его использовать с учетом других проектных ограничений, каковы будут последствия применения метода. Поскольку любой проект в конечном итоге предстоит реализовывать, в состав паттерна включается пример кода на языке C++ (иногда на Smalltalk), иллюстрирующего реализацию.

Хотя, строго говоря, паттерны используются в проектировании, они основаны на практических решениях, реализованных на основных языках объектно-ориентированного программирования типа Smalltalk и C++, а не на процедурных (Pascal, C, Ada и т.п.) или объектно-ориентированных языках с динамической типизацией (CLOS, Dylan, Self). Мы выбрали Smalltalk и C++ из прагматических соображений, поскольку чаще всего работаем с ними и поскольку они завоевывают все большую популярность.

Выбор языка программирования безусловно важен. В наших паттернах подразумевается использование возможностей Smalltalk и C++, и от этого зависит, что реализовать легко, а что – трудно. Если бы мы ориентировались на процедурные языки, то включили бы паттерны наследование, инкапсуляция и полиморфизм. Некоторые из наших паттернов напрямую поддерживаются менее распространенными языками. Так, в языке CLOS есть мультиметоды, которые делают ненужным паттерн посетитель. Собственно, даже между Smalltalk и C++ есть много различий, из-за чего некоторые паттерны проще выражаются на одном языке, чем на другом (см., например, паттерн итератор).

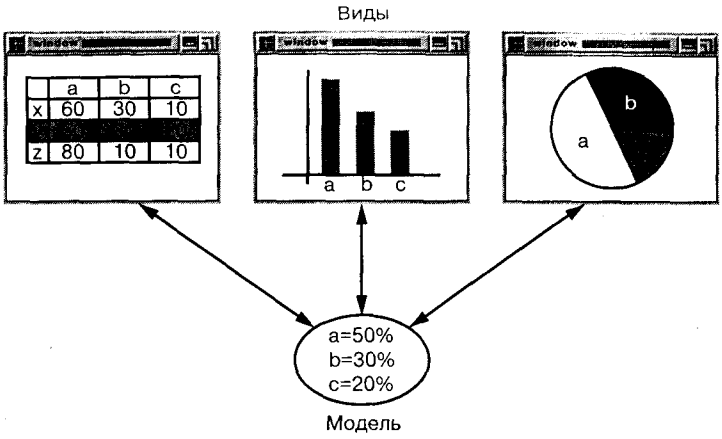
1.2. Паттерны проектирования в схеме MVC в языке Smalltalk

В Smalltalk-80 для построения интерфейсов пользователя применяется тройка классов модель/вид/контроллер (Model/View/Controller – MVC) [KP88]. Знакомство с паттернами проектирования, встречающимися в схеме MVC, поможет вам разобраться в том, что мы понимаем под словом «паттерн».

MVC состоит из объектов трех видов. *Модель* – это объект приложения, *вид* – экранное представление. *Контроллер* описывает, как интерфейс реагирует на управляющие воздействия пользователя. До появления схемы MVC эти объекты в пользовательских интерфейсах смешивались. MVC отделяет их друг от друга, за счет чего повышается гибкость и улучшаются возможности повторного использования.

MVC отделяет вид от модели, устанавливая между ними протокол взаимодействия «подписка/оповещение». Вид должен гарантировать, что внешнее представление отражает состояние модели. При каждом изменении внутренних данных модель оповещает все зависящие от нее виды, в результате чего вид обновляет себя. Такой подход позволяет присоединить к одной модели несколько видов, обеспечив тем самым различные представления. Можно создать новый вид, не переписывая модель.

На рисунке ниже показана одна модель и три вида. (Для простоты мы опустили контроллеры.) Модель содержит некоторые данные, которые могут быть представлены в виде электронной таблицы, гистограммы и круговой диаграммы.



Модель оповещает свои виды при каждом изменении значений данных, а виды обращаются к модели для получения новых значений.

На первый взгляд, в этом примере продемонстрирован просто дизайн, отделяющий вид от модели. Но тот же принцип применим и к более общей задаче: разделение объектов таким образом, что изменение одного отражается сразу на нескольких других, причем изменившийся объект не имеет информации о деталях реализации объектов, на которые он оказал воздействие. Этот более общий подход описывается паттерном проектирования **наблюдатель**.

Еще одно свойство MVC заключается в том, что виды могут быть вложенными. Например, панель управления, состоящую из кнопок, допустимо представить как составной вид, содержащий вложенные, – по одной кнопке на каждый. Пользовательский интерфейс инспектора объектов может состоять из вложенных видов, используемых также и в отладчике. MVC поддерживает вложенные виды с помощью класса `CompositeView`, являющегося подклассом `View`. Объекты класса `CompositeView` ведут себя так же, как объекты класса `View`, поэтому могут использоваться всюду, где и виды. Но еще они могут содержать вложенные виды и управлять ими.

Здесь можно было бы считать, что этот дизайн позволяет обращаться с составным видом, как с любым из его компонентов. Но тот же дизайн применим и в ситуации, когда мы хотим иметь возможность группировать объекты и рассматривать группу как отдельный объект. Такой подход описывается паттерном **компоновщик**. Он позволяет создавать иерархию классов, в которой некоторые подклассы определяют примитивные объекты (например, `Button` – кнопка), а другие – составные объекты (`CompositeView`), группирующие примитивы в более сложные структуры.

MVC позволяет также изменять реакцию вида на действия пользователя. При этом визуальное представление остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать всплывающие меню вместо командных клавиш. MVC инкапсулирует механизм определения реакции в объекте `Controller`. Существует иерархия классов контроллеров, и это позволяет без труда создать новый контроллер как вариант уже существующего.

Вид пользуется экземпляром класса, производного от `Controller`, для реализации конкретной стратегии реагирования. Чтобы реализовать иную стратегию, нужно просто подставить другой контроллер. Можно даже заменить контроллер вида во время выполнения программы, изменив тем самым реакцию на действия пользователя. Например, вид можно деактивировать, так что он вообще не будет ни на что реагировать, если передать ему контроллер, игнорирующий события ввода.

Отношение вид-контроллер – это пример паттерна проектирования стратегия. Стратегия – это объект для представления алгоритма. Он полезен, когда вы хотите статически или динамически подменить один алгоритм другим, если существует много вариантов одного алгоритма или когда с алгоритмом связаны сложные структуры данных, которые хотелось бы инкапсулировать.

В MVC используются и другие паттерны проектирования, например фабричный метод, позволяющий задать для вида класс контроллера по умолчанию, и декоратор для добавления к виду возможности прокрутки. Но основные отношения в схеме MVC описываются паттернами наблюдатель, компоновщик и стратегия.

1.3. Описание паттернов проектирования

Как мы будем описывать паттерны проектирования? Графических обозначений недостаточно. Они просто символизируют конечный продукт процесса проектирования в виде отношений между классами и объектами. Чтобы повторно воспользоваться дизайном, нам необходимо документировать решения, альтернативные варианты и компромиссы, которые привели к нему. Важны также конкретные примеры, поскольку они позволяют увидеть применение паттерна.

При описании паттернов проектировании мы будем придерживаться единого принципа. Описание каждого паттерна разбито на разделы, перечисленные ниже. Такой подход позволяет единообразно представить информацию, облегчает изучение, сравнение и применение паттернов.

Название и классификация паттерна

Название паттерна должно четко отражать его назначение. Классификация паттернов проводится в соответствии со схемой, которая изложена в разделе 1.5.

Назначение

Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью.

Известен также под именем

Другие распространенные названия паттерна, если таковые имеются.

Мотивация

Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна.