Глава 8[1]

Основы объектно-ориентированного представления программных систем

Восьмая глава вводит в круг вопросов объектно-ориентированного представления программных систем (ПС). В этой главе рассматриваются: абстрагирование понятий проблемной области, приводящее к формированию классов; инкапсуляция объектов, обеспечивающая скрытность их характеристик; модульность как средство упаковки набора классов; особенности построения иерархической структуры объектно-ориентированных систем. Последовательно обсуждаются объекты и классы как основные строительные элементы объектно-ориентированного ПО. Значительное внимание уделяется описанию отношений между объектами и классами. Кроме того, здесь определяются базовые понятия унифицированного языка моделирования.

Принципы объектно-ориентированного представления программных систем

Рассмотрение любой сложной системы требует применения техники декомпозиции — разбиения на составляющие элементы. Известны две схемы декомпозиции: алгоритмическая декомпозиция и объектно-ориентированная декомпозиция.

В основе алгоритмической декомпозиции лежит разбиение по действиям — алгоритмам. Эта схема представления применяется в обычных (процедурных) ПС.

Объектно-ориентированная декомпозиция обеспечивает разбиение по автономным лицам — объектам реального (или виртуального) мира. Эти лица (объекты) — более «крупные» элементы, каждый из них несет в себе и описания действий, и описания данных.

[1] Глава 7 «Классические методы проектирования» доступна для скачивания на сайте издательства (piter.com).

Объектно-ориентированное представление ПС основывается на принципах абстрагирования, инкапсуляции, модульности и иерархической организации. Каждый из этих принципов не нов, но их совместное применение рассчитано на проведение объектно-ориентированной декомпозиции. Это определяет модификацию их содержания и механизмов взаимодействия друг с другом. Обсудим данные принципы [38, 52, 63, 83, 88, 90].

Абстрагирование

Аппарат абстракции — удобный инструмент для борьбы со сложностью реальных систем. Создавая понятие в интересах какой-либо задачи, мы отвлекаемся (абстрагируемся) от несущественных характеристик конкретных объектов, определяя только существенные характеристики. Например, в абстракции «часы» мы выделяем характеристику «показывать время», отвлекаясь от таких характеристик конкретных часов, как форма, цвет, материал, цена, изготовитель.

Итак, абстрагирование сводится к формированию абстракций. Каждая абстракция фиксирует основные характеристики объекта, которые отличают его от других видов объектов и обеспечивают ясные понятийные границы.

Абстракция концентрирует внимание на внешнем представлении объекта, позволяет отделить основное в поведении объекта от его реализации. Абстракцию удобно строить путем выделения обязанностей объекта.

Пример 8.1. Физический объект — один из трех датчиков скорости, устанавливаемый на борту летательного аппарата (ЛА) по одному из трех направлений. Создадим его абстракцию. Для этого сформулируем обязанности датчика:

- знать проекцию скорости ЛА в заданном направлении;
- показывать текущую скорость;
- подвергаться настройке.

Анализируя обязанности, делаем следующие выводы: абстракция должна содержать два действия (показывать текущую скорость, настраивать) и два поля данных (скорость, направление). Описание абстракции датчика запишем как класс на языке Java:

```
public class ДатчикСкорости {
    // поля данных
    private Скорость скорость;
    private Направление направление;
    // конструктор, задает направление датчика
    ДатчикСкорости(Направление направление) {
        this.направление = направление;
    }
    // отображение текущей скорости
    public Скорость текущаяСкорость() {
        return скорость;
    }
    // настройка датчика
    public void настраивать(Скорость действитСкорость) {
        this.скорость = действитСкорость;
    }
}
```

ПРИМЕЧАНИЕ

Напомним, запись private Скорость скорость означает поле данных скорость с типом Скорость, причем это поле скрыто от клиента.

Подробности выполнения обязательств абстракцией ДатчикСкорости зависят от ее внутреннего представления и не должны интересовать внешних клиентов. Эта информация является секретом класса и реализуется его закрытой (приватной) частью совместно с содержанием действий-методов.

Класс ДатчикСкорости еще не объект. Собственно датчики — это его экземпляры, и их нужно создать, прежде чем с ними можно будет работать. Например, можно написать так:

```
ДатчикСкорости датчикПродольнойСкорости = new ДатчикСкорости();
ДатчикСкорости датчикПоперечнойСкорости = new ДатчикСкорости();
ДатчикСкорости датчикНормальнойСкорости = new ДатчикСкорости();
```

Итак, абстракция отражает суть объекта, опуская второстепенные детали. Выбор и правильное описание набора абстракций для заданной предметной области является главной задачей объектно-ориентированной разработки.

Инкапсуляция

Инкапсуляция и абстракция — взаимодополняющие понятия: абстракция выделяет внешнее поведение объекта, а инкапсуляция содержит и скрывает реализацию, которая обеспечивает это поведение. Инкапсуляция достигается с помощью информационной закрытости. Обычно скрывается структура данных объектов и реализация их методов.

Инкапсуляция является процессом разделения элементов абстракции на секции с различной видимостью. Инкапсуляция служит для отделения интерфейса абстракции от ее реализации.

Пример 8.2. Продолжим заниматься предметной областью системы управления летательного аппарата. Применим принцип разделения понятий (separation of concerns): разобравшись с одной абстракцией, переходим к обсуждению физического объекта «Регулятор скорости». Как и в прошлый раз, определим его обязанности.

Обязанности регулятора:

- знать свой номер, а также порт, через который он получает команды управления;
- включаться;
- выключаться;
- увеличивать скорость по своему каналу;
- уменьшать скорость по своему каналу;
- отображать свое состояние.

Исходя из обязанностей, описание класса РегуляторСкорости примет вид:

```
public class РегуляторСкорости {
    enum Режим {
        Увеличение, Уменьшение
    }
    private Размещение номер;
```

```
    private Режим состояние;
    private Порт управление;

    public РегуляторСкорости(Размещение номер, Режим состояние, Порт управление) {
        this.номер = номер;
        this.управление = управление;
        ...
    }
    public void включить() {
        ...
    }
    public void выключить() {
        ...
    }
    public void увеличитьСкорость() {
        ...
    }
    public void уменьшитьСкорость() {
        ...
    }
    public Режим опросСостояния() {
        return состояние;
    }
}
```

Три поля номер, состояние, управление формулируют инкапсулируемое представление данных класса РегуляторСкорости. При попытке клиента получить к этим полям доступ фиксируется семантическая ошибка.

Полное инкапсулированное представление класса РегуляторСкорости включает описание реализаций его методов, которое для краткости здесь опущено.

Модульность

В языках C++, Delphi Pascal, Ada 2005, Java и C# абстракции классов и объектов формируют логическую структуру системы. При производстве физической структуры эти абстракции помещаются в модули. В больших системах, где сотни классов, модули помогают управлять сложностью. Модули служат физическими контейнерами, в которых объявляются классы и объекты логической разработки.

Модульность определяет способность системы подвергаться декомпозиции на ряд сильно связанных и слабо сцепленных модулей.

Общая цель декомпозиции на модули: уменьшение сроков разработки и стоимости ПС за счет выделения модулей, которые проектируются и изменяются независимо. Каждая модульная структура должна быть достаточно простой, чтобы быть полностью понятой. Изменение реализации модулей должно проводиться без знания реализации других модулей и без влияния на их поведение. Правильно выбрать модули почти так же сложно, как выбрать правильный набор абстракций.

Определение классов и объектов выполняется в ходе логической разработки, а определение модулей — в ходе физической разработки системы. Эти действия сильно взаимосвязаны, осуществляются итеративно.

В некоторых языках программирования (например, в Smalltalk) модулей нет, и единственной физической единицей декомпозиции считается класс. Во многих

других языках, включая Delphi Pascal, C++ и Ada 2005, модуль является самостоятельной языковой конструкцией, обеспечивающей создание отдельных проектных решений. В языке Java мощным средством обеспечения модульности являются внутренние классы и пакеты.

Пример 8.3. Пусть имеется несколько классов управления полетом летательного аппарата (ЛА) — класс угловой стабилизации ЛА и класс управления движением центра масс ЛА. Нужно создать модуль, чье назначение — собрать все эти классы. Возможны два способа.

1. Встраивание классов управления непосредственно в класс-контейнер:

```
public class УпрПолетом {
    private class УгловаяСтабилизация {
        ...
    }

    private class ДвиженЦентраМасс {
        ...
    }
}
```

2. Размещение в пакете:

```
package упрполетом;

    class УгловаяСтабилизация {
        ...
    }

    class ДвиженЦентраМасс {
        ...
    }
```

Иерархическая организация

Мы рассмотрели три механизма для борьбы со сложностью:

- абстракцию (она упрощает представление физического объекта);
- инкапсуляцию (закрывает детали внутреннего представления абстракций);
- модульность (дает путь группировки логически связанных абстракций).

Прекрасным дополнением к этим механизмам является иерархическая организация — формирование из абстракций иерархической структуры. Определением иерархии в проекте упрощается понимание проблем заказчика и их реализация — сложная система становится обозримой человеком.

Иерархическая организация задает размещение абстракций на различных уровнях описания системы.

Двумя важными инструментами иерархической организации в объектно-ориентированных системах являются:

- структура из классов («*is a*»-иерархия);
- структура из объектов («*part of*»-иерархия).

Чаще всего «*is a*»-иерархическая структура строится с помощью наследования. Наследование определяет отношение между классами, где класс разделяет