Паттерны проектирования

Паттерн проектирования описывает типичную задачу, а также принцип ее решения, причем таким образом, что это решение можно использовать повторно, ничего не изобретая заново.

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.: ил. (Серия «Библиотека программиста») ISBN 5-272-00355-1

Элементы паттерна проектирования

В общем случае паттерн состоит из четырех основных элементов:

имя — уникальное название паттерна;

задача — описание условий, когда следует применять паттерн;

решение — описание элементов дизайна, отношений между ними, функций каждого элемента;

результаты — это следствия применения паттерна и разного рода компромиссы.

Назначение паттернов проектирования

Ссылка на паттерн позволяет сразу описать проблему проектирования; ее решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. При этом выбор паттерна не подразумевает конкретный дизайн или реализацию, поскольку паттерн – это шаблон, применимый в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания элементов (в частности, классов и объектов).

Каркас приложения

Это набор взаимодействующих классов, составляющих повторно используемый дизайн для конкретного класса программ.

Отличия от паттернов.

1. Паттерны проектирования более абстрактны, чем каркасы.
2. Как архитектурные элементы, паттерны проектирования мельче, чем каркасы.
3. Паттерны проектирования менее специализированы, чем каркасы.

Каталог паттернов проектирования

Каталог содержит 23 паттерна.

Категории.

1. Порождающие 5.
2. Структурные 7.
3. Поведения 11.

Порождающие паттерны

Abstract Factory (абстрактная фабрика) — предоставляет интерфейс для создания объектов, конкретные классы которых неизвестны.

Singleton (одиночка) — гарантирует, что некоторый класс может иметь только один экземпляр.

Prototype (прототип) — описывает виды создаваемых объектов с помощью прототипа и создает новые объекты путем его копирования.

Factory Method (фабричный метод) — определяет интерфейс для создания объектов.

Builder (строитель) — отделяет конструирование объекта от представления, позволяя использовать один процесс конструирования для различных представлений.

Структурные

Adapter (адаптер) — преобразует интерфейс класса в некоторый другой интерфейс, ожидаемый клиентами.

Bridge (мост) — отделяет абстракцию от реализации, благодаря чему имеется возможность независимо изменять то и другое.

Decorator (декоратор) — динамически возлагает на объект новые функции.

Proxy (заместитель) — подменяет другой объект для контроля доступа к нему.

Composite (компоновщик) — группирует объекты в древовидные структуры для представления иерархий типа часть-целое.

Facade (фасад) — предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме.

Flyweight (приспособленец) — предназначен для эффективной поддержки большого числа мелких объектов.

Поведения

Chain of Responsibility (цепочка обязанностей) — позволяет избежать жесткой зависимости отправителя запроса от его получателя, связывает объекты-получатели в цепочку, по которой передается запрос.

Command (команда) — инкапсулирует запрос в виде объекта, позволяя тем самым параметризовывать клиентов типом запроса.

Interpreter (интерпретатор) — для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.

Iterator (итератор) — дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.

Mediator (посредник) — определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества.

Memento (хранитель) — позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.

Observer (наблюдатель) — Определяет между объектами зависимость типа один-ко-многим, так что при изменении состоянии одного объекта все зависящие от него получают извещение и автоматически обновляются.

State (состояние) — позволяет объекту варьировать свое поведение при изменении внутреннего состояния.

Strategy (стратегия) — определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого.

Template Method (шаблонный метод) — определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы.

Visitor (посетитель) — представляет операцию, которую надо выполнить над элементами объекта.

Самые простые и распространенные паттерны

— абстрактная фабрика;
— адаптер;
— компоновщик;
— декоратор;
— фабричный метод;
— наблюдатель;
— стратегия;
— шаблонный метод.

Пример фабричного метода

```
class Figure
{
 public:
    enum FigType {UNDEFINED, BAR, SIRCLE, TRIANGLE};
    static Figure* new_Figure(FigType figure_type);
};

Figure* Figure::new_Figure(FigType figure_type)
{
 ASSERT(figure_type>UNDEFINED && figure_type<=TRIANGLE);
 return figure_type == BAR      ? new FigureBar()    : (
        figure_type == SIRCLE   ? new FigureSircle() : (
        figure_type == TRIANGLE ? new FigureTriangle()
                        : NULL));
}
```

Пример синглетона

```
class Single
{
 public:
    static Single* get_instance();
 private:
    Single();
    Single(const Single &);
    Single &operator=(const Single&);
    ~Single();
};

Single* single_instance = NULL;

Single* Single::get_instance()
{
  If (!single_instance)
     single_instance = new Single();
  return single_instance;
}
```

Модифицированный пример синглетона

```
class Single
{
 public:
    static Single &get_instance();
 private:
    Single();
    Single(const Single &);
    Single &operator=(const Single&);
    ~Single();
    static Single instance;
};

Single Single::instance;

Single &Single::get_instance()
{
  return instance;
}
```

Механизмы повторного использования

Известны концепции объектов, интерфейсов, классов и наследования. Трудность в том, чтобы применить эти знания для построения гибких, повторно используемых программ. С помощью паттернов проектирования сделать это проще.

