфейс для обработки запросов;
- реализует связь с преемником (необязательно);

`ConcreteHandler` (`PrintButton`, `PrintDialog`) – конкретный обработчик:

- обрабатывает запрос, за который отвечает;

- имеет доступ к своему преемнику;

- если `ConcreteHandler` способен обработать запрос, то так и делает, если не может, то направляет его своему преемнику;

`Client` – клиент:

- отправляет запрос некоторому объекту `ConcreteHandler` в цепочке.

**Отношения**

Когда клиент инициирует запрос, он продвигается по цепочке, пока некоторый объект `ConcreteHandler` не возьмет на себя ответственность за его обработку.

**Результаты**

Паттерн цепочка обязанностей имеет следующие достоинства и недостатки:

- ослабление связанности. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту – о структуре цепочки.

Таким образом, цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того чтобы хранить ссылки на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике;

- дополнительная гибкость при распределении обязанностей между объектами. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков;

- получение запроса не гарантировано. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

**Реализация**

При рассмотрении цепочки обязанностей следует обратить внимание на следующие моменты:

1) реализация цепочки преемников. Есть два способа реализовать такую цепочку:

- определить новые связи (обычно это делается в классе `Handler`, но можно и в `ConcreteHandler`);

- использовать существующие связи.

2) соединение преемников. Если готовых ссылок, пригодных для определения цепочки, нет, то их придется ввести. В таком случае класс `Handler`

не только определяет интерфейс запросов, но еще и хранит ссылку на преемника.

Следовательно у обработчика появляется возможность определить реализацию операции HandleRequest по умолчанию – перенаправление запроса преемнику (если таковой существует). Если подкласс ConcreteHandler не заинтересован в запросе, то ему и не надо замещать эту операцию, поскольку по умолчанию запрос как раз и отправляется дальше.

Определение базового класса HelpHandler, в котором хранится указатель на преемника, имеет вид:

```
class HelpHandler {
public:
     HelpHandler(HelpHandler* s) : _successor(s) { }
     virtual void HandleHelp();
private:
     HelpHandler* _successor;
};
void HelpHandler::HandleHelp () {
     if (_successor) {
          _successor->HandleHelp();
     }
}
```

3) представление запросов. Представлять запросы можно по-разному. В простейшей форме, например в случае класса HandleHelp, запрос жестко кодируется как вызов некоторой операции. Это удобно и безопасно, но переадресовывать тогда можно только фиксированный набор запросов, определенных в классе Handler.

Альтернатива – использовать одну функцию-обработчик, которой передается код запроса (скажем, целое число или строка). Так можно поддержать заранее неизвестное число запросов. Единственное требование состоит в том, что отправитель и получатель должны договориться о способе кодирования запроса.

Это более гибкий подход, но при реализации нужно использовать условные операторы для раздачи запросов по их коду. Кроме того, не существует безопасного с точки зрения типов способа передачи параметров, поэтому упаковывать и распаковывать их приходится вручную. Очевидно, что это не так безопасно, как прямой вызов операции.

Чтобы решить проблему передачи параметров, допустимо использовать отдельные объекты-запросы, в которых инкапсулированы параметры запроса.

Класс Request может представлять некоторые запросы явно, а их новые типы описываются в подклассах. Подкласс может определить другие параметры. Обработчик должен иметь информацию о типе запроса (какой именно подкласс Request используется), чтобы разобрать эти параметры.

Для идентификации запроса в классе Request можно определить функцию доступа, которая возвращает идентификатор класса. Вместо этого получатель мог бы воспользоваться информацией о типе, доступной во время выполнения, если язык программирования поддерживает такую возможность.

7
8

Приведем пример функции диспетчеризации, в которой используются объекты для идентификации запросов. Операция `GetKind()`, указанная в базовом классе `Request`, определяет вид запроса:

```
void Handler::HandleRequest (Request* theRequest) {
     switch (theRequest->GetKind()) {
     case Help:
     // привести аргумент к походящему типу
          HandleHelp((HelpRequest*) theRequest);
          break;
     case Print:
          HandlePrint((PrintRequest*) theRequest);
          // ...
          break;
     default:
     // ...
     break;
     }
}
```

Подклассы могут расширить схему диспетчеризации, переопределив операцию `HandleRequest`. Подкласс обрабатывает лишь те запросы, в которых заинтересован, а остальные отправляет родительскому классу. В этом случае подкласс именно расширяет, а не замещает операцию `HandleRequest`.

Подкласс `ExtendedHandler` расширяет операцию `HandleRequest()`, определенную в классе `Handler`, следующим образом:

```
class ExtendedHandler : public Handler {
public:
     virtual void HandleRequest(Request* theRequest);
     // . . .
};

void ExtendedHandler::HandleRequest (Request* theRequest) {
     switch (theRequest->GetKind()) {
     case Preview:
          // обработать запрос Preview
          break;
     default:
          // дать классу Handler возможность обработать
          // остальные запросы
          Handler::HandleRequest(theRequest);
     }
}
```

## 3. Порядок выполнения работы

3.1. Изучить назначение и структуру паттерна ***Цепочка обязанностей*** (выполнить в ходе самостоятельной подготовки).

3.2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна

***Цепочка обязанностей***. Для этого построить диаграмму классов, на диаграмме классов найти класс-клиент, запрос от которого необходимо передавать по цепочке объектов, и классы-получатели запросов, объекты которых целесообразно объединять в цепочку.

3.3. Выполнить перепроектирование системы, использовав паттерн ***Цепочка обязанностей***, изменения отобразить на диаграмме классов.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

3.5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу, выполнить ее тестирование и продемонстрировать ее работоспособность.

## 4. Содержание отчета

4.1. Цель работы.
4.2. Постановка задачи с описанием программного продукта, для которого проводится рефакторинг.
4.3. Словесное описание мотивации применения паттерна ***Цепочка обязанностей*** при проектировании данной системы.
4.4. UML-диаграммы классов с комментариями.
4.5. Текст программы.
4.6. Выводы по работе.

## 5. Контрольные вопросы

5.1. Для чего предназначены поведенческие паттерны проектирования?
5.5. Какие задачи решает паттерн «Цепочка обязанностей»?
5.6. Какие классы входят в состав паттерна «Цепочка обязанностей», каковы их обязанности?

Лабораторная работа №7

# Исследование способов применения порождающих паттернов проектирования при рефакторинге ПО

## 1.Цель работы

Исследовать возможность использования порождающих паттернов проектирования. Получить практические навыки применения порождающих паттернов при объектно-ориентированном проектировании и рефакторинге ПО.

## 2. Основные положения

### 2.1. Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс инстанцирования. Они помогут сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту.

Эти паттерны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Для порождающих паттернов актуальны две темы. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе.

Во-вторых, скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

### 2.2. Паттерн «Абстрактная фабрика»

**Назначение**

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

**Мотивация**