Механизмы.

- Наследование и композиция.
- Наследование и параметризованные типы.
- Агрегирование и осведомленность.

Наследование и композиция

Два наиболее распространенных приема повторного использования функциональности в объектно-ориентированных системах.

Повторное использование за счет порождения подкласса называют еще прозрачным ящиком (white-box reuse).

Композиция объектов — это альтернатива наследованию класса. В этом случае новую, более сложную функциональность мы получаем путем объединения или композиции объектов. Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. Такой способ повторного использования называют черным ящиком (black-box reuse).

Делегирование

С помощью делегирования композицию можно сделать столь же мощным инструментом повторного использования, как и наследование. При делегировании в процесс обработки запроса вовлечено два объекта: получатель поручает выполнение операций другому объекту – уполномоченному. Так же подкласс делегирует ответственность своему родительскому классу.

Унаследованная операция всегда может обратиться к объекту-получателю. Чтобы достичь того же эффекта для делегирования, получатель передает указатель на самого себя соответствующему объекту.

Наследование и параметризованные типы

Еще один (хотя и не в точности объектно-ориентированный) метод повторного использования имеющейся функциональности – это применение параметризованных типов (шаблоны C++). Данная техника позволяет определить тип, не задавая типы, которые он использует. Неспецифицированные типы передаются в виде параметров в точке использования.

Например, класс List (список) можно параметризовать типом помещаемых в список элементов. Чтобы объявить список целых чисел, вы передаете тип integer в качестве параметра параметризованному типу List. Для каждого типа элементов компилятор языка создаст отдельный вариант класса List.

Сравнение механизмов повторного использования

Многие задачи можно решить с помощью любого из этих трех методов. Чтобы параметризовать процедуру сортировки операцией сравнения элементов, мы могли бы сделать сравнение:

– операцией, реализуемой подклассами (применение паттерна шаблонный метод);

– функцией объекта, передаваемого процедуре сортировки (стратегия);

– аргументом шаблона в C++ или обобщенного типа в Ada, который задает имя функции, вызываемой для сравнения элементов.

Различия

Композиция объектов позволяет изменять поведение во время выполнения, но для этого требуются косвенные вызовы, что снижает эффективность.

Наследование разрешает предоставить реализацию по умолчанию, которую можно замещать в подклассах.

С помощью параметризованных типов допустимо изменять типы, используемые классом.

Ни наследование, ни параметризованные типы не подлежат модификации во время выполнения. Выбор того или иного подхода зависит от проекта и ограничений на реализацию.

Агрегирование и осведомленность

Агрегирование подразумевает, что один объект владеет другим или несет за него ответственность. В общем случае мы говорим, что объект содержит другой объект или является его частью. Агрегирование означает, что агрегат и его составляющие имеют одинаковое время жизни.

Осведомленность подразумевает, что объекту известно о другом объекте. Иногда осведомленность называют ассоциацией или отношением «использует». Осведомленные объекты могут запрашивать друг у друга операции, но они не несут никакой ответственности друг за друга. Осведомленность – это более слабое отношение, чем агрегирование; оно предполагает гораздо менее тесную связь между объектами.

Проектирование с учетом будущих изменений

Перепроектирование отражается на многих частях системы, поэтому непредвиденные изменения всегда оказываются дорогостоящими. Благодаря паттернам систему всегда можно модифицировать определенным образом. Каждый паттерн позволяет изменять некоторый аспект системы независимо от всех прочих, таким образом, она менее подвержена влиянию изменений конкретного вида.

Некоторые причины перепроектирования

1. При создании объекта явно указывается класс. Задание имени класса привязывает вас к конкретной реализации, а не к конкретному интерфейсу. Чтобы уйти от такой проблемы, создавайте объекты косвенно. Паттерны проектирования: абстрактная фабрика, фабричный метод, прототип.

2. Зависимость от конкретных операций. Задавая конкретную операцию, вы ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения. Паттерны проектирования: цепочка обязанностей, команда.

3. Зависимость от аппаратной и программной платформ. Паттерны проектирования: абстрактная фабрика, мост.

4. Зависимость от представления или реализации объекта. Если клиент «знает», как объект представлен, хранится или реализован, то при изменении объекта может оказаться необходимым изменить и клиента. Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель.

5. Зависимость от алгоритмов. Алгоритмы, вероятность изменения которых высока, следует изолировать. Паттерны проектирования: мост, итератор, стратегия, шаблонный метод, посетитель.

6. Сильная связанность. Для поддержки слабо связанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои. Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель.

7. Расширение функциональности за счет порождения подклассов. С каждым новым подклассом связаны фиксированные издержки реализации. Для определения подкласса необходимо знать устройство родительского класса. Композиция объектов и делегирование позволяют добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы. Паттерны проектирования: мост, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия.

8. Неудобства при изменении классов. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а его нет (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях. Паттерны проектирования: адаптер, декоратор, посетитель.